Foundational Semantics of Dynamically Scheduled Attribute Grammar Evaluation

By

Scott J. H. Buckley

A thesis submitted to Macquarie University
for the degree of Doctor of Philosophy
Department of Computing
February 2021

Typeset in LaTeX 2_ε.

Except where acknowledged in the customary manner, the material presented in this thesis is, to the best of my knowledge, original and has not been submitted in whole or part for a degree in any university.

Scott J. H. Buckley

Acknowledgements

They say it takes a village to raise a child. I'd say that's true here, except "raise" is more "put up with while he slowly battles his way through a PhD". My success in this endeavour[1] is the result of ridiculous amounts of support from dozens of people.

To my supervisors Tony and Matt, thank you for your patience and guidance. You turned something impossible into a reality.

To my parents Sue and Mike, thank you for your patience and endless support. You are the reason I believed in myself enough to make it to the end of this.

I owe a great deal to many dear friends, for hundreds of hours of listening, discussion, and advice. None have been more present and supportive in this regard than Ian. Thank you, Ian.

And thank you to the rest that I can't name here, who were there for beers and years. This achievement is thanks to all of you.

[1]The content of the following pages is a rock that I pushed up a hill for a while. The real value was in learning how to push rocks.

Abstract

The similarities and differences between attribute grammar systems are obscured by their implementations. A formalism that captures the essence of such systems would allow for equivalence, correctness, and other analyses to be formally framed and proven. We present Saiga, a core language and small-step operational semantics that precisely captures the fundamental concepts of the evaluation of dynamically scheduled attribute grammars. We also present and discuss evaluation semantics for reference, parameterised, cached, and higher-order attribute grammars. Saiga's utility is demonstrated through proofs about the system's operation, equivalence proofs between distinct Saiga attribute grammar programs, and "step count" comparisons between such programs. The language, semantics and proofs have been mechanised in Lean.

Contents

Here, I'm gonna do a thing.

ChocoTaco

1 Introduction

A programming language is an interface between a human and a computer. A human encodes some human-readable representation of a program into a source file, typically in text form. This source file is passed either to an interpreter, which performs some unknowable magic and executes the task described by the source file, or to a compiler which performs some similarly unknowable magic and produces an hieroglyphic version of that program, which somehow can be executed. Magic.

Of course this 'magic' is knowable and, once known, is no longer magical. The structure and implementation of compilers is a well-researched area, with dozens of papers published every year at multiple international conferences (POPL, PLDI, SPLASH to name a few). Most computer science departments offer at least one undergraduate unit in language design. Once an *ad hoc* endeavour, compilers are now built using various formalisms designed specifically for the task. Language workbenches [1] bring all of these formalisms together into a single software suite designed to aid in language design and implementation.

When teaching compiler design to undergraduates, we teach that there are three major stages to a compiler: structuring, translation, and encoding. Structuring involves lexical and syntactic analysis – scanning, tokenising, parsing, and tree construction. This is the part of the compiler that reads some textual representation of a program and produces a tree representing that program. Translation involves semantic analysis – understanding the semantics of the program tree, which includes type and name analysis. Translation also includes the construction of a target program tree – the program tree that represents the compiled program. Encoding involves code generation and assembly – the process of encoding a target program tree into a product that is ready for some machine (real or virtual) to execute.

While every stage of a compiler is important, in this thesis we focus on *semantic analysis*; the process of analysing a source program tree to determine if it is valid, and to prepare extra semantic information for translation or execution, such as name bindings. Since semantic analysis is a specialised task, a specialised formalism called *attribute grammars* (AGs) has been developed to make semantic analysis easier and more consistent. AGs have been around since the late 60s [2], and are heavily researched. The formalism allows for the productions of a context-free grammar to be annotated with relational equations.

Since the specification of an attribute grammar is concerned only with the relationships between nodes and values, and is not concerned with the order of evaluation, attribute grammars are considered a declarative formalism. A large portion of attribute grammar research, especially in the first few decades of its existence, was regarding *evaluation traversals*; generating algorithms that could traverse any conforming tree in a finite number of passes, guaranteeing that every attribute could be evaluated on every node in the tree. The strategy of analysing an attribute grammar and generating a finite evaluation traversal algorithm is called *static scheduling*. This process also involves rejecting attribute grammars that the scheduler cannot determine a schedule for – whether no such schedule exists, or if it does not fit within the bounds set out by the scheduler.

An alternative to static scheduling is *dynamic scheduling*, where an evaluator uses run-time information, such as the structure of the tree being decorated, to determine an evaluation order [3]. This can take the form of on-demand evaluation, where attributes are only evaluated once their values are required. The disadvantage of this approach is that cyclic dependencies are often not discovered until run-time, but this also allows attribute grammar programs to be written outside of the various *attribute grammar classifications* that place restrictions on their specification. Dynamic scheduling also allows for partial evaluation – with the rise of development environments working alongside compilers for instant error-reporting and diagnostics, it is valuable to be able to evaluate only the attributes that are requested; not every attribute in a tree all at once[1].

Dynamic scheduling gives an *attribute grammar platform* the freedom to implement evaluation in a variety of new ways. These lifted restrictions also allow new attribute features and notations to exist, which would not integrate so neatly into statically scheduled platforms. *Reference attribute grammars* [4], for example, allow an attribute's value to depend on the value of an arbitrarily distant node, which can be selected during attribute evaluation. Reference attribute grammars can be implemented in statically scheduled attribute grammar systems [5], but exist more naturally in a dynamically scheduled system.

The freedom provided by dynamic scheduling has led to more variety in attribute grammar platforms. Some platforms are deeply embedded in a general purpose programming language, as in the instance of Kiama [6]. Some platforms stand alone and parse an attribute grammar specification written in a specialised language, as in the instance of Silver [7]. This variety is interesting, as two platforms that provide similar functionality can be expressed in vastly different ways, allow different notation possibilities, and implement different particular evaluation strategies.

The problem created by this variety of approaches is that the essence of dynamically schedule attribute grammar evaluation is obfuscated by its many implementations. It is difficult to reason about any attribute grammar algorithm without implementing it in some attribute grammar platform, and being bound to the semantics of that platform's particular implementation, which may differ significantly to those of another platform.

1.1 Contributions

In this thesis we present a calculus that captures what is common between dynamically scheduled attribute grammar evaluators. We present a strategy for defining attributes that

[1]Some static schedulers also allow for incremental evaluation, which can provide similar functionality for IDEs.

is flexible enough to mimic a variety of notations while exhibiting consistent evaluation semantics. Our calculus is specific to the domain of dynamically scheduled attribute grammar evaluation but general enough to represent all common notations and behaviours in the domain, and simple enough to allow proofs of evaluation to be easily expressed.

Specifically, the contributions of this thesis are as follows.

1. A **calculus** which captures the fundamental semantics of dynamically-scheduled attribute grammar evaluation, without being obscured by a general purpose language or by a particular attribute grammar platform's implementations.

2. This calculus is defined in the form of an **expression language**, **type rules**, and a **small-step operational semantics**.

3. A key feature of the semantics presented is **the context function**, which is a very flexible framework for defining attribute grammar equations that is notation agnostic.

4. We begin with a core calculus which we extend by implementing some common and representative attribute grammar **extensions**, including parameterised attributes, attribute caching, and higher order attributes.

5. Our calculus represents a framework for reasoning about and comparing the behaviour of attribute grammar programs, aided by a set of **metatheoretic properties**, for which we provide comprehensive proofs.

6. The calculus itself, as well as proofs for the majority of these metatheoretic properties, are **mechanised** in Lean [8].

7. We **demonstrate the utility** of these techniques through analysis of a real-world scale problem: comparing two different approaches to name and type analysis for Featherweight Java [9], translated directly from implementations in two different real-world attribute grammar platforms.

8. This comparison is primarily a proof that two particular attributes always evaluate to the same value, but we also demonstrate Saiga's facility for **quantitative analysis** by comparing the number of evaluation steps taken for a particular computation.

1.2 Outline

This thesis is organised as follows: first, in Chapter 2, we discuss attribute grammars in general, including their history, and review some modern attribute grammar platforms. We examine common attribute grammar extensions, giving examples of how these extensions can be used in Silver, JastAdd, and Kiama, three modern dynamically scheduled attribute grammar platforms. The extensions we focus on are reference attributes, parameterised attributes, attribute caching, higher order attributes, and attribute forwarding. We briefly discuss circular and collection attributes.

In Chapter 3 we present Saiga, our calculus for dynamically scheduled attribute grammar evaluation. Built on top of a simple functional calculus, we present an expression language, type rules, and a step relation that describes small step operational semantics. We discuss the particulars of Saiga's implementation and the design decisions behind the calculus, and present an example that models name analysis for a simple language.

In Chapter 4 we show how Saiga can be extended with parameterised attributes and attribute caching. We explore motivating examples, presenting an alternative approach to name analysis, and showing how caching can impact the performance of an evaluation. In Chapter 5 we show how higher order attributes can be incorporated into the calculus, and how Saiga's flexible strategy for defining attribute equations give attribute forwarding for free.

Chapter 6 is concerned with proving properties of our calculus, including simple properties such as determinism and progress, as well as more involved properties such as cache irrelevance. In Chapter 7 we present an example of Saiga's use in analysing real-world attribute grammar programs by translating two different implementations of name and type analysis for Featherweight Java into Saiga specifications and proving their outputs to be equivalent. These specifications were taken as direct translations from Kiama and JastAdd programs written by the platform maintainers, and each use a different approach to name analysis as well as a different notational standard for defining their attributes. Saiga's ability to capture both programs – with their distinct individual notations – into the same low-level formalism demonstrates the strength of our approach. We also demonstrate Saiga's ability to perform quantitative analysis by comparing the number of evaluation steps taken by each approach.

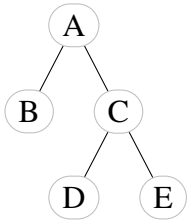
In Chapter 8 we briefly discuss the mechanisation of Saiga in Lean, and how the task of mechanising our calculus acted as a useful sanity check for many of our theorems. We demonstrate the strategies used to encode our semantics, and discuss the challenges that come with mechanising our particular approach to equation selection and higher order evaluation. In Chapter 9 we summarise our findings and discuss directions for future work.

The problem with the world is that everyone is a few drinks behind.

Humphrey Bogart

2 Background

You have a Christmas tree. You are a terrible nerd, so your Christmas tree is a binary tree.



Your tree exists, and it is the appropriate shape, but it needs some decoration. You synthesise some green integers in the lab at work and bring them home. You search the attic and find a box of red integers you inherited from your uncle Don. You will decorate your tree with these festive integers.

Due to a terminal personality flaw, you are unable to enjoy decorations that do not follow strict decoration rules. You decide that every node on your tree must have exactly one red and one green integer (otherwise your tree would look ridiculous). You want the red integer on the top node to be zero, and every other node's red integer to be one greater than the red integer directly above it on the tree. This way the red integers' values will increase as they get further down the tree.

You want every node's green integer to be the sum of the ASCII values of the labels of its direct children, plus the value of its own red integer. This seems obscure enough to provoke questions from visitors, providing opportunities for you to condescendingly explain your decoration formulae. You set down your boxes of red and green integers by the tree, ready to decorate. But where do you begin?

The integers you are using to decorate your tree are *attributes*. The formulae you have decided on to dictate the values of these attributes are *attribute equations*. The red integer is an *inherited attribute*, as its value is determined from data higher in the tree. The green integer is a *synthesised attribute*, as its value is determined from data lower in the tree. The relationships that exist between the undecorated nodes of your original tree are a *context-free grammar*, albeit a very simple one. The entirety of the scheme you have cooked up to define your tree's decoration is an *attribute grammar*.

2.1 Early Attribute Grammars

Before the birth of the attribute grammar, language designers found themselves in a difficult position. A context-free grammar could define what some text *means*, in terms of the program it represents, or why it does not represent a program at all. What was missing was a way to define what some *program* means, or if it is not valid, and why.

In a discussion with Peter Wegner in 1967, Donald Knuth spoke of formal semantics being defined by a combination of synthesised string-valued attributes, referencing an as-yet unpublished paper by Lewis *et al.* [10]. This approach, however, lacked the necessary contextual information required for things like name binding which are so common in programming language semantics. It was Wegner's idea to combine bottom-up (synthesised) definitions with top-down (inherited) definitions [11].

The following year, Knuth published the seminal paper on attribute grammars, titled "Semantics of context-free languages" [2]. The primary issue at this stage was defining synthesised and inherited attributes in a non-circular way, so that it is possible to derive a tree traversal that can compute values for all attributes. By 1990, "Attribute Grammars [had] turned into one of the most fundamental formalisms of modern Computer Science", according to Deransart *et al.* in their book "Attribute Grammars" [12], which cites about 600 attribute grammar-related research papers.

Attribute grammars are built as an extension of context-free grammars, defining both synthesised and inherited attribute equations alongside production rules, as in the following example, which specifies the Christmas tree decoration problem posed above.

Production	**Semantics**
P ::= N	N.red = 0
N_0 ::= Int N_1 N_2	N_0.green = N_1.int + N_2.int + N_0.red
	N_1.red = N_0.red + 1
	N_2.red = N_0.red + 1
N ::= Int	N.green = N.red

Above we provide a simple attribute grammar program. The two non-terminals defined are P and N. Alongside each production rule, a set of *attribute definitions* are provided. `red` is an *inherited* attribute, as its value is defined in its parent's production rule, which means that its data is influenced by values above it in the tree. `green` is a *synthesised* attribute, as its value is defined in its own production rule, which means that its data is influenced by values below it in the tree.

In the Christmas tree example, there is a simple *traversal* we can follow to assign red and green values to all nodes. We start at the top, and give red values to every node on the way to the bottom. We then start at the bottom, and give green values to every node on the way to the top. In fact, we can use this traversal to assign values to red and green for any tree that conforms to the above grammar.

It is the job of an *attribute grammar evaluator* to come up with this traversal, and to generate code that can decorate any given tree with attribute values that conform to their equations. However there are some attribute grammar programs that cannot be satisfied; if a circular dependency exists between attributes, there may be no values that satisfy their equations. For example, if we redefined the *attribute equation* for `red` attached to the production rule for P as shown below, the attribute grammar program would become circular,

and Knuth's algorithm would not be able to derive a tree traversal. Consider the Christmas tree grammar shown above, where the first production is replaced by the following production and attribute definition.

```
P ::= N  N.red = N.green
```

For a node created by the `P` production, its `red` value is determined by its `green` value. However, a node's `green` value is also determined by its `red` value. Therefore the value of `red` and `green` is undefined for such a node. This kind of logic cycle must be avoided for an attribute grammar program to be evaluable.

2.1.1 Statically Scheduled Attribute Grammar Categories

Over the years, a number of *attribute grammar categories* have developed, each with its own strengths and evaluation strategies.

Purely Synthesised Attribute Grammars

In his seminal attribute grammars paper [2], Knuth introduced *purely synthesised attribute grammars*. For an attribute to be purely synthesised, no inherited attributes can be present. It is possible to build a time-optimal system that combines both syntactic and semantic analysis under this category [12, 13].

Purely synthesised attribute grammars are able to encode the same semantics available to other categories, as all data in a tree is indirectly available to the root node through synthesis alone. However, "such a restriction leads to a very awkward and unnatural definition of semantics" [2].

Non-Circular Attribute Grammars

The attribute grammars that were the focus of Knuth's first paper [2] were *non-circular attribute grammars*: attribute grammars that did not exhibit any dependency cycles between attributes. Knuth provided an algorithm for testing an attribute grammar for circularity in polynomial time, which turned out to have an error. He issued a correction [14] that shows his algorithm was in fact exponential in complexity. It is important for static attribute grammar evaluators to be able to detect circularities statically, as it is their job to generate an algorithm that can decorate any input tree.

Strongly (or Absolutely) Non-Circular Attribute Grammars

Kennedy *et al.* introduced *absolutely non-circular attribute grammars* [15] in 1976. Courcelle *et al.* introduced *strongly non-circular attribute grammars* [16] in 1982, without prior knowledge of Kennedy's work. The two categories are considered equivalent [12, 17], and Courcelle *et al.* also reference *benign attribute grammars* [18] as a similar category.

Strongly non-circular attribute grammars are a subset of non-circular attribute grammars which consider a pessimistic view of attribute dependencies, considering an attribute's dependencies to be a union of all its productions' dependencies, even when in reality they may be disjoint between productions. Strong non-circularity is decidable in polynomial time,

while deciding if an attribute grammar is benign or non-circular is exponential [16]. Interestingly, the algorithm for detecting strong non-circularity is similar to the original flawed algorithm for detecting non-circularity by Knuth in 1968 [11, 19].

As strongly non-circular attribute grammars are a subset of non-circular attribute grammars, some valid non-circular attribute grammar programs are excluded. However, it has been found in practice that these restrictions do not limit expressibility in real-world applications [16, 17].

Kennedy *et al.* give an iterative algorithm for evaluating strongly non-circular attribute grammars, while Courcelle *et al.* take a recursive approach, showing that any strongly non-circular attribute grammar is equivalent to a set of functions recursively defined over the structure of a tree [16]. Jourdan takes the idea of recursive evaluation further [17] and shows that strongly non-circular attribute grammars (and indeed any attribute grammar [3]) can be evaluated dynamically in an "on-demand" manner, birthing the *dynamically scheduled attribute grammar* that this thesis is focused on.

Ordered Attribute Grammars

In 1980, Kastens presented *ordered attribute grammars* [20]. An attribute grammar is considered ordered if a partial order exists over all attributes such that the attributes can be evaluated in an order conforming to that partial order, on any tree conforming to the grammar. Kastens provides an algorithm for checking whether an attribute grammar is ordered, with a polynomial time complexity over the size of the grammar. The evaluator is also generated in polynomial time, as the test is constructive.

Ordered attribute grammars are a subset of strongly non-circular attribute grammars [21]. Kastens introduces the concept of visit-sequences, which is derived from the attributes' partial order, and describes the order in which a subtree can be traversed to evaluate its attributes, yielding the attribute evaluation algorithm.

Other Categories

Numerous other attribute grammar categories exist. Engelfriet *et al.* presents *l-ordered attribute grammars* [22], based on an early version of ordered attribute grammars by Kastens [23]. Filé presents *doubly non-circular attribute grammars* [24], a supercategory of *l*-ordered attribute grammars, and Barbar shows that the *partially-ordered attribute grammars* [25] are a superset of doubly non-circular attribute grammars.

2.2 Dynamically Scheduled Attribute Grammars

In recent years, the majority of the research in attribute grammars has focused on dynamically scheduled evaluation, where run-time information such as the particular tree being decorated is used to determine which attributes are to be evaluated, and in what order [6]. The concept of dynamic attribute grammar evaluation was first introduced by Jourdan [3] in 1984, who presented an evaluation method which discovered circular dependencies at runtime, and performed "evaluation by need"; a lazy approach to attribute evaluation.

One advantage provided by demand-driven evaluation is non-strict conditionals. While some statically scheduled attribute grammar systems allow for similar behaviour [26], Jourdan shows that demand-driven evaluation allows only one branch of a conditional to be

evaluated, allowing evaluation of attribute grammar programs that would not be considered non-circular in the original sense[1].

Note that not all dynamically scheduled attribute grammars use demand-driven or lazy evaluation. Some evaluators [27, 28] perform analysis of a tree at runtime to determine an evaluation order and then execute it. In this thesis we are not interested in such evaluators, so when we say "dynamically scheduled" we are referring to the more common demand-driven evaluation schemes[2].

There are numerous prominent examples of dynamically scheduled attribute grammar platforms: Saraiva's LRC [29] (1999), Hedin's JastAdd [30–32] (2003), Baars' UU AG [33] (2003), Van Wyk's Silver [7] (2008), and Sloane's Kiama [6] (2010). These are all dynamically scheduled attribute grammar platforms, but each has its own particular approach to solving the problem. To quote Paakki, who was not speaking of these particular systems but of attribute grammar systems in general: "Each system has its own specification language, in a sense a special dialect of attribute grammars" [34].

On top of notational differences, each platform adopts a subset of a large set of common attribute grammar extensions. This list includes (but is not limited to) reference attributes, parameterised attributes, attribute caching, higher order attributes, attribute forwarding, circular attributes, and collection attributes.

In this chapter we will focus on three prominent systems: JastAdd, Silver, and Kiama. In Section 2.2.1 we will discuss notational differences between these systems, implementing a common example. In Section 2.3 we will discuss which extensions are supported by each of these systems, and how they are implemented in each system.

2.2.1 Attribute Grammar Notations

Earlier in this chapter we presented an example attribute grammar written in a simple notation not belonging to any particular attribute grammar platform. In a similar notation, we present another toy program: globmin. From this point on we use sans serif text whenever we refer to a concrete attribute.

Production	**Semantics**
P ::= N	N.globmin = N.locmin
N_0 ::= N_1 N_2	N_0.locmin = min(N_1.locmin, N_2.locmin)
	N_1.globmin = N_0.globmin
	N_2.globmin = N_0.globmin
N ::= Int	N.locmin = Int.value

The purpose of globmin is to give every node access to the value of the smallest leaf value in the tree. The locmin attribute is synthesised, giving each subtree access to its smallest leaf value. The globmin attribute is inherited, copying this value back down the tree to every node.

[1] Again, there are some approaches in statically scheduled systems that allow for evaluating conditionals that would not appear non-circular in the the original sense [26]. As with reference attributes, this is a feature that is provided 'for free' by dynamic schedulers, rather than having a special algorithm provided to deal with them.

[2] Technically, demand-driven evaluation is not "scheduled" at all, as it happens on-the-fly. We use this term, however, as it is common in literature and helps to distinguish these approaches from traditional statically-scheduled approaches.

We will show how this simple attribute grammar program can be implemented in Silver, JastAdd, and Kiama.

Silver

Silver (named for "Ag", the chemical symbol for silver) is a fully generated attribute grammar platform developed in Silver via bootstrapping. Silver is *fully generated* as there is no "back-door" to any host language for writing attribute equation definitions; the entire attribute grammar language is custom made. This property is helpful as allowing arbitrary general-purpose code in expressions can lead to behaviour not supported by the attribute grammar platform.

Silver delegates its parsing to Copper, a parser generator that usually comes bundled with the framework. Silver supports syntax definition that builds an abstract syntax tree, and attributes can be defined on either the concrete tree, the abstract tree, or both. In the example we will explore, we define the attribute locmin and globmin on the abstract syntax tree.

Figure 2.1 shows globmin and locmin specified in Silver, for a binary tree made from `Expr` nodes, `Integer` leaves, and a `Root` root node. Line 1 declares that locmin is a synthesised attribute, and line 2 declares that globmin is an inherited attribute. Both are of type integer.

Lines 4 and 5 state which attributes are applicable to which nonterminals. There are three productions in this abstract grammar, as in the globmin example given previously. There are five attribute equations given, which match up with the five equations given in the previous example.

The synthesised attribute locmin is defined on lines 16 and 24. These rules have a symbol referencing the "head" of the production rule on the left hand side of the assignment, accessing symbols representing the node's children on the right hand side. This means that information flows upwards in the tree.

The inherited attribute globmin is defined on lines 10, 17, and 18. These rules have a symbol referencing a child node on the left hand side of the assignment, accessing the symbol representing the "head" of the production rule on the right hand side. This means that information flows downwards in the tree.

There are two nonterminals in use in this example, but there are three sets of equations. In Silver, an attribute's equation at any point in the tree is selected based not on the type of the node, but on the production rule that derived it. Sometimes that equation is given by the derivation of the node itself (synthesised attributes), and sometimes it is given by its parent's derivation (inherited attributes). In our terminology, this means Silver uses *production-based* equation selection.

JastAdd

JastAdd is a partially-generated attribute grammar platform developed in Java, which accepts imperative Java snippets as part of attribute equation definitions, and compiles to Java. JastAdd is *partially generated* as systems are written in a JastAdd-specific language which is parsed and processed outside of Java, but it allows a back door into Java through untranslated code snippets. This property is helpful as it allows a combination of declarative specification provided by attribute grammars and imperative programming provided by Java.

JastAdd works with any Java-based parser generator that supports semantic actions to allow the parsing specification to build JastAdd ASTs [35]. Attribute definitions are separated into inherited and synthesised definitions, as in Silver. We implement the globmin

and locmin attributes in a working example that decorates a binary tree, as we did in Silver.

Figure 2.2 shows globmin and locmin specified in JastAdd, for a binary tree made from `EPlus` fork nodes, `EInt` leaf nodes, and a `Program` root node. Line 2 declares that locmin is a synthesised attribute, and line 3 declares that globmin is an inherited attribute. Both are of type integer, and occur on `Expr` nodes. `Expr` is an abstract node type, consisting of both `EPlus` and `EInt`.

Lines 5 and 6 provide equations for the locmin attribute on the two types under `Expr`. On the right hand side of the assignment-style syntax, getters such as `getL` and `getR` provide access to component (child) nodes, and attributes are accessed using method notation, as in `getL().locmin()`. Generic Java code is allowed in these equations, so the Java method `Math.min` is used as a semantic function. In Line 6, `getV` accesses the integer component of an `EInt` node.

The inherited attribute globmin is defined on lines 8, 9, and 10. Line 8 specifies globmin's attribute equation for any node returned from the `getExpr` method of a `Program` node. The right-hand side of this assignment-style rule specifies a value in the context of the parent `Program` node.

This is an interesting notation. JastAdd attribute equations are not written alongside production rules; they are separated into "aspects" and use object-oriented style declarations, defining a "method" for each appropriate "subclass". When using this OO-style approach, an important question to ask is: given that a node does not know about its parent, how does one define inherited attributes?

Hedin *et al.* settled on the notation shown in lines 8-10, where inherited attributes are defined in the context of the parent node, which very closely mirrors the production-based inherited attributes of Silver and more traditional attribute grammar systems. However, JastAdd does *not* use production-based attribute selection; they have just implemented notation that allows a parent to define its children's attribute equations. Equations are defined according to the type of the node, which means JastAdd uses *symbol-based* equation selection.

JastAdd also provides "autocopy" rules for inherited attributes. If lines 9 and 10 are removed from the specification in Figure 2.2, the behaviour will not change. Inherited attributes are defined to automatically query the same attribute on their parent, unless otherwise specified. Leaving out lines 9 and 10 would mean the children of an `EPlus` node would request their parent's globmin, which would request its parent's attribute, and so on until a node with a differently-defined equation is met.

Silver also supports autocopy inherited attributes with the `autocopy` keyword.

Kiama

Kiama is a fully embedded attribute grammar platform embedded in Scala [36]. Unlike Silver and JastAdd, Kiama exists entirely within a general purpose programming language, so is not a code generator at all. Scala has support for domain-specific notations that allow libraries like Kiama to specify their own syntax, to an extent, which will be processed as valid Scala code. This approach grants a great deal of flexibility, as the full functionality of Scala is available alongside the attribution mechanics implemented in Kiama. Further, attribution mechanics need not stand alone; they can be added to any appropriate tree in an existing project in Scala.

Like JastAdd, Kiama works with any kind of parser or parser generator in Scala. The "input" for Kiama attribute grammars is not a source file but any tree based on `Product`. `Product` is a special high-level trait in Scala that represents simple classes that contain

```
1  synthesized attribute locmin :: Integer;
2  inherited attribute globmin :: Integer;
3
4  nonterminal Root;
5  nonterminal Expr with globmin, locmin;
6
7  abstract production root
8  r::Root ::= e::Expr
9  {
10   e.globmin = e.locmin;
11 }
12
13 abstract production add
14 sum::Expr ::= l::Expr r::Expr
15 {
16  sum.locmin = min(l.locmin, r.locmin);
17  l.globmin = sum.globmin;
18  r.globmin = sum.globmin;
19 }
20
21 abstract production integerConstant
22 e::Expr ::= i::Integer
23 {
24  e.locmin = i;
25 }
```

Figure 2.1: The globmin program specified in Silver.

```
1  aspect GlobminLocmin {
2    syn Integer Expr.locmin();
3    inh Integer Expr.globmin();
4
5    eq EPlus.locmin() = Math.min(getL().locmin(), getR().locmin());
6    eq EInt.locmin() = getV();
7
8    eq Program.getExpr().globmin() = getExpr().locmin();
9    eq EPlus.getL().globmin() = globmin();
10   eq EPlus.getR().globmin() = globmin();
11 }
```

Figure 2.2: The globmin program specified in JastAdd.

instances of other classes, with some basic scaffolding to access children arbitrarily. Kiama does, however, provide a parser combinator library which we used to build our working globmin program.

```
1  lazy val locmin : Expr => Integer =
2      attr {
3          case EPlus(l, r) => min(locmin(l), locmin(r))
4          case EInt(i)     => i
5      }
6
7  lazy val globmin : Expr => Integer =
8      attr {
9          case tree.parent(p) => globmin(p)
10         case e              => locmin(e)
11     }
```

Figure 2.3: The globmin attribute program specified in Kiama.

Figure 2.3 shows globmin and locmin specified in Kiama, for a binary tree made from `EPlus` fork nodes and `EInt` leaf nodes. Lines 1-5 define the locmin attribute, and lines 7-11 define the globmin attribute. Neither attribute is declared to be either synthesised or inherited, as Kiama does not make a distinction between these two types of attribute. Both attributes are of type `Integer`.

Line 1 indicates that locmin occurs on any `Expr` node. In our example, all nodes are `Expr` nodes, so this means that locmin occurs on every node in the tree. Further, the equation(s) provided in lines 2-5 apply to every node. It is possible to define attributes that only occur on specific types of node in Kiama. An attribute is essentially a function from node to value, and the domain of that function specifies which nodes have an attribute "occurrence" on them.

The function that defines locmin is defined on lines 2-5. It is normal in Kiama to phrase this function as a pattern match on the input node, as is shown on lines 3 and 4. This pattern provides the semantic equivalent of writing different equations for each node type, while still allowing the attribute grammar writer to write more general expressions. Pattern matching allows convenient access to child nodes, as seen on line 3. Attributes are accessed using a function notation, as in `locmin(l)`.

The globmin attribute draws information from nodes higher in the tree. In the traditional sense, globmin is an inherited attribute. Kiama provides pattern matching extractors (also known as view patterns) to grant access to related nodes in the tree. Line 9 uses the `tree.parent` extractor to attempt to retrieve `p`, the node's parent. If this parent exists, its globmin is recursively requested. If it does not exist, then the `tree.parent` extractor fails, and the generic case in line 10 is matched. Line 10 exists because it is possible for there to be a node with no parent: the root node. Instead of providing a special root node type, like `Root` in the Silver example and `Program` in the JastAdd example, Kiama simply recognises the absence of the `parent` to achieve equivalent behaviour. Kiama also supports "autocopy" inherited attributes through the use of decorators, which describe families of attribute traversals.

Kiama defines a single equation for each attribute. It is normal behaviour to define this

equation as a set of pattern matches on node types to implement the semantics of symbol-based attribute selection, but this is optional, and it is possible to write a single simple equation for any node type. Therefore Kiama uses *attribute-based equation selection*, while providing tools to separate equations in a number of useful ways.

2.3 Attribute Grammar Extensions

The attribute grammar literature has been producing *extensions* to base attribute grammars from the very beginning. In this chapter we discuss a number of common attribute grammar extensions, and how their implementations differ in modern dynamically scheduled attribute grammar platforms.

2.3.1 Reference Attribute Grammars

It makes sense to say that a tree is a collection of related nodes, where there are exactly two kinds of relationship: the *parent* relationship and the *child* relationship. Attribute grammars were designed from the beginning to facilitate computations using these two relationships. However, after a few decades of use, it became clear that non-local dependencies were a common concern when writing attribute grammar specifications. In other words, sometimes there are important relationships in a tree that are not as simple as 'parent' or 'child', which would be useful to leverage when defining attribute equations. For example, to retrieve the type of a variable use we might want to evaluate `var.decl.type`, where `decl` is the relationship between a variable use and its declaration.

A common solution to this problem is to allow attributes to evaluate to a reference to a node, and to allow access to the attributes of this returned node. Poetzsh-Heffter presented the MAX system in 1993 [37] which allowed *distant attribute occurrences*, enabling this behaviour. Boyland presented *remote attribute grammars* in 1996 [38], which similarly allow access to remote nodes. Hedin presented *reference attribute grammars* in 2000 [4], which allow such behaviour using an object-oriented approach. The term "reference attribute grammars" is used more commonly than "remote" or "distant", so it is the term we use in this thesis.

Reference attribute grammars are usually implemented by allowing attribution expressions to form a chain, such as `node.refAttr.locAttr`. To demonstrate reference attributes, we define the "refmin" problem. This is similar to the "globmin" problem of Section 2.2.1, but every node now is granted a reference to the smallest leaf in the tree. In the same simple notation we have used before, we can express this problem as follows.

```
Production    Semantics
P ::= N       N.globrefmin = N.locrefmin
N0 ::= N1 N2  N0.locrefmin = if (N1.locrefmin.val < N2.locrefmin.val)
                then N1.locrefmin
                else N2.locrefmin
              N1.globrefmin = N0.globrefmin
              N2.globrefmin = N0.globrefmin
N ::= Int     N.locrefmin = N
```

We implement refmin using two attributes; `locrefmin` and `globrefmin`. The strategy is very similar to that used in the globmin example, but with references to nodes passed around,

```
synthesized attribute locrefmin :: Expr;
autocopy attribute globrefmin :: Expr;
synthesized attribute val :: Integer;

nonterminal Root with locrefmin;
nonterminal Expr with globrefmin, locrefmin, val;

abstract production root
r::Root ::= e::Expr
{
  r.locrefmin  = e.locrefmin;
  e.globrefmin = e.locrefmin;
}

abstract production add
sum::Expr ::= l::Expr r::Expr
{
  sum.locrefmin = if (l.locrefmin.val < r.locrefmin.val)
                then l.locrefmin
                else r.locrefmin;
}

abstract production integerConstant
e::Expr ::= i::Integer
{
  e.locrefmin = e;
  e.val = i;
}
```

Figure 2.4: The refmin attribute program specified in Silver

```
lazy val locrefmin : Expr => EInt =
  attr {
    case EPlus(l, r) if locrefmin(l).i
                      < locrefmin(r).i => locrefmin(l)
    case EPlus(_, r)                   => locrefmin(r)
    case e : EInt                      => e
}

lazy val globrefmin : Expr => EInt =
  attr {
    case tree.parent(p) => globrefmin(p)
    case e              => locrefmin(e)
}
```

Figure 2.5: The refmin attribute program specified in Kiama.

```
1  aspect Refmin {
2    syn EInt Expr.locrefmin();
3    inh EInt Expr.globrefmin();
4
5    eq Program.getExpr().globrefmin() = getExpr().locrefmin();
6
7    eq EPlus.locrefmin() {
8      EInt lmin = getL().locrefmin();
9      EInt rmin = getR().locrefmin();
10     if (lmin.getV() < rmin.getV())
11       return lmin;
12     return rmin;
13   }
14
15   eq EInt.locrefmin() = this;
16 }
```

Figure 2.6: The refmin attribute program specified in JastAdd.

instead of integer values. Figure 2.4 shows these attributes implemented in Silver. The attributes locrefmin and globrefmin are defined on lines 1 and 2, and are of type `Expr`. Line 18 shows that the node represented by locrefmin can have its attributes accessed, as in `l.locrefmin.val`. The most interesting part of this specification is line 26, where one of the production symbols `e` is the entirety of the return equation. This means the attribute value in this instance is a reference to the node itself.

Figure 2.6 shows the same attributes implemented in JastAdd. Since the tree in JastAdd distinguishes in type between fork and leaf nodes (unlike the Silver implementation), the attributes locrefmin and globrefmin return an `EInt` node, which is specifically a leaf node. The advantage of this is that the type system guarantees that these attributes will never return a fork node. This also means that it is possible to directly access the non-attribute fields of the return node, as in `lmin.getV()` on line 10. Line 15 shows the attribute locrefmin returning a reference to the node being evaluated upon, using the `this` keyword.

Figure 2.5 shows these attributes implemented in Kiama. Again, the attributes return specifically a leaf node, in this case of type `EInt`. Kiama here blurs the line between equation selection and attribute definition by allowing attributes to be evaluated as a guard on a pattern match. This is again one of the advantages of a deep embedding in a general purpose programming language. Lines 3 to 5 define locrefmin for fork nodes, and Line 6 returns the pattern matched metavariable `e` to provide a reference to the node being evaluated upon.

The calculus we present in this thesis implements reference attribute grammars in a similar way to the examples depicted here; attributes can simply return a node reference, and this reference can have its attributes evaluated. In fact, our calculus uses reference attributes to implement all kinds of relative access, such as the `parent` or `child` relatives, not just remote node access. See Chapter 3 for more detail.

```
aspect CountLess {
  syn Integer Expr.countless(Integer i);
  eq EPlus.countless(Integer i) = getL().countless(i)
                                + getR().countless(i);
  eq EInt.countless(Integer i) = (getV() < i) ? 1 : 0;
}
```

Figure 2.7: The countless attribute specified in JastAdd.

```
lazy val countless : Integer => Expr => Integer =
  paramAttr {
    cap => {
      case EPlus(l, r) => countless(cap)(l) + countless(cap)(r)
      case EInt(i)     => if (i < cap) 1 else 0
    }
  }
```

Figure 2.8: The countless attribute implemented in Kiama.

```
aspect CountLess {
  syn Integer Expr.countless(Integer i);
  eq EPlus.countless(Integer i) = getL().countless(i)
                                + getR().countless(i);
  eq EInt.countless(Integer i) {
    System.out.println("countless("+i+") eval'd on leaf " + getV());
    return (getV() < i) ? 1 : 0;
  }
}
```

Figure 2.9: The countless attribute specified in JastAdd, with debug printing.

```
tree: 9 + 7 + 4 + 3 + 99
countless(15) eval'd on leaf 9
countless(15) eval'd on leaf 7
countless(15) eval'd on leaf 4
countless(15) eval'd on leaf 3
countless(15) eval'd on leaf 99
countless(15)'s value on root node: 4
countless(15)'s value on root node: 4
countless(5) eval'd on leaf 9
countless(5) eval'd on leaf 7
countless(5) eval'd on leaf 4
countless(5) eval'd on leaf 3
countless(5) eval'd on leaf 99
countless(5)'s value on root node: 2
```

Figure 2.10: The output of running the JastAdd implementation of countless with debug printing, on the tree parsed from `9 + 7 + 4 + 3 + 99`.

2.3.2 Parameterised Attribute Grammars

If you approach attribute grammars from an object-oriented point of view, then attributes can be considered parameterless methods on a node. *Parameterised attributes* [4] emerge when these "methods" become parameterised. Parameterised attributes are not implemented in statically scheduled attribute grammar systems, as adding parameterisation changes the number of attribute dependencies each attribute can have from definitely finite to potentially infinite.

While it may be an intractable task to implement parameterisation in a statically scheduled system, object-oriented attribute grammar platforms like JastAdd and Kiama implement them quite simply. Silver does not support parameterised attributes in the same sense that Kiama and JastAdd do; attributes can have a type parameter, and attributes can return a function, but attributes themselves can not be parameterised.

Generally speaking, the syntax for calling a parameterised attribute is similar to the familiar syntax of method calling, as in `node.parAttr(param)`. To demonstrate parameterised attributes, we define the "countless" problem. This is set on a binary tree with integer leaves, as in previous examples, but involves only the parameterised synthesised attribute `countless`, which counts the number of leaves under a certain value in a subtree.

Production	**Semantics**
`P ::= N`	`P.countless(c) = N.countless(c)`
N_0 `::=` N_1 N_2	N_0`.countless(c) =` N_1`.countless(c) +` N_2`.countless(c)`
`N ::= Int`	`N.countless(c) = Int.value < c ? 1 : 0`

Figure 2.7 shows `countless` implemented in JastAdd. The implementation is straightforward, with attribute parameter declarations written between previously empty parentheses on lines 2, 3, and 5. Java's ternary conditional operator is used at leaf nodes.

Figure 2.8 shows `countless` implemented in Kiama. The type signature of the attribute on line 1 shows that an integer is expected as well as an `Expr` node as an input to the computation. It is required in Kiama that parameters be placed before nodes when specifying parameterised attributes. Line 2 shows the use of the `paramAttr` wrapper instead of the usual `attr`. The integer is matched by the metavariable `cap` and used in expressing the attribute's equation. As Kiama calls attributes using a function notation, parameterised attributes are called in a curried way, as seen in `countless(cap)(l)` on line 4.

The calculus we present in this thesis implements parameterised attributes by requiring all attributes to take a single parameter, and using the unit type and tuples to accommodate attributes with zero or multiple parameters. See Chapter 4 for more detail.

2.3.3 Attribute Caching

Statically scheduled attribute grammar evaluators evaluate every attribute on every node in a tree, usually in multiple passes. Multiple passes are only useful if each pass has the side effect of decorating the tree some with computed attribute values. When using a recursive demand-driven approach to evaluation, as most dynamically scheduled platforms use, it is no longer strictly necessary to store computed attribute values; as long as there are no side effects, recomputing attributes will not change any outputs.

However, recomputing values is usually slower than recalling stored attribute values. The seminal paper on demand-driven evaluation [3] did not use the words "caching" or

"memoisation", but makes its time-optimal claims based on the same strategy. Silver, JastAdd and Kiama all support attribute caching transparently. Both JastAdd and Kiama support the caching of parameterised attributes, where the value for each attribute, and for each set of parameter values for that attribute, are cached upon evaluation. For example `node.parAttr(1)` might be cached, but `node.parAttr(2)` might still be uncached, and would be computed on demand.

As caching is not a notation-level extension, but a behaviour that happens silently during evaluation, we will not implement any new examples to demonstrate this feature. However, we will show the output when evaluating the JastAdd implementation of countless, with the extra line 6 as shown in Figure 2.9.

Figure 2.10 shows the output when running our compiled version of countless implemented in JastAdd, when the input tree is a binary tree representation of `9 + 7 + 4 + 3 + 99`. First, `countless(15)` is evaluated on the root node. Lines 2-6 show that each of the leaves are accessed in left-to-right order, as their values for `countless(15)` are evaluated. The output value of 4 is shown in line 7.

Then `countless(15)` is evaluated on the root again. Notice on line 8 that the output value of 4 is given without any intermediary leaf evaluations. This is because the value for `countless(15)` has been cached for every node in the tree, meaning evaluation is no longer necessary, so the output side-effect from line 6 of Figure 2.9 is never triggered.

Finally, `countless(5)` is evaluated on the root node. Since `countless(5)` has not been evaluated on any nodes in the tree yet, we again see that evaluation reaches each leaf node in left-to-right order, before finally a value of 2 is found for the root node, as shown on line 14. This demonstrates that JastAdd is silently caching parameterised attributes during evaluation.

The calculus we present in this thesis implements attribute caching, including the caching of parameterised attributes. See Chapter 4 for more detail.

2.3.4 Higher Order Attribute Grammars

While reference attribute grammars allow an attribute to return a reference to a node in the existing tree, *higher order attribute grammars* allow attribute evaluation to construct new nodes (and therefore trees), which can in turn be decorated. Higher order attribute grammars were introduced by Vogt and Swierstra in the late 1980s [39, 40], and are sometimes referred to as *non-terminal attributes*.

There are statically scheduled attribute grammar platforms that support higher-order attributes, such as in the work of Vogt *et al.* [39], which uses ordered attribute grammars. However, statically defining a traversal for a tree that grows during evaluation can be a complex task. Dynamic scheduling, however, can take higher-order evaluation in its stride.

Higher order attribute grammars are implemented in Silver, JastAdd, and Kiama. Silver uses higher order attributes as part of a normal work flow, to generate an abstract syntax tree as a higher order attribute of a concrete syntax tree. We used this feature to build the abstract syntax we wrote attributes for in all of the Silver examples in this chapter so far.

To demonstrate this extension, we present the repmin problem, which is similar to globmin, but every subtree is given a clone of itself, in which all leaves now hold the value of the smallest leaf node in the original tree. We adjust the notation we have been using for platform-independent grammar specifications to include a name with each production, which can be used as a constructor for new nodes. These constructors are used in the definition of the repmin attribute below. We assume the attribute globmin is available.

```
1  val repmin : Expr => Expr =
2    attr {
3      case EPlus(l, r) => EPlus(repmin(l), repmin(r))
4      case n : EInt    => EInt(globmin(n))
5    }
```

Figure 2.11: The repmin attribute implemented in Kiama, assuming globmin is already implemented.

```
1  syn nta Expr Expr.repmin();
2
3  eq EPlus.repmin() = new EPlus(getL().repmin(), getR().repmin());
4  eq EInt.repmin()  = new EInt(globmin());
```

Figure 2.12: The repmin attribute implemented in JastAdd, assuming globmin is already implemented.

```
1  synthesized attribute repmin<a> :: a;
2
3  nonterminal Root with locmin, repmin<Root>;
4  nonterminal Expr with globmin, locmin, repmin<Expr>;
5
6  abstract production root
7  r::Root ::= e::Expr
8  { r.repmin = root(e.repmin); }
9
10 abstract production add
11 sum::Expr ::= l::Expr r::Expr
12 { sum.repmin = add(l.repmin, r.repmin); }
13
14 abstract production integerConstant
15 e::Expr ::= i::Integer
16 { e.repmin = integerConstant(e.globmin); }
```

Figure 2.13: The repmin attribute implemented in Silver, assuming globmin is already implemented.

Name	Production	Semantics
root	P ::= N	P.repmin = root(N.repmin)
fork	N_0 ::= N_1 N_2	N_0.repmin = fork(N_1.repmin, N_2.repmin)
leaf	N ::= Int	N.repmin = leaf(N.globmin)

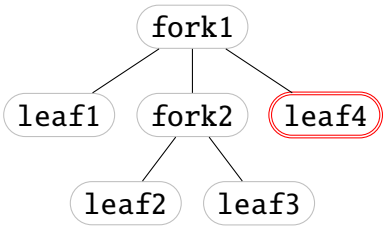
To explore how real-world attribute grammar platforms implement higher order attributes, we will implement repmin in Kiama, JastAdd, and Silver. First examine the Kiama implementation in Figure 2.11. Line 1 shows that the output type of the repmin attribute is `Expr`, which is the same as the input type. The attribute equations construct new trees in lines 3 and 4, recursively calling repmin to construct appropriate subtrees. There is no special notation to indicate that this attribute is higher order. However, there are limitations to Kiama's approach; new subtrees do not have access to their parent nodes, so inherited attributes of the root cannot be evaluated in the normal way on trees created during attribution.

The JastAdd implementation of repmin is shown in Figure 2.12. Line 1 uses the `nta` keyword to indicate that repmin is a nonterminal (or higher order) attribute. Otherwise, the implementation is very similar to the Kiama implementation, with lines 3 and 4 showing tree construction, including recursive calls to repmin.

Figure 2.13 shows the implementation of repmin in Silver, which is a little more interesting. While optional, Silver allows *type parameterised* attributes, and repmin is declared using this feature on line 1. Lines 3 and 4 indicate that repmin has type `Root` and `Expr` when called on a `Root` or `Expr` node, respectively. This is useful, as the type system now guarantees that the "clone" of each node will have the same type as its original. Such an implementation is not possible in Kiama or JastAdd. Apart from this change, implementation is similar, with new subtrees being constructed on lines 8, 12, and 16.

Higher Order Nodes in the Existing Tree

Consider the following binary tree, with fork and leaf nodes, and an additional higher order node `leaf4` (doubly outlined in red).



In the original tree, as with all trees that we aim to decorate, we assume that each node has access to its children and its parent. The relationship between `fork1` and `leaf1` is a child relationship, and the relationship between `leaf2` and `fork2` is a parent relationship.

How, then, do we classify the relationship between `fork1` and `leaf4`? How do we classify inverse relationship, between `leaf4` and `fork1`? It might seem like a good idea to consider `leaf4` to be one of `fork1`'s children, and it might seem like a good idea to consider `fork1` the parent of `leaf4`. However, not all attribute grammar platforms offer the same semantics in this regard.

In Kiama, `leaf4` is accessible to `fork1` via a higher order attribute, but it is not listed as one of its children; if a generic traversal decorator is used to visit every node in the tree, higher order nodes such as `leaf4` will be omitted. Further, the pattern matching extractor `tree.parent(p)` will not successfully recognise `fork1` as the parent of `leaf4`. If `leaf4`

had children, these would also not have access to `leaf4` as a parent node. This might be considered a shortcoming of Kiama's approach to higher order attributes, and changes to these semantics are intended for future versions of the platform.

In JastAdd, `leaf4` is considered a child of `fork1`, and `fork1` is considered the parent of `leaf4`. To be more precise, the method-style attribute reference that yields the higher order node `leaf4` can be used to define a synthesised attribute on `fork1` or an inherited attribute on `leaf4`. This behaviour is recursive, with any recursively created children of `leaf4` having full "up" and "down" access between them. This is further evidence that JastAdd implements symbol-based equation selection, as higher order attributes, even if there is no production rule associated with them, can be used to structure the equation selection process.

Silver adheres more strongly to a grammar than either Kiama or JastAdd, and this affects how higher order nodes can integrate into an existing tree. As attribute equation definitions are tied directly to the productions of a grammar, the only way a node can access its parent is if the parent was created with those children "in mind". In Silver, `leaf4` is accessible from `fork1`, but is not considered a child. Conversely, as in the Kiama case, `leaf4` can not access `fork1` as its parent node. However, if any nodes are recursively created or referenced under `leaf4`, these will know `leaf4` as their parent.

The calculus we present in this thesis implements higher order attributes such that the new node (`leaf4` in this case) is accessible as one of the "children" of `fork1` – a "child" relationship is no different to any other reference attribute, and the newly created node will always be accessible via the attribute that created it. Optionally, `fork1` can be defined as the parent of `leaf4`. To allow our semantics to match the semantics of any of the systems we have discussed, relationships between nodes can be influenced arbitrarily during higher order construction. There could be no relationships at all between new nodes and existing nodes, except of course via the attribute that created the new node. See Chapter 5 for more detail.

2.3.5 Attribute Forwarding

Attribute forwarding [41] is a method of defining the semantics of one type of node in terms of the semantics of another, via higher order construction. Of the platforms discussed in this thesis, forwarding is only implemented in Silver. Forwarding was designed for modularity, and involves building some subtree structure that is tied into the existing tree, and forwarding attribute requests to the new subtree, while explicitly defining some attributes on the original tree.

For example, let us assume we have a statement language that supports *while loops*, and the semantics of while loops are already implemented. If a new production allowing *for loops* was included, it would be useful to define the semantics of for loops in terms of the semantics of while loops. Below we show a snippet of an attribute grammar which implements the semantics of for loops by forwarding to while loops.

```
  Name  Production          Semantics
 while  S1 ::= E S2         S1.pp = 'while' ++ ...
                            S1.eval = ...
   for  S1 ::= S2 E S3 S4   S1.pp = 'for' ++ ...
                            forwards to block(S2, while(E, block(S4, S3))
```

Above we have defined the pp (pretty-print) attribute on both *while* and *for* productions; the new *for* production will have its own pretty-printing semantics. However, the **`forwards to`** syntax indicates that any other attribute, if requested on a *for* node, should be "forwarded" to the higher-order structure constructed by `block(`S_2`, while(E, block(`S_4`, `S_3`))`. For example if the eval attribute is requested of a *for* node, a higher order node will be constructed, eval will be evaluated on that node, and the result will be returned.

The tree below represents the original *for* subtree, and the corresponding *while* tree that will be constructed and forwarded to during evaluation. The children of the original tree are reused as the children of the newly constructed tree.

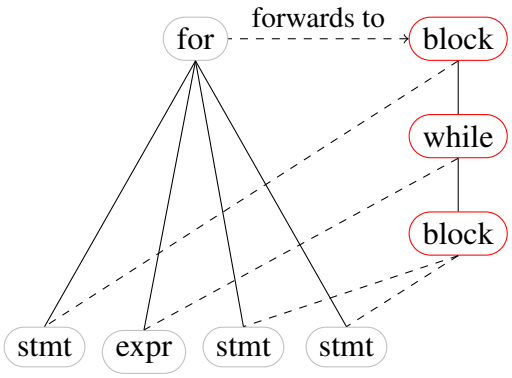


Figure 2.14 shows an implementation of these semantics in Silver. The two attributes we are interested in (pp and eval) are declared on lines 1 and 2. Lines 4-8 define the *while* production, which has normal definitions for the pp and eval attributes. Lines 10-14 define the *for* production, which defines the pp attribute in the normal way, but does not explicitly define the eval attribute. Instead, the *for* attribute is forwarded to a higher order construction, on line 13. As described previously, if any attribute except pp (for example eval) is evaluated on a *for* node, this evaluation will be forwarded to its associated *while* construction.

We consider forwarding to be primarily a notational extension, and as such we do not implement forwarding explicitly in our work. We discuss this in more detail in Section 5.3.4, and demonstrate how the semantics of forwarding can be expressed in our calculus.

2.3.6 Circular Attribute Grammars

When attribute grammars were first introduced, much emphasis was put on the restriction that attribute dependencies could not be circular (meaning there could be no cycles in attribute dependencies). However, circular attributes [42, 43] can be well defined, if a finite fixed-point can be reached in a finite number of iterations. Dynamically scheduled attribute grammar systems often contain run-time checks for attribute circularity, with some platforms (including JastAdd [44] and Kiama) supporting fixed-point iteration for circular attributes.

JastAdd allows circular attributes using fixed-point iteration, with the `circular` keyword and a starting value. Figure 2.15 shows the definition of the two attributes v and u, which are mutually recursive. Both attributes are specified as circular, with a starting value of `false`. Evaluation of either one of these attributes will begin the fixed-point evaluation process, which will quickly terminate with the value `false`.

Kiama also implements fixed-point iteration of circular attributes. See Figure 2.16 for an implementation of the same toy program in Kiama. Lines 2 and 7 define the attributes u and

```
1  synthesized attribute eval :: Integer;
2  synthesized attribute pp   :: String;
3
4  abstract production while
5  w::Stmt ::= c::Expr b::Stmt
6  { w.pp = "while" ++ ... ;
7    w.eval = ... ;
8  }
9
10 abstract production for
11 f::Stmt ::= i::Stmt c::Expr b::Expr p::Expr
12 { f.pp = "for" ++ ... ;
13   forwards to block(i, while(c, block(b, p)));
14 }
15
16 abstract production block
17 b::Stmt ::= s1::Stmt s2::Stmt
18 { ... }
```

Figure 2.14: Forwarding a *for* production, implemented in Silver.

v as circular, with base value `false` for both attributes. The rest of the attribute is defined as usual, with fixed-point iteration occurring transparently through the use of the `circular` keyword.

We consider circular attributes outside the scope of this thesis, and the calculus we present does not support this extension. This decision is discussed in Section 9.2, along with an outline for how fixed-point iteration for circular attributes might be implemented.

2.3.7 Collection attributes

Boyland introduced collection attributes in 1996 [38]. Collection attributes are a way of aggregating attribute values from a number of nodes, either from an entire program tree or from some sub-traversal of that tree. Collection attributes are a convenience extension, and save attribute authors from writing traversal and collection boilerplate attributes. It is common in compiler construction to use collection attributes for tasks such as collecting error messages for an entire program tree.

Kiama, JastAdd, and Silver all include notations for collection attributes. Consider the binary tree we have used for many of the examples in this chapter. We will define a collection attribute that collects all leaf values for such a tree.

Figure 2.17 shows the collection attribute `allInts` implemented in JastAdd. Line 1 uses the `coll` keyword to indicate that `allInts` is a collection attribute, and specifies it occur on a `Program` node and return a collection of integers. The syntax of a JastAdd collection attribute definition follows the pattern *type* `contributes` *expr1* `when` *expr2* `to` *def*. The type of node to contribute is listed in *type* (in this case `EInt`). An expression to calculate a value to contribute is given in *expr1* (in this case `getV()`). A boolean expression that decides whether or not to contribute is given in *expr2* (in this case the literal `true`), and *def* defines what attribute is being contributed to (in this case `Program.allInts()`). JastAdd's

```
1  aspect Circular {
2    syn boolean A.u() circular [false] = v();
3    syn boolean A.v() circular [false] = u();
4  }
```

Figure 2.15: A simple set of circular attributes defined in JastAdd.

```
1  lazy val u : A => Boolean =
2    circular (false) {
3      case n => v(n)
4    }
5
6  lazy val v : A => Boolean =
7    circular (false) {
8      case n => u(n)
9    }
```

Figure 2.16: A simple set of circular attributes defined in Kiama.

evaluator will traverse every node in the tree that can possibly contribute to `allInts`, evaluate its `when` expression, and optionally evaluate its `contributes` expression and contribute its value to the `allInts` list. JastAdd's collection syntax also provides the option to contribute a value not to the "root" of the collection, but to some other node. This allows collection behaviour to be customised beyond the usual "collect from all children" strategy.

Figure 2.18 shows a similar collection attribute implemented in Kiama. The `collectAll` decorator is used, which takes a partial function returning integers. Line 3 indicates that any `EInt` node should contribute its integer to the collection. As in the JastAdd implementation, a guard could be added to line 3 to not always contribute a value. As `collectAll` is just an instance of a decorator, an attribute author can define whatever kind of nonstandard collection behaviour they desire by defining a separate decorator for their task.

Figure 2.19 shows the `allInts` attribute defined in Silver. Silver's implementation of collection attributes is different than Kiama's and JastAdd's, as the traversal for collecting attributes must be given explicitly. Line 1 indicates that the `allInts` attribute will calculate a list of integers, which will be combined with the ++ (list concatenation) operator. Collection attributes in Silver do not use the standard = operator, but use both the `:=` and `<-` operators, which define a base value and a contribution respectively.

In the implementations shown for JastAdd and Kiama, both collection attributes assume that collection will begin with the empty list. Both JastAdd and Kiama allow for this base value to be customised, but Silver goes one step further and allows this base value to be defined *per production*. Lines 9, 15, and 21 define the base values for the `allInts` collection attribute, for each production. Line 22 is the only place where this list is explicitly contributed to, yielding the value of an `integerConstant` node.

In this case we only contribute in one place, but Silver is designed for modular language design. The "traversal" dictated on lines 9, 15, and 21 may be defined in one file, which lays out the basic structure of a collection attribute, while other definition files may contribute values here or there. A common use-case for this is compiling error messages, which may

```
coll ArrayList<Integer> Program.allInts();

EInt contributes
  getV()
  when true
  to Program.allInts();
```

Figure 2.17: A simple collection attribute defined in JastAdd.

```
lazy val allInts =
  attr(collectAll {
    case EInt(i) => i
  })
```

Figure 2.18: A simple collection attribute defined in Kiama.

```
synthesized attribute allInts :: [Integer] with ++;

nonterminal Root with allInts;
nonterminal Expr with allInts;

abstract production root
r::Root ::= e::Expr
{
  r.allInts := e.allInts;
}

abstract production add
sum::Expr ::= l::Expr r::Expr
{
 sum.allInts := l.allInts ++ r.allInts;
}

abstract production integerConstant
e::Expr ::= i::Integer
{
 e.allInts := [ ];
 e.allInts <- [i];
}
```

Figure 2.19: A simple collection attribute defined in Silver.

be produced by different aspects of a compiler: name checking, type checking, *etc.*. While Silver's approach is more verbose, it is also more generalised, allowing collection traversal to ignore entire subtrees if required (without guarding each node in that subtree from collection).

Collection attributes are not in the scope of this thesis, partly due to the fact that they can be implemented using the other extensions described in this chapter. Collection attributes can be difficult to evaluate in a dynamically scheduled system if you combine them with reference or circular attributes, but their semantics are about evaluating a very specific kind of value. The focus of Saiga is to model attribute grammars with as simple a calculus as possible, and this is somewhat at odds with extensions like collection attributes, which can be seen as almost a notational extension, with complex and specifically targeted semantics behind them.

2.4 Previous Attempts

The primary goal of the research presented in this thesis is to provide a framework for formally specifying attribute grammars such that the semantics of their evaluation can be formally examined. Much work has been published that uses attribute grammars to formalise various kinds of semantics, but we are not interested in the semantics that attribute grammars can model, but the semantics of attribute grammars themselves; particularly, the semantics of dynamic attribute evaluation.

A number of projects have embedded attribute grammars into functional languages to model their evaluation semantics. De Moor *et al.* showed how to define compositional attribute grammars in Haskell using an aspect-oriented approach [45, 46], following earlier work from Johnsson and Jourdan on implementing attribute grammars as functional programs [3, 47]. Backhouse also defined a Haskell-based implementation of attribute grammars that was used to reason in "a calculational style" and to derive a new test for definedness [48].

These projects focused on embedding attribute grammar semantics into Haskell, both to allow Haskell authors access to new kinds of declarative specification and to model new approaches to attribute grammar construction and composition. The focus of our work, in contrast, is to model existing attribute grammar evaluation schemes in a formalism simple enough that their semantics are not obfuscated by their encoding. Embedding in Haskell leads to powerful expressibility while providing a window to analyse the properties of a particular attribute grammar program. Our work in formalising attribute grammar evaluation using inference rules allows us to analyse the semantics not only of particular attribute grammar programs, but of the semantics of attribute evaluation itself.

Schaefer *et al.* implements some work more closely related to ours, implementing circular reference attribute grammars as a shallow embedding in Coq [49]. They use zippers to keep track of locations in trees, an approach that has been explored further by others [50]. Embedding in Coq allows mechanised proofs to be written about non-trivial attribute grammar programs, but this implementation suffers from the same issue as the Haskell approaches; a tight relationship with the semantics of the host language. The evaluation semantics in Schaefer's work is considerably more complex than the semantics we present in this thesis. In our earlier work we deeply embed our semantics in Coq [51], and we discuss a deep embedding in Lean in Chapter 8. A deep embedding allows us to mechanise metatheoretic properties about our calculus.

While many shallow embeddings of attribute grammars exist, and even some frameworks for writing mechanised proofs about attribute grammars, we have found no other work that is directly interested in reasoning about the evaluation semantics of dynamically scheduled attribute grammars. We have certainly not found any other attempts to encode attribute evaluation on a level as low as the calculus we present in this thesis.

Fermentation may have been a greater discovery than fire.

David Rains Wallace

3 Saiga

In this chapter we develop a core calculus for dynamically scheduled attribute grammar evaluation, which we call "Saiga". We say a "core" calculus because in this chapter we will model only the most basic attribute grammar features we can; in Chapter 4 we extend this calculus to include caching and parameterised attributes, and in Chapter 5 we explore higher-order attributes in Saiga. The three key design principles we aim to follow are simplicity, generality, and domain specificity.

Simplicity dictates that our calculus will be as minimal as possible while encoding all of the required features. For example, we want to minimise both the number of semantic rules we implement and the complexity of each of these rules.

Generality dictates that our calculus should not be tied to any existing attribute grammar notation or evaluation approaches. There are a number of standards for notation, for equation selection, and for attribute evaluation (as discussed in Chapter 2). Our calculus should be able to comfortably encode attribute grammar programs from any of these families using the same calculus features.

Domain specificity dictates that our calculus should focus only on attribute grammar evaluation. More general computing concerns such as arithmetic, the construction of complex values like lists, value comparisons etc should be delegated to an external system.

3.1 Calculus

We build our calculus on top of an assumed underlying calculus, which is explored in Section 3.1.1. We define some types in Section 3.1.2, an expression language in Section 3.1.3, type rules for this language in Section 3.1.4, and a set of operational semantics in Section 3.1.5. We also demonstrate these semantics with a simple example in Section 3.1.7.

3.1.1 An Underlying System

To satisfy the domain specificity requirement, we build our calculus on top of an assumed underlying functional calculus. When we talk about the "underlying type system", we refer to the type system of this calculus. When we say "underlying function application", we refer to function application in the context of this calculus.

We use $\boldsymbol{T}$ to refer to the set of all types in the underlying system. In this thesis we assume that types in $\boldsymbol{T}$ are monomorphic, but our calculus could support non-monomorphic underlying type systems in the future. We do not specify all of the types contained in $\boldsymbol{T}$, but we know that we will require $\boldsymbol{T}$ to include booleans, functions, and eventually the node type (which we will define later in this chapter). While these three types alone are enough to encode our semantics, we will expand $\boldsymbol{T}$ in various examples in this and later chapters. For example, we will expand $\boldsymbol{T}$ with integers, strings, arbitrary enumerated types, as well as monomorphic lists and tuples.

$$\begin{array}{rcl} \boldsymbol{T} & ::= & \boldsymbol{T} \to \boldsymbol{T} \\ & | & \text{boolean} \\ & | & \ldots \end{array}$$

We use the metavariables t and v, or subscripted versions thereof, to respectively indicate a type in $\boldsymbol{T}$ or a value in some t.

$$\begin{array}{rl} \text{Type} & t \in \boldsymbol{T} \\ \text{Value} & v \in t \end{array}$$

We assume that function application exists in the underlying system. We use the notation $t_1 \to t_2$ to specify the type of a function that takes a parameter of type t_1 and returns a value of type t_1. If we have some $f \in t_1 \to t_2$ and some $x \in t_1$, we use the notation $f(x)$ to represent the value returned by the function f when applied to the parameter x. Standard beta reduction rules apply in the underlying system.

We allow for concrete anonymous function values to be expressed using lambda notation, for example $\lambda x.x + 1$. We also allow the definition of named functions in italics as below.

$$\mathit{plusOne}(x) = x + 1$$

We assume that currying is performed automatically, so if we define a multiparameter function *plus* as below, this is short-hand for the curried sequence of single-parameter functions.

$$\mathit{plus}(a, b) = a + b$$

The definition of *plus* above is identical to the definition of *plus* below.

$$\mathit{plus} = \lambda a.\lambda b.a + b$$

This allows for transparent partial function application, such that we can write $\mathit{plus}(2)$ to indicate the function that will add two to the given number. The underlying calculus is functional, so all functions are pure.

From the perspective of Saiga, *terms* and *values* in the underlying system are indistinguishable. Evaluation in the underlying system is not part of Saiga's semantics, and we assume that it happens transparently. We are happy to treat the term $\mathit{min}(5)(2)$ as equivalent to the term 2, as underlying evaluation is transparent to Saiga. We use the phrase "underlying

values" or "values in the underlying system" to refer to either terms or values of the underlying system, as each term in the underlying system is evaluable to some value, whether we know that value or not.

The underlying calculus is also typed. To be more verbose, we could also define *plus* as follows.

$$plus = \lambda(a : integer).\lambda(b : integer).a + b$$

We allow types to be omitted however, as long as they can be sensibly assumed or inferred.

3.1.2 All Types

Type	$t \in \boldsymbol{T}$
Value	$v \in t$
Expression	$e \in \boldsymbol{E}$
Attribute	$a \in \boldsymbol{A}$
Node	$n \in \boldsymbol{N}$
Attribute Type	$\tau \in \boldsymbol{A} \rightarrow \boldsymbol{T}$
Context	$\sigma \in \boldsymbol{N} \rightarrow \boldsymbol{A} \rightarrow \boldsymbol{E}$

$$\begin{array}{rcl} \boldsymbol{T} & ::= & \boldsymbol{T} \rightarrow \boldsymbol{T} \\ & | & \text{boolean} \\ & | & \boldsymbol{N} \\ & | & \ldots \end{array}$$

We start by defining the types in Saiga. We call types in $\boldsymbol{T}$ "underlying types", as most types in $\boldsymbol{T}$ are inherited from the underlying system (the exception to this rule is the node type $\boldsymbol{N}$). The intention of $\boldsymbol{T}$ is that any Saiga expression should be evaluated some value of a type in $\boldsymbol{T}$. We use the term "value" to refer to values or terms of the underlying system - Saiga does not differentiate between terms and values in the underlying system, but considers a term to be equivalent to the value it will evaluate to. $\boldsymbol{E}$ is the expression type, whose grammar is described in Section 3.1.3. $\boldsymbol{A}$ and $\boldsymbol{N}$ are simple enumerable types which provide labels for attributes and nodes respectively. The function τ provides the expected (underlying) type of any attribute in $\boldsymbol{A}$. The context function σ provides the attribute equation expression for any node and attribute. We also specify that functions, booleans, and nodes are types in $\boldsymbol{T}$. We allow that more monomorphic types can be added to $\boldsymbol{T}$ as required.

3.1.3 Expression Language

We define an expression language as follows.

e	::=	$[\![v]\!]$	value
	|	`IF` e_1 `THEN` e_2 `ELSE` e_3	conditional
	|	e_1`(`e_2`)`	function application
	|	e`.`a	attribution

The structure of this grammar is straightforward. We lift a value v from the underlying system into our expression language using the syntax $[\![v]\!]$. This notation is used to make clear the difference between a value v and an expression that holds that value, $[\![v]\!]$.

Conditional expressions and function application expressions are straightforward in their structure and semantics. Attribution expressions allow the encoding of expressions that will

request a node's attribute. The semantics of each expression form is shown in Section 3.1.5.

3.1.4 Type Rules

As all expressions are intended to evaluate to a value expression, and value expressions use types from the underlying type system ($\boldsymbol{T}$), all well-typed expressions have a type in $\boldsymbol{T}$. We provide type inference rules below.

$$\frac{v \in t}{[\![v]\!] : t}\ \text{(TypeVal)} \qquad \frac{e_1 : \text{boolean} \qquad e_2 : t \qquad e_3 : t}{\texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 : t}\ \text{(TypeCond)}$$

$$\frac{e_1 : t_1 \to t_2 \qquad e_2 : t_1}{e_1(e_2) : t_2}\ \text{(TypeFun)} \qquad \frac{e : \boldsymbol{N}}{e\,.\,a : \tau(a)}\ \text{(TypeAttr)}$$

There is nothing out of the ordinary in these type rules. Value expressions have the same type as their contained underlying value. Conditional expressions require one subexpression to have the boolean type, and the other two subexpressions to have the same type. Function expressions require subexpressions to have types that will allow underlying function application to occur during evaluation. Attribution expressions require their subexpression to be of type $\boldsymbol{N}$, as it is required to evaluate to a node. The special function τ provides the type expected from an attribute equation.

3.1.5 Semantic Rules

We provide a small-step operational semantics for the evaluation of our expression language. Proofs of type determinism, type preservation, step determinism, and progress for these semantics are given in Chapter 6.

$$\frac{e_1 \longrightarrow e_1'}{\texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \longrightarrow \texttt{IF}\ e_1'\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3}\ \text{(CondStep)}$$

$$\frac{}{\texttt{IF}\ [\![\mathit{true}]\!]\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \longrightarrow e_2}\ \text{(CondTrue)} \qquad \frac{e_1 \longrightarrow e_1'}{e_1(e_2) \longrightarrow e_1'(e_2)}\ \text{(FunStep)}$$

$$\frac{}{\texttt{IF}\ [\![\mathit{false}]\!]\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \longrightarrow e_3}\ \text{(CondFalse)} \qquad \frac{e \longrightarrow e'}{[\![v]\!](e) \longrightarrow [\![v]\!](e')}\ \text{(FunParStep)}$$

$$\frac{e \longrightarrow e'}{e\,.\,a \longrightarrow e'\,.\,a}\ \text{(AttrNodeStep)} \qquad \frac{}{[\![v_1]\!]([\![v_2]\!]) \longrightarrow [\![v_1(v_2)]\!]}\ \text{(FunApp)}$$

$$\frac{}{[\![n]\!]\,.\,a \longrightarrow \sigma(n,a)}\ \text{(AttrFetch)}$$

The general evaluation strategy used is to step all non-value subexpressions until they reach value expressions. When all subexpressions are values, the expression itself can perform its own particular evaluation step. Subexpressions are evaluated from left to right in all expression forms. The steps CondStep, CondTrue, CondFalse, FunStep, FunParStep, and AttrNodeStep are all straightforward implementations of this strategy.

$$\frac{}{[\![v_1]\!]([\![v_2]\!]) \longrightarrow [\![v_1(v_2)]\!]} \text{ (FunApp)}$$

The function application step FunApp begins with a function application expression which has two value subexpressions containing the values v_1 and v_2. This step is only possible when the expression is correctly typed according to the type rules shown in Section 3.1.4, meaning v_1 must be a function from some t_1 to some t_2, and v_2 must be some value of type t_1. The function expression steps into a value expression, where the value held is the result of applying the function v_1 to the parameter v_2. While the expressions on either side of the arrow in this rule may look similar, the expression on the left is a function application expression, and the expression on the right is a value expression containing the result of applying the given function to the given parameter.

$$\frac{}{[\![n]\!] \,.\, a \longrightarrow \sigma(n, a)} \text{ (AttrFetch)}$$

The attribute fetch step AttrFetch begins with an attribute expression containing a value expression holding some node n, and an attribute label a, and steps directly to whatever expression is returned by the context function σ for those two parameters. This is Saiga's strategy for equation selection (explored further in Section 3.2.1).

An expression returned from a context function is Saiga's version of an attribute equation. There is no differentiation between inherited and synthesised attributes in Saiga; both simply step from an attribution expression into the expression that is returned from the context function.

3.1.6 Intrinsic, Structural, and Extrinsic Attributes

Not all expressions returned from the context function will represent what is traditionally called an attribute equation. The context function is also used to fetch what we call *intrinsic attributes*. Intrinsic attributes store everything there is to know about a node; not just attributes that need to be evaluated. As nodes are not typed in Saiga, we might represent the type of a node in an intrinsic attribute. If a node has some contents, such as a string value or an integer, this is stored in an intrinsic attribute. As intrinsic attributes need no evaluation, the context would always return a value expression for an intrinsic attribute.

For example if we are considering a binary search tree, every node will have some integer value associated with it, which is created as part of tree construction. In this case, we might have some intrinsic attribute `value`, and a context function that will always return a value expression for the `value` attribute.

Another type of intrinsic attribute is what we call *structural attributes*. Just as some integer value may be an important property of a node, its children are also a property of that node, and therefore are stored in an intrinsic attribute. By the same logic, we would expect that the parent of a node is also stored in a structural attribute (usually called `parent`), although this is not necessary in purely synthesised attribute grammars. The difference between intrinsic and structural attributes is that structural attributes always return a value expression containing a node. This is the strategy we use to store an entire tree in a context function: every fact about a node is an intrinsic attribute, and the direct relationships between nodes are stored in structural attributes.

On a side note, it would be expected that any structural attributes that represent a downward relationship (*i.e.* a child relationship) and any structural attributes that represent an upward relationship (*i.e.* a parent relationship) would be mirrored. Using the attributes parent and child as an example, it would not make much sense for a context function to have some value n_2 for $\sigma(n_1, \mathsf{child})$ and have any value other than n_1 for $\sigma(n_2, \mathsf{parent})$.

We use the term *extrinsic attribute* to refer to an attribute that is not intrinsic or structural. Generally speaking, extrinsic attributes are those that have some associated equation(s). The context function and the stepping semantics do not differentiate between extrinsic, intrinsic, and structural attributes; these classifications are merely descriptors of common use patterns.

3.1.7 Simple Example

We showed an implementation of the globmin attribute for Silver, JastAdd, and Kiama in Section 2.2.1. Here we will show an implementation of globmin in Saiga. This is only a small example to demonstrate how evaluations can be traced. A more comprehensive example of use of Saiga's core semantics is given in Section 3.3, and an example of detailed type analysis is given in Chapter 7.

We will say that σ_1 is a context function that defines globmin and locmin for a binary tree, and that there is a boolean intrinsic attribute isLeaf that differentiates between fork and leaf nodes. We will assume a boolean intrinsic attribute isRoot to signify whether or not a node is the root node, and an integer intrinsic attribute value, which holds the integer value of leaf nodes. The structural attributes parent, leftChild, and rightChild are also assumed. We also assume an underlying function *min* which returns the smaller of two integers.

The attributes globmin and locmin can be defined as follows.

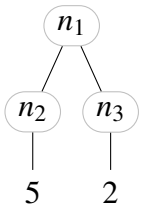
$$\begin{aligned}\sigma_1(n, \mathsf{globmin}) = \texttt{IF}\ & [\![n]\!].\mathsf{isRoot} \\ \texttt{THEN}\ & [\![n]\!].\mathsf{locmin} \\ \texttt{ELSE}\ & [\![n]\!].\mathsf{parent}.\mathsf{globmin}\end{aligned}$$

$$\begin{aligned}\sigma_1(n, \mathsf{locmin}) = \texttt{IF}\ & [\![n]\!].\mathsf{isLeaf} \\ \texttt{THEN}\ & [\![n]\!].\mathsf{value} \\ \texttt{ELSE}\ & [\![min]\!]([\![n]\!].\mathsf{leftChild}.\mathsf{locmin})([\![n]\!].\mathsf{rightChild}.\mathsf{locmin})\end{aligned}$$

The context function σ_1 will return the expressions shown above when the attributes globmin or locmin are requested during evaluation. The metavariable n is replaced with whatever node parameter is given, using standard beta reduction.

The definition here is rather verbose, using conditionals to differentiate between various attribute behaviours. Alternatives to this strategy are explored in coming sections, but for now we consider as simple a definition as possible.

To demonstrate evaluation of the globmin program in Saiga, we first define a tree to evaluate upon. Say that we have some very simple binary tree, for example as shown below.



We can assume that the context function σ_1 will return sensible results for the structural attributes leftChild, rightChild, and parent that represent the tree above. For example, $\sigma_1(n_2, \mathsf{parent})$ would return $[\![n_1]\!]$. We will not show the full definition of σ_1, we just assume that it will return sensible results that represent the tree above, including values for isLeaf and isRoot.

Evaluating the expression $[\![n_2]\!].\mathsf{globmin}$ under these conditions will yield the sequence of steps shown below. Each step is annotated with the semantic rule that allows the "highest" level of each step, with following annotations for any rules that are recursively applied. To make it easier to see what is changing over the course of many steps, we underline (the before state) and overline (the after state) each subexpression that changes, if only a subexpression changes over a step.

In this example, the starting expression $[\![n_2]\!].\mathsf{globmin}$ evaluates to the value expression $[\![2]\!]$ in 22 steps. Once evaluation reaches a value expression, there are no more steps that can be taken, and evaluation is considered complete. It is proven that value expressions can not take steps in Theorem 2 (Section 6.2).

$[\![n_2]\!].\mathsf{globmin}$

(1) $\longrightarrow$ (AttrFetch)

$\mathtt{IF}\ \underline{[\![n_2]\!].\mathsf{isRoot}}\ \mathtt{THEN}\ [\![n_2]\!].\mathsf{locmin}\ \mathtt{ELSE}\ [\![n_2]\!].\mathsf{parent}.\mathsf{globmin}$

(2) $\longrightarrow$ (CondStep, AttrFetch)

$\mathtt{IF}\ \overline{[\![\mathit{false}]\!]}\ \mathtt{THEN}\ [\![n_2]\!].\mathsf{locmin}\ \mathtt{ELSE}\ [\![n_2]\!].\mathsf{parent}.\mathsf{globmin}$

(3) $\longrightarrow$ (CondFalse)

$\underline{[\![n_2]\!].\mathsf{parent}}.\mathsf{globmin}$

(4) $\longrightarrow$ (AttrNodeStep, AttrFetch)

$\overline{[\![n_1]\!]}.\mathsf{globmin}$

(5) $\longrightarrow$ (AttrFetch)

$\mathtt{IF}\ \underline{[\![n_1]\!].\mathsf{isRoot}}\ \mathtt{THEN}\ [\![n_1]\!].\mathsf{locmin}\ \mathtt{ELSE}\ [\![n_1]\!].\mathsf{parent}.\mathsf{globmin}$

(6) $\longrightarrow$ (CondStep, AttrFetch)

$\mathtt{IF}\ \overline{[\![\mathit{true}]\!]}\ \mathtt{THEN}\ [\![n_1]\!].\mathsf{locmin}\ \mathtt{ELSE}\ [\![n_1]\!].\mathsf{parent}.\mathsf{globmin}$

(7) $\longrightarrow$ (CondTrue)

$[\![n_1]\!].\mathsf{locmin}$

(8) $\longrightarrow$ (AttrFetch)

$\mathtt{IF}\ \underline{[\![n_1]\!].\mathsf{isLeaf}}\ \mathtt{THEN}\ [\![n_1]\!].\mathsf{value}$
$\mathtt{ELSE}\ [\![\mathit{min}]\!]([\![n_1]\!].\mathsf{leftChild}.\mathsf{locmin})([\![n_1]\!].\mathsf{rightChild}.\mathsf{locmin})$

(9) $\longrightarrow$ (CondStep, AttrFetch)

$\mathtt{IF}\ \overline{[\![\mathit{false}]\!]}\ \mathtt{THEN}\ [\![n_1]\!].\mathsf{value}$

$\texttt{ELSE}\ [\![min]\!]([\![n_1]\!].\mathsf{leftChild}.\mathsf{locmin})([\![n_1]\!].\mathsf{rightChild}.\mathsf{locmin})$

(10) $\longrightarrow$ (CondFalse)

$[\![min]\!]([\![n_1]\!].\mathsf{leftChild}.\mathsf{locmin})([\![n_1]\!].\mathsf{rightChild}.\mathsf{locmin})$

(11) $\longrightarrow$ (FunStep, FunParStep, AttrNodeStep, AttrFetch)

$[\![min]\!]([\![n_2]\!].\mathsf{locmin})([\![n_1]\!].\mathsf{rightChild}.\mathsf{locmin})$

(12) $\longrightarrow$ (FunStep, FunParStep, AttrFetch)

$[\![min]\!](\texttt{IF}\ [\![n_2]\!].\mathsf{isLeaf}\ \texttt{THEN}\ [\![n_2]\!].\mathsf{value}$
$\texttt{ELSE}\ [\![min]\!]([\![n_2]\!].\mathsf{leftChild}.\mathsf{locmin})([\![n_2]\!].\mathsf{rightChild}.\mathsf{locmin}))$
$([\![n_1]\!].\mathsf{rightChild}.\mathsf{locmin})$

(13) $\longrightarrow$ (FunStep, FunParStep, CondStep, AttrFetch)

$[\![min]\!](\texttt{IF}\ [\![true]\!]\ \texttt{THEN}\ [\![n_2]\!].\mathsf{value}$
$\texttt{ELSE}\ [\![min]\!]([\![n_2]\!].\mathsf{leftChild}.\mathsf{locmin})([\![n_2]\!].\mathsf{rightChild}.\mathsf{locmin}))$
$([\![n_1]\!].\mathsf{rightChild}.\mathsf{locmin})$

(14) $\longrightarrow$ (FunStep, FunParStep, CondTrue)

$[\![min]\!]([\![n_2]\!].\mathsf{value})([\![n_1]\!].\mathsf{rightChild}.\mathsf{locmin})$

(15) $\longrightarrow$ (FunStep, FunParStep, AttrFetch)

$[\![min]\!]([\![5]\!])([\![n_1]\!].\mathsf{rightChild}.\mathsf{locmin})$

(16) $\longrightarrow$ (FunStep, FunApp)

$[\![min(5)]\!]([\![n_1]\!].\mathsf{rightChild}.\mathsf{locmin})$

(17) $\longrightarrow$ (FunParStep, AttrNodeStep, AttrFetch)

$[\![min(5)]\!]([\![n_3]\!].\mathsf{locmin})$

(18) $\longrightarrow$ (FunStep, AttrFetch)

$[\![min(5)]\!](\texttt{IF}\ [\![n_3]\!].\mathsf{isLeaf}\ \texttt{THEN}\ [\![n_3]\!].\mathsf{value}$
$\texttt{ELSE}\ [\![min]\!]([\![n_3]\!].\mathsf{leftChild}.\mathsf{locmin})([\![n_3]\!].\mathsf{rightChild}.\mathsf{locmin}))$

(19) $\longrightarrow$ (FunStep, CondStep, AttrFetch)

$[\![min(5)]\!](\texttt{IF}\ [\![true]\!]\ \texttt{THEN}\ [\![n_3]\!].\mathsf{value}$
$\texttt{ELSE}\ [\![min]\!]([\![n_3]\!].\mathsf{leftChild}.\mathsf{locmin})([\![n_3]\!].\mathsf{rightChild}.\mathsf{locmin}))$

(20) $\longrightarrow$ (FunStep, CondTrue)

$[\![min(5)]\!]([\![n_3]\!].\mathsf{value})$

(21) $\longrightarrow$ (FunStep, AttrFetch)

$[\![min(5)]\!]([\![2]\!])$

(22) $\longrightarrow$ (FunApp)

$[\![2]\!]$

This is quite a lengthy process, but we have examined every fine detail of evaluation. The notation shortcuts demonstrated in coming sections will make these evaluation traces considerably more concise, and the use of the multistep and big step relations (Sections 3.2.8

and 3.2.9) further reduce the footwork required to trace an evaluation.

3.2 Discussion

In this section we will discuss some details, design decisions, and caveats of the calculus presented in Section 3.1.

3.2.1 Equation Selection and the Context Function

Central to attribute grammar evaluation is the equation selection problem. Generally speaking, for an attribute to be evaluated on a node, first the evaluator must select an appropriate equation, then that equation must be used to find a value for the attribute. However, the equation selection process can be quite sophisticated; sometimes there is more semantic logic encoded into the equation selection process than in the equation itself.

Consider the role a type system plays in a general purpose programming language. In many programs, types merely enable safety checks; making sure the programmer does not assign a value of the wrong type to a variable. In some types of object-oriented programming applications, the type of a reference variable defines which version of a method will be used; type information here is more significant. There are applications where everything that is known about an entity is in its type; consider the Propositional universe in Coq, where the evaluator discards all non-type information about an entity, only caring that a value *exists*, not caring about what that value is; type information here is everything.

It is possible to encode zero behaviour into a program's types, and it is possible to encode almost the entirety of a program's behaviour into its types. The same is true for equation selection systems. Some attribute grammar programs use only the most basic equation selection features and encode their entire program logic into attribute equations. Other attribute programs use equation selection features to express a significant portion of the program logic, and only write simple attribute equations.

Let us consider a simple concrete example. Let's say we have some tree of arbitrary size, whose root node has some attribute k. It is desired that every node in the tree have access to this same value via an attribute getK.

Encoding the traversal to the root as part of evaluation would be straightforward, as shown below.

$$\sigma_1(n, \mathsf{getK}) = \texttt{IF } [\![n]\!].\mathsf{isRoot} \texttt{ THEN } [\![n]\!].\mathsf{k} \texttt{ ELSE } [\![n]\!].\mathsf{parent.getK}$$

However, we can define the context function however we want to define it. We could define some σ as follows.

$$\sigma_2(\_, \mathsf{getK}) = [\![n_{root}]\!].\mathsf{k}, \text{ where } n_{root} \text{ is the root node}$$

If it is always the root node that is required, then this approach is useful. However, it is common to examine more fine-grained contextual information for equation selection. Let's say that perhaps there are multiple sub-roots, whose k is 'copied' to its subtree. These sub-root nodes return *true* for the intrinsic attribute isSubRoot. An approach like the following

will be useful in such a circumstance.

$$\sigma_3(n, \texttt{getK}) = \begin{cases} [\![n]\!].\texttt{k} & \sigma_3(n, \texttt{isSubRoot}) = [\![\mathit{true}]\!] \\ [\![n_p]\!].\texttt{getK} & \sigma_3(n, \texttt{parent}) = [\![n_p]\!] \\ [\![0]\!] & \text{otherwise} \end{cases}$$

In the first case, equation selection examines the expression returned by the intrinsic attribute isSubRoot. If this expression is a value expression, and that value expression contains the value *true*, then the expression $[\![n]\!].\texttt{k}$ is returned. Note that the step relation is not used here; it is not possible for the context function to perform Saiga evaluation. However, we do allow the definition of a context function to interrogate the values returned by *intrinsic* attributes. If $\sigma_3(n, \texttt{isSubRoot})$ returned some non-value expression e_1, this expression would not be evaluated during attribute selection; the "selector" would not match.

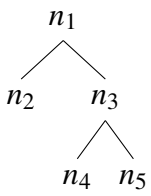
We perform a similar operation in the second case, interrogating the context via reflection for the value returned by parent. This is a form of autocopy inherited attribute, similar to autocopy attributes from Silver or JastAdd.

Note that σ must be a mathematical function – that is, it must be defined for all inputs. With the definition of getK in σ_3 above, if the definition of parent also interrogates the value of getK, it is possible that a cycle would be formed, making σ_3 undefined for some inputs and therefore not a properly-defined function. Therefore it is best practice to only interrogate strictly intrinsic attributes as part of equation selection.

These are only suggested implementation techniques; the context function can be any mathematical function that returns an expression for any input node and attribute. The user can implement this function however they like, or leave it as a black box with only some assertions about it used to guide their reasoning. We explore more equation selection possibilities in Section 3.2.2.

Demonstration

To demonstrate how the three given specifications of getK differ, we will evaluate the expression $[\![n_4]\!].\texttt{getK}$ in each of the context functions σ_1, σ_2, and σ_3. In each case, we assume that the context function holds values for appropriate structural attributes that represent the following tree.



First we evaluate using the definition of getK in σ_1. This evaluation reaches the expression $[\![n_1]\!].\texttt{k}$ in 11 steps.

$[\![n_4]\!].\texttt{getK}$

(1) $\longrightarrow$ (AttrFetch)

`IF` $[\![n_4]\!]$`.isRoot THEN` $[\![n_4]\!]$`.k ELSE` $[\![n_4]\!]$`.parent.getK`

(2) $\longrightarrow$ (CondStep, AttrFetch)

$$
\begin{aligned}
&\texttt{IF } \overline{[\![\mathit{false}]\!]} \texttt{ THEN } [\![n_4]\!]\texttt{.k ELSE } [\![n_4]\!]\texttt{.parent.getK} \\
(3) \longrightarrow\ & \text{(CondFalse)} \\
&\underline{[\![n_4]\!]\texttt{.parent}}\texttt{.getK} \\
(4) \longrightarrow\ & \text{(AttrNodeStep, AttrFetch)} \\
&\overline{[\![n_3]\!]}\texttt{.getK} \\
(5) \longrightarrow\ & \text{(AttrFetch)} \\
&\texttt{IF } \underline{[\![n_3]\!]\texttt{.isRoot}} \texttt{ THEN } [\![n_3]\!]\texttt{.k ELSE } [\![n_3]\!]\texttt{.parent.getK} \\
(6) \longrightarrow\ & \text{(CondStep, AttrFetch)} \\
&\texttt{IF } \overline{[\![\mathit{false}]\!]} \texttt{ THEN } [\![n_3]\!]\texttt{.k ELSE } [\![n_3]\!]\texttt{.parent.getK} \\
(7) \longrightarrow\ & \text{(CondFalse)} \\
&\underline{[\![n_3]\!]\texttt{.parent}}\texttt{.getK} \\
(8) \longrightarrow\ & \text{(AttrNodeStep, AttrFetch)} \\
&\overline{[\![n_1]\!]}\texttt{.getK} \\
(9) \longrightarrow\ & \text{(AttrFetch)} \\
&\texttt{IF } \underline{[\![n_1]\!]\texttt{.isRoot}} \texttt{ THEN } [\![n_1]\!]\texttt{.k ELSE } [\![n_1]\!]\texttt{.parent.getK} \\
(10) \longrightarrow\ & \text{(CondStep, AttrFetch)} \\
&\texttt{IF } \overline{[\![\mathit{true}]\!]} \texttt{ THEN } [\![n_1]\!]\texttt{.k ELSE } [\![n_1]\!]\texttt{.parent.getK} \\
(11) \longrightarrow\ & \text{(CondTrue)} \\
&[\![n_1]\!]\texttt{.k}
\end{aligned}
$$

Next we evaluate $[\![n_4]\!]\texttt{.getK}$ using the definition of getK in σ_2. The expression $[\![n_1]\!]\texttt{.k}$ is reached in a single step.

$$
\begin{aligned}
&[\![n_4]\!]\texttt{.getK} \\
(1) \longrightarrow\ & \text{(AttrFetch,as } n_{root} = n_1\text{)} \\
&[\![n_1]\!]\texttt{.k}
\end{aligned}
$$

Finally we evaluate $[\![n_4]\!]\texttt{.getK}$ using the definition of getK in σ_3, assuming the only sub-root in the tree is n_1. The expression $[\![n_1]\!]\texttt{.k}$ is reached in three steps.

$$
\begin{aligned}
&[\![n_4]\!]\texttt{.getK} \\
(1) \longrightarrow\ & \text{(AttrFetch, as } \sigma_3(n_4, \texttt{isSubRoot}) \neq [\![\mathit{true}]\!] \text{ and } \sigma_3(n_4, \texttt{parent}) = [\![n_3]\!]\text{)} \\
&[\![n_3]\!]\texttt{.getK} \\
(2) \longrightarrow\ & \text{(AttrFetch, as } \sigma_3(n_3, \texttt{isSubRoot}) \neq [\![\mathit{true}]\!] \text{ and } \sigma_3(n_3, \texttt{parent}) = [\![n_1]\!]\text{)} \\
&[\![n_1]\!]\texttt{.getK} \\
(3) \longrightarrow\ & \text{(AttrFetch, as } \sigma_3(n_1, \texttt{isSubRoot}) = [\![\mathit{true}]\!]\text{)} \\
&[\![n_1]\!]\texttt{.k}
\end{aligned}
$$

There is no "winning" implementation here. The solution in σ_1 is very explicit, with many of Saiga's evaluation steps detailing what might be considered "equation selection"

processes such as checking intrinsic flags like isRoot. The solution in σ_2 skips over all kinds of boiler plate, and jumps immediately to an expression based at the root of the tree n_1, but assumes prior knowledge of the identity of the root node. The solution in σ_3 sits in between these extremes, skipping some of the boiler plate involved in checking intrinsic properties, but still making its way up the structure of the tree one level at a time. In practice, we find approaches similar to that in σ_3 to be a good balance, where evaluation is less verbose, but no "magic" is happening. We use a similar strategy in the notations explored in Section 3.2.2.

3.2.2 Equation Selection and Node Types

It is common practice that equations be selected based on the type of the node being evaluated upon. For example, some attribute eval may have one equation for an *AddExp* node, and a different equation for a *MultExp* node. We have opted not to include node types in our calculus, and allowed N to remain abstract.

There are two reasons behind this decision: simplicity and flexibility. The baseline for equation selection is simpler without node types; the context function is not required to care about the type of a node to select an equation. Further, allowing the context function to be as general as it is opens up the possibility of any kind of relationship between node, type, and equation that the user wants, instead of a predefined one.

The openness of the context function design allows for production-based, symbol-based, and attribute-based equation selection strategies to be implemented, even synchronously.

Symbol-Based Equation Selection

JastAdd uses symbol-based equation selection, as discussed in Section 2.2.1. For example, synthesised attributes in the JastAdd attribute grammar system may be encoded as follows, where the node types *AddExp* and *MultExp* have different equations for eval.

```
1  syn int AddExp.eval()  = getLeft().eval() + getRight().eval();
2  syn int MultExp.eval() = getLeft().eval() * getRight().eval();
```

Let us consider the ways this can be encoded in Saiga. If we want to write a proof about the process of selecting attributes, we may want to include this selection as part of the evaluation logic. Assuming the intrinsic attribute nodeType, along with underlying helper functions *isAdd* and *isMult* for matching against these types, we could implement eval as follows.

$$
\begin{aligned}
\sigma(n,\mathsf{eval}) = \ & \mathtt{IF}\ [\![\mathit{isAdd}]\!]([\![n]\!].\mathsf{nodeType}) \\
& \mathtt{THEN}\ [\![\mathit{plus}]\!]([\![n]\!].\mathsf{left.eval})([\![n]\!].\mathsf{right.eval}) \\
& \mathtt{ELSE\ IF}\ [\![\mathit{isMult}]\!]([\![n]\!].\mathsf{nodeType}) \\
& \mathtt{THEN}\ [\![\mathit{mult}]\!]([\![n]\!].\mathsf{left.eval})([\![n]\!].\mathsf{right.eval}) \\
& \mathtt{ELSE}\ [\![0]\!]
\end{aligned}
$$

This implements the same semantics as the JastAdd example given above, but the context function returns a large expression containing all possible targeted attribute expressions. The user may prefer for the context function to perform this selection process implicitly and only

return targeted attribute expressions. In this case, we could define eval as follows.

$$\sigma(n,\mathsf{eval}) = \begin{cases} [\![plus]\!]([\![n]\!].\mathsf{left.eval})([\![n]\!].\mathsf{right.eval}) & \sigma(n,\mathsf{nodeType}) = [\![add]\!] \\ [\![mult]\!]([\![n]\!].\mathsf{left.eval})([\![n]\!].\mathsf{right.eval}) & \sigma(n,\mathsf{nodeType}) = [\![mult]\!] \\ [\![0]\!] & \text{otherwise} \end{cases}$$

It is assumed here that nodeType is an *intrinsic* attribute. Recall that the context function cannot perform expression evaluation, and will only match against reflected expressions that hold values. If there is a possibility of some node whose nodeType is evaluated from a more complex expression, one could combine the two approaches shown, so that the attribute nodeType is evaluated if a value is not found. Such an approach is shown below, but is quite verbose. It is possible to abstract away this notation into the notations explored later in this section.

$$\sigma(n,\mathsf{eval}) = \begin{cases} [\![plus]\!]([\![n]\!].\mathsf{left.eval})([\![n]\!].\mathsf{right.eval}) & \sigma(n,\mathsf{nodeType}) = [\![add]\!] \\ [\![mult]\!]([\![n]\!].\mathsf{left.eval})([\![n]\!].\mathsf{right.eval}) & \sigma(n,\mathsf{nodeType}) = [\![mult]\!] \\ \mathtt{IF}\ [\![isAdd]\!]([\![n]\!].\mathsf{nodeType}) & \text{otherwise} \\ \ \mathtt{THEN}\ [\![plus]\!]([\![n]\!].\mathsf{left.eval})([\![n]\!].\mathsf{right.eval}) & \\ \ \mathtt{ELSE\ IF}\ [\![isMult]\!]([\![n]\!].\mathsf{nodeType}) & \\ \ \mathtt{THEN}\ [\![mult]\!]([\![n]\!].\mathsf{left.eval})([\![n]\!].\mathsf{right.eval}) & \\ \ \mathtt{ELSE}\ [\![0]\!] & \end{cases}$$

Production-Based Equation Selection

It is also common to define attribute equations based on the grammar rule used to parse and construct a node. Consider the following example, using syntax from the Silver attribute grammar system, which uses production-based equation selection, as discussed in Section 2.2.1.

```
1  concrete production add
2  sum::Expr ::= e::Expr '+' t::Term
3  { sum.eval = e.eval + t.eval ; }
4
5  concrete production prd
6  prd::Expr ::= e::Expr '*' t::Term
7  { prd.eval = e.eval * t.eval ; }
```

In this circumstance, it appears that nodes being created in both productions are *Expr* nodes, so the type of a node is not defining the attribute equation; its production rule is. To implement this kind of equation selection in Saiga, we would need to assume that the context function knows what production rule was used to build each node. This could be implemented with an intrinsic attribute similar to the nodeType attribute, called nodeProd for example. So far this does not differ significantly from the symbol based approach, using the same strategy with a different intrinsic attribute.

$$\sigma(n,\mathsf{eval}) = \begin{cases} [\![plus]\!]([\![n]\!].\mathsf{left.eval})([\![n]\!].\mathsf{right.eval}) & \sigma(n,\mathsf{nodeProd}) = [\![addProd]\!] \\ [\![mult]\!]([\![n]\!].\mathsf{left.eval})([\![n]\!].\mathsf{right.eval}) & \sigma(n,\mathsf{nodeProd}) = [\![multProd]\!] \\ \ldots & \end{cases}$$

There are times where not only the type of a node matters, but the node's position in a subtree. This is typically the situation created by inherited attributes in a production-based selection scheme. Consider the following simplified snippet from an attribute grammar written in Silver, and the same attribute written in JastAdd.

```
1  // in Silver
2  call::Expr ::= c::Expr '.' i::Expr
3  { c.decl = 4 ;
4    i.decl = 6 ; }
5
6  // in JastAdd
7  inh Integer Expr.decl();
8  eq CallExpr.getLeftExpr().decl() = 4;
9  eq CallExpr.getRightExpr().decl() = 6;
```

Above we have given decl the value 4 for *Expr* nodes who are the left child of a *CallExpr* node, and the value 6 for *Expr* nodes who are the right child of a *CallExpr* node. For simplicity here we will assume that a *CallExpr* node has a nodeType of *call*. We could describe equivalent equation selection as follows.

$$\sigma(n,\mathsf{decl}) = \begin{cases} [\![4]\!] & \sigma(n,\mathsf{nodeType}) = [\![expr]\!] \text{ and } \sigma(n,\mathsf{parent}) = [\![n_p]\!] \\ & \text{and } \sigma(n_p,\mathsf{nodeType}) = [\![call]\!] \text{ and } \sigma(n_p,\mathsf{leftexpr}) = [\![n]\!] \\ [\![6]\!] & \sigma(n,\mathsf{nodeType}) = [\![expr]\!] \text{ and } \sigma(n,\mathsf{parent}) = [\![n_p]\!] \\ & \text{and } \sigma(n_p,\mathsf{nodeType}) = [\![call]\!] \text{ and } \sigma(n_p,\mathsf{rightexpr}) = [\![n]\!] \\ [\![0]\!] & \text{otherwise} \end{cases}$$

The above definition of decl may look complex, but is made from a number of simple selectors using only intrinsic attributes. Our goal is a simple and general calculus, and here we leverage our simple context functions to express a particular kind of equation selection. In the first case, given some n, the context function is reflecting on its value for $(n, \mathsf{nodeType})$ (intrinsic) to ensure that n is an *expr* node. Then, it reflects on parent (intrinsic) to access the identity of the parent. Using this parent node, it reflects on nodeType (again, intrinsic) to ensure that the parent is a *call* node. Finally, it reflects on leftexpr (intrinsic), to compare against the original node n. This verbose specification is given a more user-friendly notation with *selector notations*, introduced below.

This entire selection process could be replaced with the following expression. However, it is often desirable for such decisions to be made as part of attribute selection, and not as part of evaluation.

$$
\begin{aligned}
\sigma(n, \mathsf{decl}) = \texttt{IF}\ & [\![and]\!]([\![isExpr]\!]([\![n]\!].\mathsf{nodeType})) \\
& ([\![and]\!]([\![isCall]\!]([\![n]\!].\mathsf{parent.nodeType})) \\
& ([\![nodeEq]\!]([\![n]\!])([\![n]\!].\mathsf{parent.leftexpr}))) \\
\texttt{THEN}\ & [\![4]\!] \\
\texttt{ELSE IF}\ & [\![and]\!]([\![isExpr]\!]([\![n]\!].\mathsf{nodeType})) \\
& ([\![and]\!]([\![isCall]\!]([\![n]\!].\mathsf{parent.nodeType})) \\
& ([\![nodeEq]\!]([\![n]\!])([\![n]\!].\mathsf{parent.rightexpr}))) \\
\texttt{THEN}\ & [\![6]\!] \\
\texttt{ELSE}\ & [\![0]\!]
\end{aligned}
$$

Selector Notations

Given that these kinds of equation selection strategies are common, and especially given that they can be verbose (as in the example above), we use some notations to make writing these selections more convenient and easier to read. Below we show seven *selectors*, representing selection schemes we have shown, and some extras. For each of these, we will show an expanded selection notation, as well as an expression that would evaluate to the same result.

The definition of the attr attribute is given below, using all seven selector notations, which will be explained individually.

$$
\sigma(n, \mathsf{attr}) = \begin{cases}
e_a & n = n_{null} \\
e_b & \text{(typeOne)} \\
e_c & \text{(typeTwo).*} \\
e_d & \text{(typeThree).childAttr} \\
e_e & \text{(typeFour).children*} \\
\text{inherited} & \\
e_f & \text{otherwise}
\end{cases}
$$

In the first selector, the expression e_a is returned when the input node is n_{null}. The same semantics could be achieved by the definition of attr below as an expression, assuming a boolean underlying function *nodeEq* which checks two nodes for equality.

$$
\sigma(n, \mathsf{attr}) = \texttt{IF}\ [\![nodeEq]\!]([\![n]\!])([\![n_{null}]\!])\ \texttt{THEN}\ e_a\ \texttt{ELSE}\ \ldots
$$

In the second selector, the expression e_b is returned when the input node is of type *typeOne*. The selector notation is equivalent to the following.

$$
\sigma(n, \mathsf{attr}) = \begin{cases}
e_b & \sigma(n, \mathsf{nodeType}) = [\![typeOne]\!] \\
\ldots &
\end{cases}
$$

The same semantics could be achieved without reflection by returning the following expression for attr.

$$\sigma(n, \mathsf{attr}) = \texttt{IF } [\![isTypeOne]\!]([\![n]\!].\mathsf{nodeType}) \texttt{ THEN } e_b \texttt{ ELSE } \ldots$$

In the third selector, the expression e_c is returned when the input node's parent is of type *typeTwo*. The selector notation is equivalent to the following.

$$\sigma(n, \mathsf{attr}) = \begin{cases} e_c & \sigma(n, \mathsf{parent}) = [\![n_p]\!] \text{ and } \sigma(n_p, \mathsf{nodeType}) = [\![typeTwo]\!] \\ \ldots \end{cases}$$

The same semantics could be achieved without reflection by returning the following expression for attr.

$$\sigma(n, \mathsf{attr}) = \texttt{IF } [\![isTypeTwo]\!]([\![n]\!].\mathsf{parent.nodeType}) \texttt{ THEN } e_c \texttt{ ELSE } \ldots$$

In the fourth selector, the expression e_d is returned when the input node's parent is of type *typeThree*, and the input node is the node returned by the parent's childAttr attribute. The selector notation is equivalent to the following.

$$\sigma(n, \mathsf{attr}) = \begin{cases} e_d & \sigma(n, \mathsf{parent}) = [\![n_p]\!] \\ & \text{and } \sigma(n_p, \mathsf{nodeType}) = [\![typeThree]\!] \\ & \text{and } \sigma(n_p, \mathsf{childAttr}) = [\![n]\!] \\ \ldots \end{cases}$$

The same semantics could be achieved without reflection by returning the following expression for attr, assuming an underlying function *and* representing boolean conjunction.

$$\begin{aligned} \sigma(n, \mathsf{attr}) = \texttt{IF } & [\![and]\!]([\![isTypeThree]\!]([\![n]\!].\mathsf{parent.nodeType})) \\ & ([\![nodeEq]\!]([\![n]\!])([\![n]\!].\mathsf{parent.childAttr})) \\ \texttt{THEN } & e_d \\ \texttt{ELSE } & \ldots \end{aligned}$$

In the fifth selector, the expression e_e is returned when the input node's parent is of type *typeFour*, and the input node is contained in the list of nodes returned by the parent's children attribute. We assume a boolean underlying function *nodeContained*, which checks if a node is present in a list of nodes. The selector notation is equivalent to the following.

$$\sigma(n, \mathsf{attr}) = \begin{cases} e_e & \sigma(n, \mathsf{parent}) = [\![n_p]\!] \\ & \text{and } \sigma(n_p, \mathsf{nodeType}) = [\![typeFour]\!] \\ & \text{and } \sigma(n_p, \mathsf{children}) = [\![l]\!] \\ & \text{and } nodeContained(n)(l) = true \\ \ldots \end{cases}$$

The same semantics could be achieved without reflection by returning the following expression for attr.

$$\begin{aligned}\sigma(n,\texttt{attr}) = \texttt{IF}\ & [\![and]\!]([\![isTypeFour]\!]([\![n]\!].\texttt{parent.nodeType})) \\ & ([\![nodeContained]\!]([\![n]\!])([\![n]\!].\texttt{parent.children})) \\ \texttt{THEN}\ & e_e \\ \texttt{ELSE}\ & \ldots\end{aligned}$$

If none of the above selectors have matched, the "inherited" case will be used. This selector returns an expression that calls the same attribute on the node's parent, assuming a node's parent is available as a value in the context. The notation shown is equivalent to the following.

$$\sigma(n,\texttt{attr}) = \begin{cases}[\![n_p]\!].\texttt{attr} & \sigma(n,\texttt{parent}) = [\![n_p]\!] \\ \ldots\end{cases}$$

The same semantics could be achieved without reflection by returning the following expression for attr.

$$\sigma(n,\texttt{attr}) = [\![n]\!].\texttt{parent.attr}$$

If none of the previous selectors matched, then the final case e_f will be returned.

These selector notations are not adding extra functionality to the context function, they are merely a notational convenience for common use patterns. They assume the existence of the attributes nodeType and parent. If used in a context without intrinsic attributes nodeType and parent, these notations do not make sense.

Selector Example

To demonstrate the use of selectors, we will re-implement the globmin example from Section 3.1.7. We assume that nodes can have one of three types (in nodeType): *leaf*, *fork*, and *root*. We first define locmin as follows.

$$\sigma(n,\texttt{locmin}) = \begin{cases}[\![n]\!].\texttt{value} & \text{(leaf)} \\ [\![min]\!]([\![n]\!].\texttt{left.locmin})([\![n]\!].\texttt{right.locmin}) & \text{otherwise}\end{cases}$$

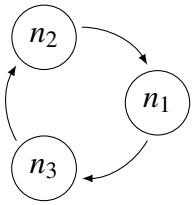
We then define globmin as follows.

$$\sigma(n,\texttt{globmin}) = \begin{cases}[\![n]\!].\texttt{locmin} & \text{(root)} \\ \text{inherited}\end{cases}$$

The attributes locmin and globmin represent the same semantics as those given in Section 3.1.7, but we have used some useful notations to more closely mimic the equation definition schemes used in real-world attribute grammar systems. The notations shown here demonstrate that Saiga is general enough to express attribute grammar programs using a variety of notations.

Invalid Context Functions and Non-Terminating Evaluation

Consider the case where nodes exist that do not represent a tree, but a cyclic graph. If a node's parent is itself, or if a node's parent's parent is itself, or similar, then "following" the parent chain can lead to non-terminating evaluation.



We will work with a "tree" (not actually a tree) represented by the graph above, with the arrows between nodes representing the parent attribute relationship. Let us assume that all nodes in this graph have the type *leaf*. We will evaluate $[\![n_2]\!].\mathsf{globmin}$ in this context, using the definition of globmin from the previous example.

$$
\begin{aligned}
& [\![n_2]\!].\mathsf{globmin} \\
(1) \longrightarrow\ & \text{(AttrFetch)} \\
& [\![n_1]\!].\mathsf{globmin} \\
(2) \longrightarrow\ & \text{(AttrFetch)} \\
& [\![n_3]\!].\mathsf{globmin} \\
(3) \longrightarrow\ & \text{(AttrFetch)} \\
& [\![n_2]\!].\mathsf{globmin}
\end{aligned}
$$

Quickly we have arrived at the expression we started with. This evaluation will not terminate. However, this is acceptable – we do not build machinery to disallow the definition of an attribute that will not terminate. Similarly, it is possible to build an attribute grammar program in Kiama or JastAdd that will never terminate.

Consider now a definition of globmin that reflects on itself not just to decide an attribute equation, but to produce such an equation. The idea here is that the context function itself will follow the parent chain as far is it can, returning an expression when it reaches the 'top'.

$$
\sigma(n, \mathsf{globmin}) = \begin{cases} [\![n]\!].\mathsf{locmin} & \text{(root)} \\ \sigma(n_p, \mathsf{globmin}) & \sigma(n, \mathsf{parent}) = [\![n_p]\!] \end{cases}
$$

The advantage of such an approach is that the context function will follow the parent chain all the way to its terminus in one call. Asking for some deep *leaf* node's globmin equation will 'immediately' return an expression based at the root. The issue here is when there is a cycle in the parent chain. If the nodes in the above cycle are present in some context function σ, then $\sigma(n_2, \mathsf{globmin})$ is not defined, as it is infinitely recursive.

$$
\sigma(n_2, \mathsf{globmin}) = \sigma(n_1, \mathsf{globmin}) = \sigma(n_3, \mathsf{globmin}) = \sigma(n_2, \mathsf{globmin}) = \ldots
$$

As a context function must be a mathematical function, this σ is not a valid context function. It is still possible to create invalid context functions without including recursive calls on the "left hand" of our attribute definitions, but it is not as common. Generally speaking, it is always safe to reflect on attributes that are known to be simply defined - that is, no special selection logic is applied for that attribute. If parent is always a simply defined attribute (as all intrinsic attributes are expected to be), then reflecting on the parent definition during attribute selection will not cause undefined outputs and therefore invalid context functions.

Further, recall that the context function can be a black box function. The user is free to define any function they want for σ, as long as it produces an output for every input. Here we are merely discussing the strengths and weaknesses of the notation standards introduced in this section.

3.2.3 Conditional Expressions

$$e \ ::= \ \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \qquad \text{conditional}$$

One of our goals was to create as simple a calculus as possible, part of which means keeping the expression language as simple as possible. We explored a few options for avoiding a conditional expression, but it is not possible to implement non-strict conditional evaluation without a specific production for this. If we had some underlying function *ifthenelse* which takes the same three parameters that a conditional expression takes, the function application semantics in Saiga would always evaluate both the *then* and *else* subexpressions to a value, independently of the value of the conditional subexpression. This is not acceptable behaviour when a conditional is used to guard against infinite circularity [26].

3.2.4 Multiple-Parameter Functions

$$e \ ::= \ e_1(e_2) \qquad \text{function application}$$

A function application expression is made up of two subexpressions: a function and a parameter. As shown in some examples already, we use currying to implement multi-parameter functions. Consider the following example.

$$\underline{\underline{[\![plus]\!]([\![5]\!])}}\,\underline{([\![99]\!])}$$

This is a function application expression. The "function" subexpression (a function application expression itself) is doubly underlined above, and the "parameter" subexpression is singly underlined. *plus* is an underlying function of type $integer \to integer \to integer$. Below we show the same expression with extra parentheses to make the structure clearer, although these parentheses are not strictly necessary.

$$([\![plus]\!]([\![5]\!]))([\![99]\!])$$

The above expression will take two steps to evaluate. The first step will be derived from FunApp, where the function subexpression steps from a function application expression to a value expression, as shown.

$$\dfrac{\dfrac{}{[\![plus]\!]([\![5]\!]) \longrightarrow [\![plus(5)]\!]}\ \text{(FunApp)}}{[\![plus]\!]([\![5]\!])([\![99]\!]) \longrightarrow [\![plus(5)]\!]([\![99]\!])}\ \text{(FunApp)}$$

Here the first subexpression has stepped to a value expression. The value expression $[\![plus(5)]\!]$ contains the value $plus(5)$, which is of type $integer \to integer$. We use this notation to describe a partially applied function, as discussed in Section 3.1.1. The next step is to apply the newly formed function application expression.

$$\dfrac{}{[\![plus(5)]\!]([\![99]\!]) \longrightarrow [\![plus(5)(99)]\!]}\ \text{(FunApp)}$$

The value $plus(5)(99)$ is the underlying value 104. We could also write the above step as shown below, where we reduce the underlying function application to its final value. As our semantics are not interested in beta reduction in the underlying system, we can assume that $plus(5)(99)$ and 104 are exactly the same thing.

$$\frac{}{[\![plus(5)]\!]([\![99]\!]) \longrightarrow [\![104]\!]} \text{ (FunApp)}$$

As it takes one semantic step to apply a function to its (single) parameter, it therefore takes two semantic steps to apply a two-parameter function to its two parameters. This evaluation occurs in the following order: evaluate first parameter expression; (partially) apply function; evaluate next parameter expression; (partially) apply function; and so on.

3.2.5 Tuples

Alternatively to multi-parameter functions via currying, it is possible to use functions that take a single tuple as an argument. While not often useful for direct function application, this technique is often required when using parameterised attributes, which are discussed in Section 4.1.

As tuples are used often enough, we will define some notations to more conveniently express tuple construction. We begin with an underlying function *tuple*, which takes two arguments and produces a tuple from them. Since (in this thesis) we assume only monomorphic underlying types, a different *tuple* function is required for each type of tuple that is being constructed. We will assume a *tuple* function is available for combining all types in $\boldsymbol{T}$.

We define some notations to describe tuple construction, as follows.

$$(e_1\,,\, e_2) \quad = \quad [\![tuple]\!](e_1)(e_2)$$

As curried function application requires the first parameter to be applied before the second parameter is evaluated, we cannot evaluate both elements of a tuple before combining them into a tuple. We must evaluate the first element, partially apply the *tuple* function, evaluate the second element, and finally use the partially applied function to finish tuple construction. To allow our notation to accurately represent a tuple during construction, we must also define some notation for the partially applied tuple function, as follows.

$$(.[\![v_1]\!]\,,\, e_2) \quad = \quad [\![tuple(v_1)]\!](e_2)$$

Let us explore the normal process of constructing a tuple from two arbitrary expressions e_1 and e_2, which we assume will evaluate to some values v_1 and v_2 respectively. First e_1 is evaluated to v_1, then the function *tuple* is (partially) applied to v_1, then e_2 is evaluated to v_2, and then the partially applied $tuple(v_1)$ is applied to v_2.

This process is shown on the left below, with the same expressions shown using our notation on the right. We use the $\longrightarrow^*$ symbol to represent multiple steps being taken (see Section 3.2.8). As we use tuples in many of our examples, we will be using the notation on the right to represent tuple construction, as well as the intermediate representation of partially applied tuple constructors.

$$
\begin{array}{ll}
[\![tuple]\!](e_1)(e_2) & (e_1, e_2) \\
\longrightarrow^* \text{(FunStep, FunParStep, given)} & \longrightarrow^* \text{(FunStep, FunParStep, given)} \\
[\![tuple]\!]([\![v_1]\!])(e_2) & ([\![v_1]\!], e_2) \\
\longrightarrow \text{(FunStep, FunApp)} & \longrightarrow \text{(FunStep, FunApp)} \\
[\![tuple(v_1)]\!](e_2) & (.[\![v_1]\!], e_2) \\
\longrightarrow^* \text{(FunParStep, given)} & \longrightarrow^* \text{(FunParStep, given)} \\
[\![tuple(v_1)]\!]([\![v_2]\!]) & (.[\![v_1]\!], [\![v_2]\!]) \\
\longrightarrow \text{(FunApp)} & \longrightarrow \text{(FunApp)} \\
[\![(v_1, v_2)]\!] & [\![(v_1, v_2)]\!]
\end{array}
$$

3.2.6 Functions Returning Expressions

Consider the two following versions of the FunApp rule.

$$\frac{}{[\![v_1]\!]([\![v_2]\!]) \longrightarrow [\![v_1(v_2)]\!]} \text{(FunAppA)} \qquad \frac{}{[\![v_1]\!]([\![v_2]\!]) \longrightarrow v_1(v_2)} \text{(FunAppB)}$$

In FunAppA, we assume that the function v_1 always returns an underlying value, and that value is then put into a value expression. In FunAppB, we assume that the function v_1 can return a non-value expression, and we step immediately into that expression.

The reason we might want to use FunAppB is to allow for substitution. For example, say that we have some complex expression e_1 that will evaluate to some integer i_1. If i_1 is less than 10, we will return i_1, otherwise we will evaluate some other expensive expression e_2. Shown below are two ways to write such an expression, using each of the function application methods shown above.

$$
\begin{array}{rl}
\texttt{IF } [\![less(10)]\!](e_1) \texttt{ THEN } e_1 \texttt{ ELSE } e_2 & \text{(FunAppA)} \\
[\![\lambda x.\texttt{IF } [\![less(10)]\!](x) \texttt{ THEN } x \texttt{ ELSE } e_2]\!](e_1) & \text{(FunAppB)}
\end{array}
$$

In the FunAppA instance, e_1 may be evaluated twice (if i_1 is less than 10). In the FunAppB instance, e_1 will always be evaluated exactly once. In the interest of efficiency, we might want to choose FunAppB. However, we have opted for FunAppA, for the following three reasons.

- For FunAppB, every underlying function would need to return an expression. This complicates the use of simple functions in the underlying system and blurs the line between function application and attribution. This compromises the designed separation of attribution and underlying logic.
- If an expression is expensive, it can be moved to an attribute, which will be cached (when caching is implemented, as shown in Section 4.2). This means that it will only be evaluated once.
- If the user desires to compute values and substitute these into expressions, this can be achieved with parameterised attributes, as described in Section 4.1.

3.2.7 Option Types and the Null Node

There will be cases where it will be convenient for attributes to return a value that is an option. For example, one might want many nodes to return $some(n_p)$ for the intrinsic parent attribute, while the root node returns *none*. Assuming the existence of some set of monomorphic *option* types (or polymorphic *option* in a future version of Saiga), alongside associated *isSome* and *getOrElse* underlying functions, this is simple to implement in Saiga.

We assume the attribute parent, where $\tau(\text{parent}) = option(node)$. We can now redefine the globmin attribute as described in Section 3.1.7 using option types instead of the isRoot or a *root* node type.

$$\begin{aligned}\sigma(n, \text{globmin}) = \texttt{IF}\ & [\![isSome]\!]([\![n]\!].\text{parent}) \\ \texttt{THEN}\ & [\![getOrElse(n_{null})]\!]([\![n]\!].\text{parent}).\text{globmin} \\ \texttt{ELSE}\ & [\![n]\!].\text{locmin}\end{aligned}$$

Since Saiga does not have pattern matching features, we must rely on functions like *isSome* and *getOrElse* to manage the *some* and *none* cases of an option type. Note that *getOrElse* must always return some value, so the default value n_{null} is supplied. We know that this value will not be used in this case, assuming the function *isSome* behaves as expected. There is some redundancy here, as *getOrElse* is, in a way, guarding against the *none* case, which has already been guarded against with the conditional.

Given that we already need some null node to exist for this approach, we often use a null-check approach to allow relationships to "not exist". We can again redefine the globmin attribute, where $\tau(\text{parent}) = node$, and assuming an underlying function *isNull* which returns *true* if its argument is n_{null}.

$$\begin{aligned}\sigma(n, \text{globmin}) = \texttt{IF}\ & [\![isNull]\!]([\![n]\!].\text{parent}) \\ \texttt{THEN}\ & [\![n]\!].\text{locmin} \\ \texttt{ELSE}\ & [\![n]\!].\text{parent}.\text{globmin}\end{aligned}$$

This approach allows us to reason about a "non existent" node n_{null}, without requiring the use of *getOrElse*. The only complication here is the behaviour of an expression like $[\![n]\!].\text{parent}.\text{globmin}$ when n's parent is n_{null}. This means we need to be ready for expressions like $[\![n_{null}]\!].\text{globmin}$ to be evaluated, which is not a problem if null checks are used in the right places.

3.2.8 The Multistep Relation

Sometimes it is useful to show a relation between two expressions that represents more than one step. Here we present the multistep relation, denoted by the $\longrightarrow^*$ symbol.

$$\frac{}{e \longrightarrow^* e}\ \text{(MultiRefl)} \qquad \frac{e_1 \longrightarrow e_2 \qquad e_2 \longrightarrow^* e_3}{e_1 \longrightarrow^* e_3}\ \text{(MultiStep)}$$

This is a simple extension of the step relation which we use throughout this thesis to express evaluations consisting of more than one step. In Section 6.7 we prove that multistep

and big step (below) are equivalent.

3.2.9 The Big Step Relation

We use a small step operational semantics so we can examine the finer details of evaluation. However, there is an equivalent big step semantics that can be useful when writing proofs in Saiga. We do not modify the types, type rules, or expression language; we merely present an alternative step relation, which we prove equivalent to the multistep relation in Section 6.7. We use the double-headed arrow $\twoheadrightarrow$ to indicate a big step relation.

$$\frac{}{[\![v]\!] \twoheadrightarrow v} \text{ (BRefl)} \qquad \frac{e_1 \twoheadrightarrow \mathit{true} \qquad e_2 \twoheadrightarrow v}{\texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3 \twoheadrightarrow v} \text{ (BCondTrue)}$$

$$\frac{e_1 \twoheadrightarrow \mathit{false} \qquad e_3 \twoheadrightarrow v}{\texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3 \twoheadrightarrow v} \text{ (BCondFalse)} \qquad \frac{e_1 \twoheadrightarrow v_1 \qquad e_2 \twoheadrightarrow v_2}{e_1(e_2) \twoheadrightarrow v_1(v_2)} \text{ (BFun)}$$

$$\frac{e \twoheadrightarrow n \qquad \sigma(n,a) \twoheadrightarrow v}{e.a \twoheadrightarrow v} \text{ (BAttr)}$$

The big step relation expresses an expression evaluating to a value. Notice that on the right hand side of the $\twoheadrightarrow$ relation indicator a value is given; not an expression. A big step relation is always between an expression (on the left) and an underlying value (on the right). We have found that some proofs are more easily reasoned about using the big step semantics, as more focus is placed on the "terminals" of evaluation, and less on the intermediary stages.

3.3 Name Analysis Example

To demonstrate Saiga's use on a larger problem, let us consider name analysis on an expression language that allows for addition and let bindings. The grammar for such a language is shown below.

```
E  ::= 'let' varName '=' E 'in' E
   ::= E '+' F
   ::= F
F  ::= Integer
   ::= VarName
```

The abstract grammar for such a language might look like the following. We write this to formalise what node types are expected, and what intrinsic and structural attributes must be defined.

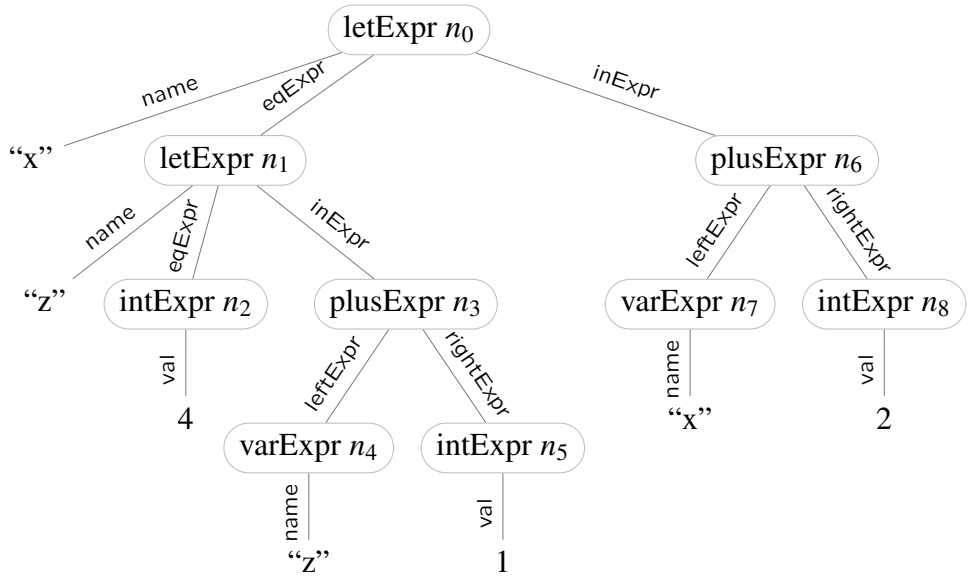
```
letExpr ::= name(string), eqExpr(node), inExpr(node)
plusExpr ::= leftExpr(node), rightExpr(node)
intExpr ::= val(integer)
varExpr ::= name(string)
```

The above notation indicates that the attribute nodeType has four possible values: *letExpr*, *plusExpr*, *intExpr*, and *varExpr*. *letExpr* nodes can be expected to have an intrinsic attribute name of type String, and so on.

A legal expression in such a language might look like the following.

```
1  let x = (let z = 4 in z + 1) in x + 2
```

We can represent this expression in the following tree, which contains annotations for each relevant intrinsic attribute of each node. Nodes are outlined, with their nodeType value annotated at their root, for ease of reading. Edges represent intrinsic attributes, and are labeled with their attribute name. The parent attribute is not shown here, but is assumed with sensible values representing the tree below. As n_0 has no parent, its parent attribute returns n_{null}.



Note that not every attribute has a value shown for every node. This is because, for example, the val attribute is not relevant to a *plusExpr* node, as a plus node does not have an integer value attached to it at parsing time in this example. Since the context function is a mathematical function, there will be an attribute expression available for val even on *plusExpr* nodes, but this value will be some default value we define when defining the context function. Since val is an intrinsic integer attribute, a sensible default might be a value expression containing zero ⟦0⟧.

As part of performing semantic analysis on such a tree, we need to be able to match each use of a variable name (in this case "x" and "z") to their defining uses. We will define the attribute env to generate an "environment", which is an ordered list of tuples that bind the name of a variable to their defining occurrences. To achieve this, we define the attribute env, which represents the scope of available names at each node. We define env as follows.

$$\sigma(n,\mathsf{env}) = \begin{cases} ⟦[]⟧ & n = n_{null} \\ ⟦\mathit{prepend}⟧ & (\text{letExpr}).\mathsf{inExpr} \\ \quad((⟦n⟧.\mathsf{parent}, ⟦n⟧.\mathsf{parent.name})) & \\ \quad(⟦n⟧.\mathsf{parent.env}) & \\ \text{inherited} & \end{cases}$$

The underlying function *prepend* appends a node/string tuple to a list of such tuples. The function *get* searches such a list and returns the node in the first tuple whose string matches the given string. *get* will return n_{null} when no matching entry is found. We then define the attribute decl to link a *varExpr* node to its defining occurrence, using the env attribute.

$$\sigma(n, \mathsf{decl}) = [\![get]\!]([\![n]\!].\mathsf{name})([\![n]\!].\mathsf{env})$$

To test this example we will evaluate the expression $[\![n_4]\!].\mathsf{decl}$ on the context described by the tree above. Since n_4 is a use of the name "z", we expect the defining occurrence n_1 to be reached.

$[\![n_4]\!].\mathsf{decl}$

(1) $\longrightarrow$ (AttrFetch)

$[\![get]\!]([\![n_4]\!].\mathsf{name})([\![n_4]\!].\mathsf{env})$

(2) $\longrightarrow$ (FunStep, FunParStep)

$[\![get]\!]([\![\text{"z"}]\!])([\![n_4]\!].\mathsf{env})$

(3) $\longrightarrow$ (FunStep, FunApp)

$[\![get(\text{"z"})]\!]([\![n_4]\!].\mathsf{env})$

(4) $\longrightarrow$ (FunParStep, AttrFetch, using the inherited case)

$[\![get(\text{"z"})]\!]([\![n_3]\!].\mathsf{env})$

(5) $\longrightarrow$ (FunParStep, AttrFetch)

$[\![get(\text{"z"})]\!]([\![prepend]\!]([\![n_3]\!].\mathsf{parent}, [\![n_3]\!].\mathsf{parent.name}))([\![n_3]\!].\mathsf{parent.env}))$

(6) $\longrightarrow$ (FunParStep, FunStep, FunParStep, FunStep, AttrFetch)

$[\![get(\text{"z"})]\!]([\![prepend]\!]([\![n_1]\!], [\![n_3]\!].\mathsf{parent.name}))([\![n_3]\!].\mathsf{parent.env}))$

(7) $\longrightarrow$ (FunParStep, FunStep, FunParStep, FunParStep, FunStep, FunApp, partial tuple construction)

$[\![get(\text{"z"})]\!]([\![prepend]\!](.[\![n_1]\!], [\![n_3]\!].\mathsf{parent.name}))([\![n_3]\!].\mathsf{parent.env}))$

(8) $\longrightarrow$ (FunParStep, FunStep, FunParStep, FunParStep, AttrNodeStep, AttrFetch)

$[\![get(\text{"z"})]\!]([\![prepend]\!]([\![n_1]\!], [\![n_1]\!].\mathsf{name}))([\![n_3]\!].\mathsf{parent.env}))$

(9) $\longrightarrow$ (FunParStep, FunStep, FunParStep, FunParStep, AttrFetch)

$[\![get(\text{"z"})]\!]([\![prepend]\!]([\![n_1]\!], [\![\text{"z"}]\!]))([\![n_3]\!].\mathsf{parent.env}))$

(10) $\longrightarrow$ (FunParStep, FunStep, FunParStep, FunParStep, FunApp, finishing tuple construction)

$[\![get(\text{"z"})]\!]([\![prepend]\!]([\![(n_1, \text{"z"})]\!])([\![n_3]\!].\mathsf{parent.env}))$

(11) $\longrightarrow$ (FunParStep, FunStep, FunApp)

$[\![get(\text{"z"})]\!]([\![prepend((n_1, \text{"z"}))]\!]([\![n_3]\!].\mathsf{parent.env}))$

(12) $\longrightarrow$ (FunParStep, FunParStep, AttrNodeStep, AttrFetch)

$[\![get(\text{"z"})]\!]([\![prepend((n_1, \text{"z"}))]\!]([\![n_1]\!].\mathsf{env}))$

(13) $\longrightarrow$ (FunParStep, FunParStep, AttrFetch, inherited case)

$$[\![get(\text{“z”})]\!]([\![prepend((n_1, \text{“z”}))]\!]([\![n_0]\!].\mathsf{env}))$$

(14) $\longrightarrow$ (FunParStep, FunParStep, AttrFetch, inherited case)

$$[\![get(\text{“z”})]\!]([\![prepend((n_1, \text{“z”}))]\!]([\![n_{null}]\!].\mathsf{env}))$$

(15) $\longrightarrow$ (FunParStep, FunParStep, AttrFetch, null case)

$$[\![get(\text{“z”})]\!]([\![prepend((n_1, \text{“z”}))]\!]([\![[]]\!]))$$

(16) $\longrightarrow$ (FunParStep, FunApp)

$$[\![get(\text{“z”})]\!]([\![[(n_1, \text{“z”})]]\!])$$

(17) $\longrightarrow$ (FunApp)

$$[\![n_1]\!]$$

This example has shown Saiga's ability to model and evaluate real-world attribute grammar programs.

3.4 Conclusion

We have presented the core Saiga calculus, including the underlying type system, the Saiga expression grammar, type rules, and semantic rules, in Section 3.1. We discussed some of the finer details of this calculus and presented the multistep and big step semantics in Section 3.2. We showed how core Saiga can be used to express and evaluate name analysis on a simple expression language in Section 3.3.

We believe we have achieved our goal of *simplicity*, with an expression language made up of only four production rules, each with one type rule, and eight semantic rules in total. The underlying system being assumed allowed us to forego implementing our own detailed type system for values and function application semantics, further simplifying our approach and allowing us to focus on only *domain specific* tasks; defining and evaluating attributes. The entirety of the equation selection process, along with our method of accessing tree relationships and intrinsic tree data are relegated to the context function, allowing for simple and *generalised* expression of a variety of attribute grammar notations.

In Chapter 4 we present the extended Saiga calculus, which includes parameterised attributes and attribute caching. In Chapter 5 we present the higher order Saiga calculus, allowing decoratable nodes to be created during evaluation. These chapters will take a similar shape to this chapter, with full semantics shown first, followed by a discussion of the details of the system, and an example for each new feature presented.

The beer I had for breakfast wasn't bad, so I had one more for dessert.
Sunday Morning Coming Down – Kris Kristofferson

4
Saiga Extended

In Chapter 3 we presented a core calculus for attribute grammar evaluation. However, there is a set of common attribute extensions, as explored in Section 2.3, of which every modern attribute grammar platform implements a subset, and we want to model the semantics of some of these extensions. Attribute caching (Section 2.3.3) is one of the most fundamental extensions, allowing the computed values of attributes to be stored for later use. Parameterised attributes (Section 2.3.2) extend the semantics of attribute evaluation to allow parameters to be passed, expanding an attribute's output from a single value to a map of input to output values. We choose to implement these two extensions in this chapter as they are commonly implemented in modern platforms, integrate smoothly into our calculus, and are *pure* in the sense that they are focused on the semantics of attribute evaluation rather than equation selection or notation.

In Section 4.1 we present parameterised attributes, and in Section 4.2 we present attribute caching. For each feature we will present the new expression language forms, type rules, and operational semantics, along with a brief discussion and an example. In Section 4.3 we discuss the reason behind some of the design choices made in developing these extensions. We consider the full semantics shown in Chapter 3 to be "core Saiga", and the extended semantics shown in this chapter to be "extended Saiga".

4.1 Parameterised Attributes

Parameterisation extends attribution mechanics to accept parameters when calling attributes (see Section 2.3.2). Instead of each node having a single value available for each attribute, some attributes represent a map from parameter values to output values. For example, one could define an attribute `getDecl` that takes a string parameter and searches the tree for the nearest declaration of that name.

4.1.1 Expression Language

To allow for parameterised attributes we extend the attribution production of Saiga's expression grammar as follows.

e	::=	$[\![v]\!]$	value
	|	`IF` e_1 `THEN` e_2 `ELSE` e_3	conditional
	|	$e_1(e_2)$	function application
	|	$e_1.a(e_2)$	attribution

Attribution expressions now contain a second subexpression, underlined in the grammar above. In the attribution production, e_1 is an expression that will evaluate to the node to be evaluated upon, and e_2 is an expression that will evaluate to the parameter for such an attribute evaluation. The semantics for this new expression form are shown in Section 4.1.4.

4.1.2 All Types

To account for parameterised attributes we adjust the type signature of a context function, and add the new type function ρ, as shown below.

Type	$t \in \boldsymbol{T}$
Value	$v \in t$
Expression	$e \in \boldsymbol{E}$
Attribute	$a \in \boldsymbol{A}$
Node	$n \in \boldsymbol{N}$
Attribute Type	$\tau \in \boldsymbol{A} \to \boldsymbol{T}$
Parameter Type	$\rho \in \boldsymbol{A} \to \boldsymbol{T}$
Context	$\sigma \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}$

Just as τ produces the expected output type of an attribute, the new ρ produces the expected parameter type for an attribute. The context function is redefined to have three parameters: a node, an attribute, and a parameter. The type of the parameter is defined by ρ and the input attribute, so the context function is now dependently typed. The context function now requires a parameter to be given for every attribute definition. See Section 4.3.1 for a discussion of non-parameterised attributes under this model.

4.1.3 Type Rules

$$\frac{v : t}{[\![v]\!] : t}\ \text{(TypeVal)} \qquad \frac{e_1 : \text{boolean} \qquad e_2 : t \qquad e_3 : t}{\texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 : t}\ \text{(TypeCond)}$$

$$\frac{e_1 : t_1 \to t_2 \qquad e_2 : t_1}{e_1(e_2) : t_2}\ \text{(TypeFun)} \qquad \frac{e_1 : \boldsymbol{N} \qquad e_2 : \rho(a)}{e_1.a(e_2) : \tau(a)}\ \text{(TypeAttr)}$$

Only a simple adjustment to Saiga's type rules is required to account for parameterised attributes. We add a second condition to the TypeAttr rule to ensure that the second subexpression of an attribution expression has a type that matches the appropriate attribute, using the type function ρ.

4.1.4 Semantic Rules

$$\frac{e_1 \longrightarrow e_1'}{\texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3 \longrightarrow \texttt{IF } e_1' \texttt{ THEN } e_2 \texttt{ ELSE } e_3} \text{ (CondStep)}$$

$$\frac{}{\texttt{IF } [\![\mathit{true}]\!] \texttt{ THEN } e_2 \texttt{ ELSE } e_3 \longrightarrow e_2} \text{ (CondTrue)} \qquad \frac{e_1 \longrightarrow e_1'}{e_1(e_2) \longrightarrow e_1'(e_2)} \text{ (FunStep)}$$

$$\frac{}{\texttt{IF } [\![\mathit{false}]\!] \texttt{ THEN } e_2 \texttt{ ELSE } e_3 \longrightarrow e_3} \text{ (CondFalse)} \qquad \frac{e \longrightarrow e'}{[\![v]\!](e) \longrightarrow [\![v]\!](e')} \text{ (FunParStep)}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1.a(e_2) \longrightarrow e_1'.a(e_2)} \text{ (AttrNodeStep)} \qquad \frac{e \longrightarrow e'}{[\![n]\!].a(e) \longrightarrow [\![n]\!].a(e')} \text{ (AttrParStep)}$$

$$\frac{}{[\![v_1]\!]([\![v_2]\!]) \longrightarrow [\![v_1(v_2)]\!]} \text{ (FunApp)} \qquad \frac{}{[\![n]\!].a([\![p]\!]) \longrightarrow \sigma(n,a,p)} \text{ (AttrFetch)}$$

We extend Saiga's operational semantics with the new AttrParStep rule, and we adjust the AttrNodeStep and AttrFetch rules. The new rule and changes to the existing rules are underlined in these semantics. The adjusted AttrNodeStep is functionally unchanged; it is only modified to allow for the new shape of an attribution expression. The new AttrParStep rule allows an attribution expression's second subexpression to take steps once the first subexpression has reached a value. The effect of these changes is that an attribution expression will first evaluate its node subexpression to a value, then its parameter subexpression. This is similar semantics to the function application expression.

Once both the node and parameter subexpressions have reached values, AttrFetch performs a similar task as before, stepping into whichever expression is returned by the context function. Note that the AttrFetch rule can only proceed if n is a node and p's type is $\rho(a)$, otherwise the context function has no defined output. These changes are all that is required to add parameterised attributes to Saiga. See Section 4.3.2 for a discussion of how these semantics differ from non-parameterised attributes that return functions.

4.1.5 Name Analysis Example Revisited

In Section 3.3, we performed name analysis by constructing a full environment (the env attribute), and searching that environment for a variable use's declaration. The environment is a list of all names in scope, along with a reference to their declarations. In this example, we begin with the same tree structure presented in Section 3.3, but we implement name analysis using a different approach. Our new approach is to start with a name, and search the tree specifically for that name's declaration. Such a task is made possible with the use of parameterised attributes.

In this example we use a number of shorthands for 'non-parameterised attributes' which are discussed in Section 4.3.1. When an attribute expression is written without a parameter included, this is shorthand for including a parameter subexpression holding the unit value, which we use to represent non-parameterised attributes.

We define the parameterised attribute getDecl, where $\rho(\text{getDecl}) = \textit{string}$, below.

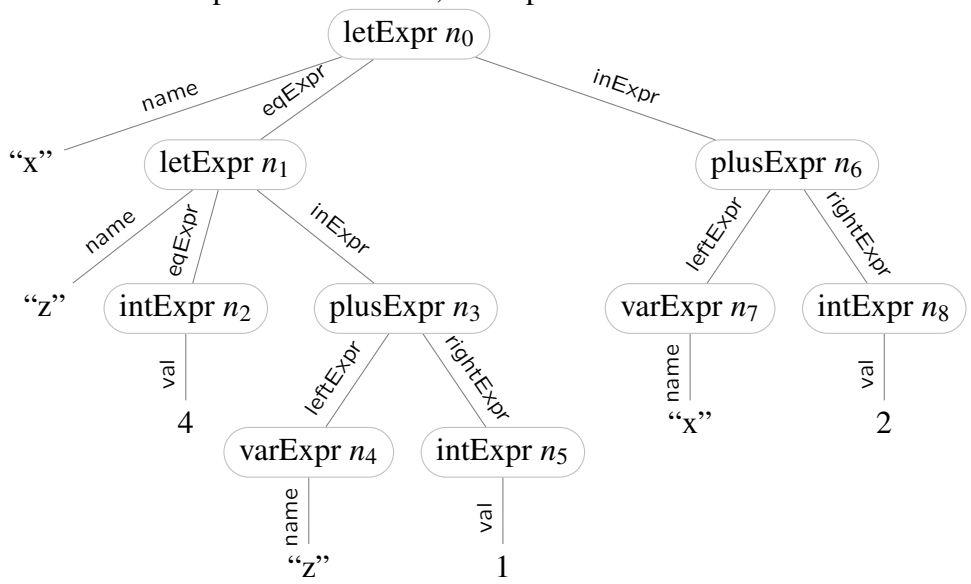
$$\sigma(n, \text{getDecl}, s) = \begin{cases} [\![n_{null}]\!] & n = n_{null} \\ \texttt{IF } [\![eq(s)]\!]([\![n]\!].\text{parent.name}) & (\text{letExpr}).\text{inExpr} \\ \texttt{ THEN } [\![n]\!].\text{parent} & \\ \texttt{ ELSE } [\![n]\!].\text{parent.getDecl}([\![s]\!]) & \\ \text{inherited} & \end{cases}$$

The getDecl attribute traverses upwards in the tree, checking against any *letExpr* nodes, specifically when traversing from such a node's inExpr child. If the discovered *letExpr* node is the declaration of the name given by the parameter s, this node is returned. We can now define the simple attribute decl, which finds the declaration of any *varExpr* node using the getDecl attribute.

$$\sigma(n, \text{decl}, \_) = \begin{cases} [\![n]\!].\text{getDecl}([\![n]\!].\text{name}) & (\text{varExpr}) \\ [\![n_{null}]\!] & \text{otherwise} \end{cases}$$

The decl attribute, like the intrinsic attributes parent and name, ignores its parameter. Non-parameterised attributes are discussed further in Section 4.3.1.

To test out this attribute grammar program, we will evaluate n_4.decl. The tree we are evaluating upon is shown again below. As the attributes in this example are designed to work with the same kind of tree as in Section 3.3, we will evaluate on the same tree used in that example. As in the example in Section 3.3, we expect the evaluation to reach the value $[\![n_1]\!]$.



We proceed by evaluating $[\![n_4]\!]$.decl, using the new definitions of decl and getDecl.

$$\begin{aligned} & [\![n_4]\!].\text{decl} \\ (1) \longrightarrow & \text{ (AttrFetch)} \\ & [\![n_4]\!].\text{getDecl}([\![n_4]\!].\text{name}) \\ (2) \longrightarrow & \text{ (AttrParStep, AttrFetch)} \\ & [\![n_4]\!].\text{getDecl}([\![\text{"z"}]\!]) \end{aligned}$$

(3) ⟶ (AttrFetch, by inheritance)
$[\![n_3]\!]$.getDecl($[\![$"z"$]\!]$)

(4) ⟶ (AttrFetch)
IF $[\![eq(\text{"z"})]\!]$($[\![n_3]\!]$.parent.name) THEN $[\![n_3]\!]$.parent
ELSE $[\![n_3]\!]$.parent.getDecl($[\![$"z"$]\!]$)

(5) ⟶ (CondStep, FunParStep, AttrNodeStep, AttrFetch)
IF $[\![eq(\text{"z"})]\!]$($[\![n_1]\!]$.name) THEN $[\![n_3]\!]$.parent
ELSE $[\![n_3]\!]$.parent.getDecl($[\![$"z"$]\!]$)

(6) ⟶ (CondStep, FunParStep, AttrFetch)
IF $[\![eq(\text{"z"})]\!]$($[\![\text{"z"}]\!]$) THEN $[\![n_3]\!]$.parent
ELSE $[\![n_3]\!]$.parent.getDecl($[\![$"z"$]\!]$)

(7) ⟶ (CondStep, FunApp)
IF $[\![true]\!]$ THEN $[\![n_3]\!]$.parent
ELSE $[\![n_3]\!]$.parent.getDecl($[\![$"z"$]\!]$)

(8) ⟶ (CondTrue)
$[\![n_3]\!]$.parent

(9) ⟶ (AttrFetch)
$[\![n_1]\!]$

As expected, evaluation terminated with the node value n_1. Implementing parameterisation in Saiga has been a very simple exercise, demonstrating the flexibility of the Saiga model. Further discussion of the implementation of parameterisation and its strengths and caveats can be found in Section 4.3, but first we will discuss attribute caching in Saiga.

4.2 Attribute Caching

Attribute caching (or memoisation) involves storing the computed value of an attribute after evaluation so that it does not need to be recomputed if requested again (see Section 2.3.3). For example, after evaluating $[\![n_4]\!]$.decl as in Section 4.1.5, we reach the value $[\![n_1]\!]$. If there is cause to again access $[\![n_4]\!]$.decl, it would be desirable for the computed value $[\![n_1]\!]$ to be returned immediately, rather than having this value computed again.

To implement caching, we need some way to record the computed value of an attribute, and a way for evaluation to retrieve this value. Instead of implementing another storage method for cached values, we opt to allow the context function to transform during evaluation. When an attribute a on a node n with a parameter p is evaluated to some value v, we modify σ such that $\sigma(n,a,p)$ will now return $[\![v]\!]$. We use the $\oplus$ operator to indicate a context function has been modified with an overridden output value.

$$\sigma \oplus \{(n,a,p) \mapsto [\![v]\!]\}$$

To implement caching behaviour, we add an expression form, appropriate type rules, and modify and extend our operational semantics.

4.2.1 Expression Language

We extend the expression grammar as follows.

e	::=	$[\![v]\!]$	value
	|	`IF` e_1 `THEN` e_2 `ELSE` e_3	conditional
	|	$e_1(e_2)$	function application
	|	$e_1.a(e_2)$	attribution
	|	$n.a(p) := e_1$	cache

One production has been added to the Saiga expression grammar: the cache expression. The new expression form has been highlighted above. The purpose of this expression is to keep track of which attributes are being evaluated, so that their results can be written to the cache when a value is reached. We assume, in the construction of a cache expression, that $n \in \boldsymbol{N}$, $a \in \boldsymbol{A}$, and $p \in \rho(a)$.

$$n.a(p) := e_1 \quad \text{vs} \quad [\![n]\!].a([\![p]\!])$$

Note that the syntax of a cache expression is similar to the syntax of an attribution expression. The caching expression's syntax is designed to represent that a particular attribute is being calculated, and will be written to cache once its evaluation is complete. A cache expression only has a single subexpression (e_1 in the case above); the values n, a, and p are components of a cache expression, but are not expressions themselves (and therefore their values are not wrapped as in $[\![n]\!]$ vs n).

4.2.2 Type Rules

$$\frac{v : t}{[\![v]\!] : t}\ \text{(TypeVal)} \qquad \frac{e_1 : \text{boolean} \qquad e_2 : t \qquad e_3 : t}{\texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 : t}\ \text{(TypeCond)}$$

$$\frac{e_1 : t_1 \to t_2 \qquad e_2 : t_1}{e_1(e_2) : t_2}\ \text{(TypeFun)} \qquad \frac{e_1 : \boldsymbol{N} \qquad e_2 : \rho(a)}{e_1.a(e_2) : \tau(a)}\ \text{(TypeAttr)}$$

$$\frac{e : \tau(a)}{n.a(p) := e : \tau(a)}\ \text{(TypeCache)}$$

With the addition of the new expression form, we provide a new typing rule TypeCache, which simply ensures that the single subexpression of a cache expression matches the type associated with the given attribute. The new type rule is highlighted above.

4.2.3 Semantic Rules

The changes to Saiga's semantic rules to allow caching are not as simple as they were for parameterised attributes. Below we present the new semantic rules, which include a transforming context function, as well as four new semantic rules.

$$\frac{\sigma \vdash e_1 \longrightarrow \sigma' \vdash e_1'}{\sigma \vdash \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \longrightarrow \sigma' \vdash \texttt{IF}\ e_1'\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3}\ \text{(CondStep)}$$

$$\frac{}{\sigma \vdash \texttt{IF}\ [\![true]\!]\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \longrightarrow \sigma \vdash e_2}\ \text{(CondTrue)}$$

$$\frac{}{\sigma \vdash \texttt{IF}\ [\![false]\!]\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \longrightarrow \sigma \vdash e_3}\ \text{(CondFalse)}$$

$$\frac{\sigma \vdash e_1 \longrightarrow \sigma' \vdash e_1'}{\sigma \vdash e_1(e_2) \longrightarrow \sigma' \vdash e_1'(e_2)}\ \text{(FunStep)} \qquad \frac{\sigma \vdash e \longrightarrow \sigma' \vdash e'}{\sigma \vdash [\![v]\!](e) \longrightarrow \sigma' \vdash [\![v]\!](e')}\ \text{(FunParStep)}$$

$$\frac{}{\sigma \vdash [\![v_1]\!]([\![v_2]\!]) \longrightarrow \sigma \vdash [\![v_1(v_2)]\!]}\ \text{(FunApp)}$$

$$\frac{\sigma \vdash e_1 \longrightarrow \sigma' \vdash e_1'}{\sigma \vdash e_1.a(e_2) \longrightarrow \sigma' \vdash e_1'.a(e_2)}\ \text{(AttrNodeStep)}$$

$$\frac{\sigma \vdash e \longrightarrow \sigma' \vdash e'}{\sigma \vdash [\![n]\!].a(e) \longrightarrow \sigma' \vdash [\![n]\!].a(e')}\ \text{(AttrParStep)}$$

$$\frac{\sigma(n,a,p) = [\![v]\!]}{\sigma \vdash [\![n]\!].a([\![p]\!]) \longrightarrow \sigma \vdash [\![v]\!]}\ \text{(AttrFetchValue)}$$

$$\frac{\sigma(n,a,p) \text{ is not a value}}{\sigma \vdash [\![n]\!].a([\![p]\!]) \longrightarrow \sigma \vdash n.a(p) := \sigma(n,a,p)}\ \text{(AttrFetchCached)}$$

$$\frac{\sigma \vdash e \longrightarrow \sigma' \vdash e'}{\sigma \vdash n.a(p) := e \longrightarrow \sigma' \vdash n.a(p) := e'}\ \text{(CacheStep)}$$

$$\frac{}{\sigma \vdash n.a(p) := [\![v]\!] \longrightarrow \sigma \oplus \{(n,a,p) \mapsto [\![v]\!]\} \vdash [\![v]\!]}\ \text{(CacheWrite)}$$

A Changing Context Function

To implement caching, we add two terms to the step relation. Previously, we would say that some expression e steps to some expression e' under some context function σ. This would be written as follows, with σ considered a global variable that does not need to be written alongside the step relation.

$$e \longrightarrow e'$$

Now that it is possible for the context function to transform during evaluation, the context function is not only an input, but also an output of the step relation. Therefore, the step relation now steps the pair σ and e to the pair σ' and e', as in the new notation.

$$\sigma \vdash e \longrightarrow \sigma' \vdash e'$$

In the above semantics, we have added context stepping notations (highlighted) to all of the steps for all conditional rules, function application rules, and the AttrNodeStep and

AttrParStep rules, which remain otherwise unchanged.

Caching Rules

$$\frac{\sigma \vdash e \longrightarrow \sigma' \vdash e'}{\sigma \vdash n.a(p) := e \longrightarrow \sigma' \vdash n.a(p) := e'} \text{ (CacheStep)}$$

$$\frac{}{\sigma \vdash n.a(p) := [\![v]\!] \longrightarrow \sigma \oplus \{(n,a,p) \mapsto [\![v]\!]\} \vdash [\![v]\!]} \text{ (CacheWrite)}$$

We have added the two semantic rules CacheStep and CacheWrite. The CacheStep rule dictates that if a cache expression contains a subexpression that can make some step, then the cache expression carries that step, adjusting its subexpression and context function in the process. The CacheWrite rule says that when a cache expression holds a value subexpression, the context function is updated and the cache wrapper is removed, yielding the contained value expression. At this point CacheWrite is the only step rule that can change a context function, but any step that requires a substep could contain a CacheWrite step, so we account for a context transformation on any semantic rule with a step sub-relation. In Chapter 5 we introduce some more rules that modify the context function, to implement higher order attributes.

Two Attribution Rules

$$\frac{\sigma(n,a,p) = [\![v]\!]}{\sigma \vdash [\![n]\!].a([\![p]\!]) \longrightarrow \sigma \vdash [\![v]\!]} \text{ (AttrFetchValue)}$$

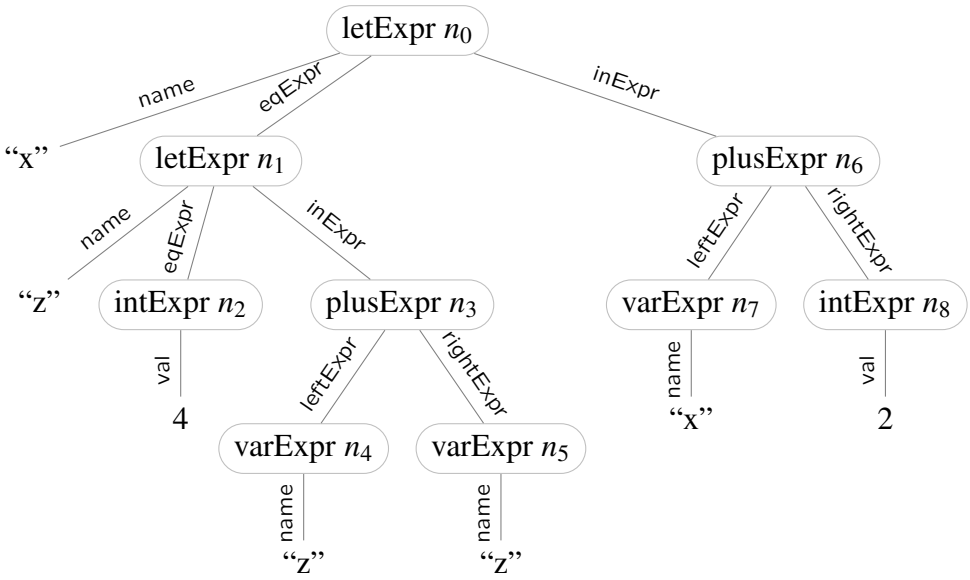
$$\frac{\sigma(n,a,p) \text{ is not a value}}{\sigma \vdash [\![n]\!].a([\![p]\!]) \longrightarrow \sigma \vdash n.a(p) := \sigma(n,a,p)} \text{ (AttrFetchCached)}$$

The cache expression is designed to be produced by attribution expressions. The pattern of attribute evaluation is now: evaluate the node; evaluate the parameter; wrap the context function results in a cache expression; evaluate the cache expression; write the result to cache; return the result. However, if the context function returns a value, this caching operation is redundant; we would be reading some value v_1 from the context, and then writing it back to the context in the same place, which makes no change at all. Therefore, we split attribution into two rules; we wrap the result in a cache expression when the context function returns a non-value expression, and we return the result directly when a value is returned. This strategy is discussed further in Section 4.3.3.

4.2.4 Caching Example

To demonstrate the mechanics of caching, we will evaluate the `decl` attribute from Section 4.1.5 on a similar tree. In this tree, the variable name "z" appears twice in the subtree containing n_4 and n_5. This tree represents the following expression.

```
1  let x = (let z = 4 in z + z) in x + 2
```



We will begin with a context function that contains a description of the tree above, as well as the definitions of decl and getDecl from Section 4.1.5, which we will call σ_1. We will first evaluate $[\![n_4]\!]$.decl under σ_1, which produces the value v_1 and "output" context function σ_4. We will then evaluate $[\![n_5]\!]$.decl under σ_4.

For simplicity of notation, we will not show the details of context transformations during evaluation, but will instead show the numbered context functions σ_1 through σ_4, showing a full definitions of these context functions below the evaluation steps. The evaluation of $[\![n_4]\!]$.decl will be very similar to that shown in Section 4.1.5, but computed attributes will be cached along the way.

$\sigma_1 \vdash [\![n_4]\!]$.decl

(1) $\longrightarrow$ (AttrFetchCached)

$\sigma_1 \vdash n_4$.decl := $[\![n_4]\!]$.getDecl($[\![n_4]\!]$.name)

(2) $\longrightarrow$ (CacheStep, AttrParStep, AttrFetchValue)

$\sigma_1 \vdash n_4$.decl := $[\![n_4]\!]$.getDecl($[\![$"z"$]\!]$)

(3) $\longrightarrow$ (CacheStep, AttrFetchCached)

$\sigma_1 \vdash n_4$.decl := n_4.getDecl("z") := $[\![n_3]\!]$.getDecl($[\![$"z"$]\!]$)

(4) $\longrightarrow$ (CacheStep, CacheStep, AttrFetchCached)

$\sigma_1 \vdash n_4$.decl := n_4.getDecl("z") := n_3.getDecl("z") :=
IF $[\![eq$("z")$]\!]$($[\![n_3]\!]$.parent.name) THEN $[\![n_3]\!]$.parent
ELSE $[\![n_3]\!]$.parent.getDecl($[\![$"z"$]\!]$)

(5) $\longrightarrow$ (CacheStep, CacheStep, CacheStep, CondStep, FunParStep, AttrNodeStep, AttrFetchValue)

$\sigma_1 \vdash n_4$.decl := n_4.getDecl("z") := n_3.getDecl("z") :=
IF $[\![eq$("z")$]\!]$($[\![n_1]\!]$.name) THEN $[\![n_3]\!]$.parent
ELSE $[\![n_3]\!]$.parent.getDecl($[\![$"z"$]\!]$)

(6) $\longrightarrow$ (CacheStep, CacheStep, CacheStep, CondStep, FunParStep, AttrFetchValue)

$\sigma_1 \vdash n_4$.decl := n_4.getDecl("z") := n_3.getDecl("z") :=

IF $[\![eq(\text{“z”})]\!]([\![\text{“z”}]\!])$ THEN $[\![n_3]\!]$.parent
ELSE $[\![n_3]\!]$.parent.getDecl($[\![\text{“z”}]\!]$)

(7) $\longrightarrow$ (CacheStep, CacheStep, CacheStep, CondStep, FunApp)

$\sigma_1 \vdash n_4$.decl:= n_4.getDecl(“z”):= n_3.getDecl(“z”):=
IF $[\![true]\!]$ THEN $[\![n_3]\!]$.parent
ELSE $[\![n_3]\!]$.parent.getDecl($[\![\text{“z”}]\!]$)

(8) $\longrightarrow$ (CacheStep, CacheStep, CacheStep, CondTrue)

$\sigma_1 \vdash n_4$.decl:= n_4.getDecl(“z”):= n_3.getDecl(“z”):= $[\![n_3]\!]$.parent

(9) $\longrightarrow$ (CacheStep, CacheStep, CacheStep, AttrFetchValue)

$\sigma_1 \vdash n_4$.decl:= n_4.getDecl(“z”):= n_3.getDecl(“z”):= $[\![n_1]\!]$

(10) $\longrightarrow$ (CacheStep, CacheStep, CacheWrite)

$\sigma_2 \vdash n_4$.decl:= n_4.getDecl(“z”):= $[\![n_1]\!]$

(11) $\longrightarrow$ (CacheStep, CacheWrite)

$\sigma_3 \vdash n_4$.decl:= $[\![n_1]\!]$

(12) $\longrightarrow$ (CacheStep, CacheWrite)

$\sigma_4 \vdash [\![n_1]\!]$

Here evaluation has followed the same process as in Section 4.1.5, but with three extrinsic attributes being cached along the way, in steps (10), (11), and (12). The definitions of the context functions σ_1 to σ_4 are as follows. Note that in σ_4, decl is cached with the special attribute parameter (), which is the unit value (see Section 4.3.1).

$$\sigma_2 = \sigma_1 \oplus \{(n_3, \text{getDecl}, \text{“z”}) \mapsto [\![n_1]\!]\}$$
$$\sigma_3 = \sigma_2 \oplus \{(n_4, \text{getDecl}, \text{“z”}) \mapsto [\![n_1]\!]\}$$
$$\sigma_4 = \sigma_3 \oplus \{(n_4, \text{decl}, ()) \mapsto [\![n_1]\!]\}$$

The result of evaluating $[\![n_4]\!]$.decl under σ_1 is the value n_1 and the updated context function σ_4. We will now evaluate $[\![n_5]\!]$.decl under σ_4 to make use of the values cached during the first evaluation.

$\sigma_4 \vdash [\![n_5]\!]$.decl

(1) $\longrightarrow$ (AttrFetchCached)

$\sigma_4 \vdash n_5$.decl:= $[\![n_5]\!]$.getDecl($[\![n_5]\!]$.name)

(2) $\longrightarrow$ (CacheStep, AttrParStep, AttrFetchValue)

$\sigma_4 \vdash n_5$.decl:= $[\![n_5]\!]$.getDecl($[\![\text{“z”}]\!]$)

(3) $\longrightarrow$ (CacheStep, AttrFetchCached)

$\sigma_4 \vdash n_5$.decl:= n_5.getDecl(“z”):= $[\![n_3]\!]$.getDecl($[\![\text{“z”}]\!]$)

(4) $\longrightarrow$ (CacheStep, CacheStep, AttrFetchValue, as this attribute is cached)

$$\sigma_4 \vdash n_5.\mathsf{decl} := n_5.\mathsf{getDecl}(\text{“z”}) := \overline{[\![n_1]\!]}$$
$$(5) \longrightarrow \text{(CacheStep, CacheWrite)}$$
$$\sigma_5 \vdash n_5.\mathsf{decl} := \overline{[\![n_1]\!]}$$
$$(6) \longrightarrow \text{(CacheWrite)}$$
$$\sigma_6 \vdash [\![n_1]\!]$$

The first three steps proceeded in a similar fashion to $[\![n_4]\!].\mathsf{decl}$. Notice, however that step (4) uses AttrFetchValue, as the value for $[\![n_3]\!].\mathsf{getDecl}([\![\text{“z”}]\!])$ had already been cached in σ_4. The updated context functions σ_5 and σ_6 are defined as below.

$$\sigma_5 = \sigma_4 \oplus \{(n_5, \mathsf{getDecl}, \text{“z”}) \mapsto [\![n_1]\!]\}$$
$$\sigma_6 = \sigma_5 \oplus \{(n_5, \mathsf{decl}, ()) \mapsto [\![n_1]\!]\}$$

This example demonstrates that caching does not affect the output behaviour of evaluation, but adds some book-keeping along the way with cache expressions, and updates the context function every time a non-value attribute is evaluated to a value. A proof of this property, which we call *cache irrelevance*, is given in Section 6.8. While allowing a context function to transform during evaluation has added some complexity to the calculus, caching is an important part of modern attribute grammar evaluation, so it is a necessary complication. Further, higher order attributes (Chapter 5) also require a context function to transform during evaluation, so accounting for this transformation would not be avoidable, even if attribute caching were excluded.

4.3 Discussion

In this section we will discuss some of the details, design decisions, and caveats of the extended calculus presented in Sections 4.1 and 4.2.

4.3.1 Non-Parameterised Attributes

In Section 4.1 we extend regular attribution with parameters. Our solution for allowing attributes to have one parameter, multiple parameters, or no parameters at all is to force all attributes to have exactly one, and delegate tuples to do the work of multiple parameterisation. Tuples in Saiga are discussed in detail in Section 3.2.5.

We have shown examples of the use of attributes with one parameter. Attributes with multiple parameters are straightforward. Let us say we want to design some attribute `pattr` with two parameters of type *boolean* and *string*. In this case, $\rho(\mathsf{pattr}) = \mathit{TupleBoolString}$. The type *TupleBoolString* might also be represented using the notation $\mathit{boolean} \times \mathit{string}$. Evaluating such an attribute would require some expression to evaluate to a value of this kind, and it would be passed to the context function, which can treat the parts of the tuple separately if it chooses to.

Non-parameterised attributes can use the special type *unit*, which has exactly one value, which we write like the empty tuple (). The common structural attribute `parent` is likely to have no parameter. In this case we would have $\rho(\mathsf{parent}) = \mathit{unit}$.

As non-parameterised attributes are common, we use a number of shorthands to make their notation more concise. In the case where the parameter is omitted from an attribution expression, this will be considered equivalent to the full attribution expression, with a unit value expression for the parameter subexpression.

$$e_1.\texttt{attr} \quad = \quad e_1.\texttt{attr}([\![()]\!])$$

Similarly, omitting the parameter subexpression from a cache expression will also be shorthand for the unit parameter, as follows.

$$n_1.\texttt{attr:=}\ e_1 \quad = \quad n_1.\texttt{attr}(()):=\ e_1$$

When defining an attribute, we often use the underscore to indicate that we don't care what some input value is. Here we define the `attr` attribute, which has a unit parameter, which is not used.

$$\sigma(n, \texttt{attr}, \_) \quad = \quad e_1$$

We will use these notations for non-parameterised attributes frequently in examples throughout this thesis.

4.3.2 Attributes Returning Functions

As an attribute can return a function, it is possible to implement something similar to parameterised attributes using only the core Saiga semantics. If we have $\tau(\texttt{getDecl2}) = \mathit{string} \rightarrow \mathit{node}$, we can write the following expression in core Saiga.

$$\underline{\underline{([\![n_1]\!].\texttt{getDecl2})([\![\text{“a”}]\!])}}$$

Note that the above expression is written using the *core Saiga* grammar, with no parameterised attribution. The doubly-underlined expression is a regular attribute fetch, which is assumed to return a function typed expression. The singly-underlined expression is the parameter that will be used to call the result of the first expression. With this kind of format, we can allow an attribute to return an expression that will return different results depending on the parameter it is given.

However, consider the semantics of this scenario if `getDecl2` is supposed to behave similarly to `getDecl` (Section 4.1.5). The expression returned by $\sigma(n_1, \texttt{getDecl2})$ must evaluate to a function that is ready to return the declaration node for *any* input string. This means that it cannot perform a targeted search up the tree, but must prepare all possible outputs, before the parameter expression ($[\![\text{“a”}]\!]$ in this case) is even evaluated. This is not a parameterised attribute; this is a function-typed attribute.

The purpose of a parameterised attribute is for a parameter to influence how the evaluation of an attribute proceeds. This does not occur if all evaluation is completed before the parameter is "received", which is always the case for a function application expression.

Further, parameterised attributes are cached along with their parameters. With the extended calculus, if we evaluate $[\![n_1]\!].\texttt{getDecl}([\![\text{“x”}]\!])$ under σ, then the output of $\sigma(n_1, \texttt{getDecl}, \text{“x”})$ is modified; the attribute *with that particular parameter* is cached for that particular node. Returning a function as a parameter's value will cause the function itself to be cached, not any of its particular outputs.

Note that Silver does not support parameterised attributes (although it does support attributes with type parameters). Silver does support function values, however, and could implement `getDecl2` in the manner described above.

4.3.3 Re-Caching Values and Simplicity

Let us consider a version of our caching semantics (shown in Section 4.2.3) where attribution expressions always step to a cache expression, without distinguishing between value and non-value attribute expressions. In this instance, we would replace the two rules AttrFetchValue and AttrFetchCached with the following single rule.

$$\frac{}{\sigma \vdash [\![n]\!].a([\![p]\!]) \longrightarrow \sigma \vdash n.a(p) := \sigma(n,a,p)} \text{ (AttrFetch)}$$

Now let us consider a context function σ_1, with the following two specified entries.

$$\sigma_1(n_1, \texttt{attr1}, ()) = \texttt{IF } [\![\mathit{true}]\!] \texttt{ THEN } [\![4]\!] \texttt{ ELSE } [\![5]\!]$$
$$\sigma_1(n_1, \texttt{attr2}, ()) = [\![6]\!]$$

When evaluating `attr1` and `attr2` on n_1 under σ_1, it would make sense intuitively to cache the results of `attr1` but not to cache the results of `attr2`. Indeed, if we evaluate $[\![n_1]\!]$`.attr1` under the modified 'always cache' semantics here, we would see the following.

$$\begin{aligned}
&\sigma_1 \vdash [\![n_1]\!].\texttt{attr1} \\
(1) \longrightarrow\ &\text{(AttrFetch)} \\
&\sigma_1 \vdash n_1.\texttt{attr1} := \texttt{IF } [\![\mathit{true}]\!] \texttt{ THEN } [\![4]\!] \texttt{ ELSE } [\![5]\!] \\
(2) \longrightarrow\ &\text{(CacheStep, CondTrue)} \\
&\sigma_1 \vdash n_1.\texttt{attr1} := [\![4]\!] \\
(3) \longrightarrow\ &\text{(CacheStep, CondTrue)} \\
&\sigma_2 \vdash [\![4]\!] \\
&\text{where } \sigma_2 = \sigma_1 \oplus \{(n_1, \texttt{attr1}, ()) \mapsto [\![4]\!]\}
\end{aligned}$$

As expected, the output context is σ_2, which contains a cached value for `attr1`. Now, consider the steps taken to evaluate $[\![n_1]\!]$`.attr2` under σ_1.

$$\begin{aligned}
&\sigma_1 \vdash [\![n_1]\!].\texttt{attr2} \\
(1') \longrightarrow\ &\text{(AttrFetch)} \\
&\sigma_1 \vdash n_1.\texttt{attr2} := [\![6]\!] \\
(2') \longrightarrow\ &\text{(CacheWrite)} \\
&\sigma_3 \vdash [\![6]\!] \\
&\text{where } \sigma_3 = \sigma_1 \oplus \{(n_1, \texttt{attr2}, ()) \mapsto [\![6]\!]\}
\end{aligned}$$

However, we can show that $\sigma_3 = \sigma_1$. The $\oplus$ operator changes only one output from a

context function. In this instance, the inputs that have been overridden $(n_1, \texttt{attr2}, ())$ return the expression $[\![6]\!]$, which is the same as the value it is being replaced with. By functional extensionality, as all inputs return the same output, we have $\sigma_3 = \sigma_1$.

Therefore wrapping a value expression in a cache statement takes one extra step (step $(2')$ above), and results in the same output value and context. Similarly, if we were to evaluate $[\![n_1]\!].\texttt{attr1}$ again on σ_2, where it has already been cached, the newly cached value would again be cached. While the Saiga calculus is not designed with efficiency in mind, attribute caching is a feature that is usually implemented for the purpose of efficiency. Further, if attributes that had already been cached were re-cached every time they were accessed, this would not accurately mirror caching behaviour in real-world attribute grammar systems.

Consider the evaluations shown in Section 4.2.4. If we had to perform the extra caching step every time we requested the intrinsic attributes `name` and `parent`, both the number of steps shown would increase as well as the size of the expressions shown. Separately from 'step efficiency', such semantics would make analysis of these simple attribute grammar programs more verbose; evaluation would frequently produce new context functions which are identical, but are not written the same way (as in σ_1 and σ_3 above). Regularly proving the equality of redundantly updated context functions would increase the effort of evaluation analysis, and is avoided by the inclusion of the second attribution rule. For these reasons we opt to implement the two separate rules AttrFetchValue and AttrFetchCached.

4.3.4 The Multistep Relation

As the step relation has changed in the extended calculus, we must also redefine the multistep relation that was presented in Section 3.2.8. The only modification necessary is to take into account not only a changing expression but also a changing context function.

$$\frac{}{\sigma \vdash e \longrightarrow^* \sigma \vdash e} \text{ (MultiRefl)}$$

$$\frac{\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2 \qquad \sigma_2 \vdash e_2 \longrightarrow^* \sigma_3 \vdash e_3}{\sigma_1 \vdash e_1 \longrightarrow^* \sigma_3 \vdash e_3} \text{ (MultiStep)}$$

4.3.5 The Big Step Relation

We extend the big step semantics presented in Section 3.2.9 to include the full semantics of the extended Saiga calculus. As with the multistep relation, we have included a possibly transforming context function into every rule. We have also included the new rules BAttrValue, BAttrCached, and BCache, which implement the new features presented in this chapter.

$$\frac{}{\sigma \vdash [\![v]\!] \twoheadrightarrow \sigma \vdash v} \text{ (BRefl)} \qquad \frac{\sigma_1 \vdash e_1 \twoheadrightarrow \sigma_2 \vdash \mathit{true} \qquad \sigma_2 \vdash e_2 \twoheadrightarrow \sigma_3 \vdash v}{\sigma_1 \vdash \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \twoheadrightarrow \sigma_3 \vdash v} \text{ (BCondTrue)}$$

$$\frac{\sigma_1 \vdash e_1 \twoheadrightarrow \sigma_2 \vdash \mathit{false} \qquad \sigma_2 \vdash e_3 \twoheadrightarrow \sigma_3 \vdash v}{\sigma_1 \vdash \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \twoheadrightarrow \sigma_3 \vdash v} \text{ (BCondFalse)}$$

$$\frac{\sigma_1 \vdash e_1 \twoheadrightarrow \sigma_2 \vdash v_1 \qquad \sigma_2 \vdash e_2 \twoheadrightarrow \sigma_3 \vdash v_2}{\sigma_1 \vdash e_1(e_2) \twoheadrightarrow \sigma_3 \vdash v_1(v_2)} \text{ (BFun)}$$

$$\frac{\sigma_1 \vdash e_1 \twoheadrightarrow \sigma_2 \vdash n \qquad \sigma_2 \vdash e_2 \twoheadrightarrow \sigma_3 \vdash v_1 \qquad \sigma_3(n,a,v_1) = [\![v_2]\!]}{\sigma_1 \vdash e_1.a(e_2) \twoheadrightarrow \sigma_3 \vdash v_2} \text{ (BAttrValue)}$$

$$\frac{\begin{array}{c}\sigma_1 \vdash e_1 \twoheadrightarrow \sigma_2 \vdash n \qquad \sigma_2 \vdash e_2 \twoheadrightarrow \sigma_3 \vdash v_1 \\ \sigma_3 \vdash \sigma_3(n,a,v_1) \twoheadrightarrow \sigma_4 \vdash v_2 \qquad \sigma_3(n,a,v_1) \text{ is not a value expression}\end{array}}{\sigma_1 \vdash e_1.a(e_2) \twoheadrightarrow \sigma_4 \oplus \{(n,a,v_1) \mapsto [\![v_2]\!]\} \vdash v_2} \text{ (BAttrCached)}$$

$$\frac{\sigma_1 \vdash e \twoheadrightarrow \sigma_2 \vdash v_2}{\sigma_1 \vdash n.a(v_1) := e \twoheadrightarrow \sigma_2 \oplus \{(n,a,v_1) \mapsto [\![v_2]\!]\} \vdash v_2} \text{ (BCache)}$$

Big Step and Cache Expressions

An astute reader may have noticed that the big step semantics do not require the use of the cache expression to keep track of attribute evaluations, but still includes a semantic rule to evaluate cache expressions. Especially considering the concept of user-level expressions (Section 4.3.6), the inclusion of these semantics in the big step relation may seem unnecessary.

The reason that the big step relation includes semantics for the cache expression is to more closely match the multistep (and therefore single step) semantics. In Section 6.7 we prove that the multistep and big step relations are equivalent for *any* input expression and context function. If we did not include cache expression semantics in the big step relation, we could prove this property about all expressions *except* cache expressions, but it is a stronger property to prove that multistep and big step have completely identical semantics, and it does not cost much to add the extra rule to the big step semantics.

The single step relation is considered the "primary" expression of Saiga's semantics, and the big step relation is intended to model the same semantics from a different angle, providing alternate routes for proofs. Thus we include the evaluation of cache expressions in the big step semantics.

4.3.6 User-Level Expressions

In Section 4.2 we introduced a new expression form to represent attribute caching. The new step rule AttrFetchCached is the only step rule that 'creates' a non-value expression form, rather than just modifying an existing expression form. AttrFetchCached steps from an attribution expression into a new expression wrapped in a cache expression. The cache expression is not intended to be used as a *user-level expression*; it is only intended to be used as an intermediary expression state.

One of the reasons for this is that the cache expression makes no guarantees about the content of its subexpression. If a cache expression could come from anywhere, we would have no guarantee of its contents, and nonsensical modifications could be made to the context function. Caching is not intended to allow arbitrary context function modifications, but only to write the computed value of an attribute expression over its initial form. Therefore we say that the cache expression is not a production of *user-level expressions*, and we assume that we never start evaluation with a cache expression and no context function ever directly returns a cache expression.

4.4 Conclusion

We have presented the extended Saiga calculus, which builds on the core Saiga calculus by introducing parameterised attributes and attribute caching. We introduced the full semantics of parameterisation and caching in Sections 4.1 and 4.2 respectively, along with an example showing the use of these new features for each section. In Section 4.3 we explored some of the design decisions made while developing this calculus, and some of the implications of these decisions.

As we did for the core calculus, we aimed to create an extended calculus that is *simple*, *general*, and *domain-specific*. Introducing parameterised attributes was achieved with only one additional semantic step rule, and modifications to one rule in the type rules and one production rule in the expression grammar. Introducing attribute caching required all step rules to carry a changing context function, and for one rule (CacheWrite) to modify the context function. The simplicity of these changes to implement significant additional functionality demonstrate the flexibility of the Saiga calculus as a base model for attribute grammar evaluation. All changes introduced are directly related to attribute evaluation, so the calculus remains as domain-specific as in the core calculus.

In Chapter 5 we present the higher order Saiga calculus, allowing decoratable nodes to be created during evaluation. The higher order calculus builds from the extended calculus presented in this chapter and is presented in the same manner as in Chapters 3 and 4.

Behold the rain which descends from heaven upon our vineyards, and which incorporates itself with the grapes to be changed into wine; a constant proof that God loves us, and loves to see us happy.

Benjamin Franklin

5 Higher Order Saiga

In this chapter we describe how higher order attributes are implemented in Saiga. We build upon the semantics presented in Chapter 4. While the new semantic rules required to specify higher order attribution are not complex, their implications are considerable enough to warrant their own chapter for discussion. Further, reasoning about programs that allow higher order attributes can be more difficult than reasoning about programs that use the features in our extended calculus, so it is useful to present a clear line between "extended Saiga" (Chapter 4) and "higher order Saiga" (this chapter).

We present the higher order calculus in Section 5.1, an example of its use in Section 5.2, and a discussion of the implications of the new semantics in Section 5.3.

5.1 Higher Order Attributes

Higher order attributes, as discussed in Section 2.3.4, allow evaluation to create new trees, such that nodes in these new trees can have attributes evaluated upon them as if they were in the original tree. However to allow a node to be "created", we must first revise our notion of what it means for a node to "exist".

Saiga is an expression language, and all evaluation in Saiga starts with an expression. It is reasonable to assume that the entry point to any Saiga program will be an attribution expression, which queries some node for some attribute (as is the case in all examples in this thesis). Therefore, to start some evaluation, we need to *know about* some node that the context function has some knowledge of.

In this thesis, the set of nodes $\boldsymbol{N}$ has always been considered an abstract type; the only property of $\boldsymbol{N}$ that we care about is that nodes can be compared to each other. Everything we know about a particular node is stored in a context function, accessible via some attributes. A node's properties exist only in terms of the expressions that a given context function can produce for it. Further, a context function is a total function – an output is available for every possible set of inputs – so if we request the properties of any node n under any context function σ, some expressions will be returned.

However, we would like to draw a line between nodes that "exist" and nodes that "do

not exist", according to our particular definition of node existence. Firstly we say that node existence is dependent upon a particular context function; some node n might exist in σ_1 but not in σ_2. Secondly, we say that there is a special node n_{null} which exists in all context functions. Beyond this, we say that n exists in σ iff there is some set of inputs (n_1, a, p) (where n_1 is a different node that exists in σ) where $\sigma(n_1, a, p) = [\![n]\!]$.

In simpler terms, we say that n_{null} exists in all contexts, and any nodes that are structural attributes of existent nodes are also existent. This means that for any tree rooted at n_0 to exist in σ, there must be some attribute a and parameter p for which $\sigma(n_{null}, a, p) = [\![n_0]\!]$. It is likely that the attribute a in this case is a tree construction attribute (see Section 5.3.3).

Definition 5.1.1.

$$\begin{aligned}\forall n \forall \sigma, (n \text{ exists in } \sigma) \\ \iff (\exists n_1 \exists a_1 \exists p_1, (n_1 \text{ exists in } \sigma) \wedge \sigma(n_1, a_1, p_1) = [\![n]\!]) \\ \vee\ n = n_{null}\end{aligned}$$

For brevity, we sometimes use the notation $n \in \sigma$ to indicate that n exists in σ, and $n \notin \sigma$ to indicate that n does not exist in σ. We formally define node existence here to pave the way for *node construction*, which is necessary for the construction of trees during evaluation, which is the basis for higher order attribute evaluation.

The strategy we employ to allow higher order attributes to be defined in Saiga is to provide a mechanism for attribute evaluation to "spawn" a fresh node n, with some dynamically defined attribute values assigned to it. These attribute values are, as usual, defined in terms of a context function's output expressions. The new node n can have intrinsic and extrinsic attributes evaluated on it, which is what makes this "higher order" evaluation. Some of the properties of the new node n may also be higher order nodes, meaning that new trees can be created recursively.

5.1.1 Expression Language

We begin by defining a new production for Saiga's expression grammar, as shown below.

e	::=	$[\![v]\!]$	value
	|	`IF` e_1 `THEN` e_2 `ELSE` e_3	conditional
	|	e_1`(`e_2`)`	function application
	|	e_1`.`a`(`e_2`)`	attribution
	|	n`.`a`(`p`):=` e_1	cache
	|	`MK`f_l	node construction

We add the highlighted *node construction* expression, written `MK`f_l. The metavariable f_l represents a function in $\boldsymbol{N} \rightarrow list(\boldsymbol{A}_u \times \boldsymbol{E})$. $\boldsymbol{A}_u$ represents the set of attributes a where $\rho(a) = unit$.

Essentially f_l is a list of attribute expressions to write to a context function for a newly created node, where each attribute must be non-parameterised (see Section 4.3.1). We add this non-parameterisation restriction because it simplifies our semantics, and because it is expected that only intrinsic attributes will be set during node construction, and intrinsic attributes tend to be non-parameterised.

Sometimes it is useful for the attributes of a new node to reference the node itself, so we parameterise this list over $\boldsymbol{N}$. This allows the expressions that define a node's attributes

to include a reference to the node itself, which is not known until the higher order attribute is evaluated. For example, if we wish to create a node which needs to recursively create some child, it will need to know its own identity to set the child's `parent` attribute. These semantics will be discussed further later in this chapter.

5.1.2 Type Rules

To accommodate the new expression production we add the new type rule TypeCstr, given below. We omit here the existing type rules that are given in Section 4.2.2.

$$\frac{\forall(n \in \boldsymbol{N}),\ ((a,e) \in f_l(n)),\ e : \tau(a)}{\texttt{MK}f_l : \boldsymbol{N}}\ \text{(TypeCstr)}$$

The new type rule TypeCstr restricts f_l such that for any input node n, every pair of attribute and expression (a,e) in $f_l(n)$ will have its expression e typed according to its attribute a. If this is satisfied, the type of a node construction expression is always $\boldsymbol{N}$.

5.1.3 Semantic Rules

For brevity we omit the existing semantic rules given in Section 4.2.3, showing only new and modified semantic rules.

$$\frac{\sigma(n,a,p) \text{ is not a value or } \texttt{MK} \text{ expression}}{\sigma \vdash [\![n]\!].a([\![p]\!]) \longrightarrow \sigma \vdash n.a(p) := \sigma(n,a,p)}\ \text{(AttrFetchCached)}$$

$$\frac{\sigma(n,a,p) = \texttt{MK}f_l \qquad n_1 \notin \sigma}{\sigma \vdash [\![n]\!].a([\![p]\!]) \longrightarrow \sigma \oplus \{(n,a,p) \mapsto [\![n_1]\!]\} \otimes n_1/f_l(n_1) \vdash [\![n_1]\!]}\ \text{(AttrFetchHO)}$$

Specifying higher order semantics involves modifying the AttrFetchCached rule and adding the new rule AttrFetchHO. We now have three ways to perform an attribute fetch, given by AttrFetchValue (simple fetch), AttrFetchCached (cached fetch), and AttrFetchHO (higher order construction). From here on we use the phrase "e is a `MK` expression" to indicate that e was derived from the node construction production, and takes the form $\texttt{MK}f_l$.

The new step rule AttrFetchHO requires that the context function returns some expression in the form $\texttt{MK}f_l$. Some node n_1 is selected that does not exist in σ, is used as part of a modification to the context function, and is returned as a value expression. f_l is a function which takes some node and returns some list l. l is a list of tuples of attributes (in $\boldsymbol{A}_u$) and expressions.

The AttrFetchHO step modifies the context function in two ways. The first modification is to immediately cache the node, attribute, and parameter that were used to create this higher order node with the value n_1. The second modification is a new type of context function transformation we define as follows.

$$\begin{aligned} \sigma \otimes n/[] &= \sigma \\ \sigma \otimes n/((a,e) :: l) &= \sigma \oplus \{(n,a,()) \mapsto e\} \otimes n/l \end{aligned}$$

The $\otimes$ operator used in AttrFetchHO indicates that each (a, e) in l will be written to the context function for the new node n (n_1 in AttrFetchHO) using the $\oplus$ operator. The unit value () is always used as the parameter, as each attribute a is in $\boldsymbol{A}_u$ by construction.

The high level effect is that when the context function σ returns some expression $\texttt{MK}f_l$, a node n_1 will be selected that does not exist in σ, the attribute occurrence being evaluated will be cached with the value n_1, and the list given by $f_l(n_1)$ will be used to overwrite a number of attribute equations for n_1 in σ. This is how new nodes are constructed and given their own attribute values. Everything known about a node is known via attributes, and this method is used to write intrinsic, structural, and even extrinsic attributes for the new node. A newly constructed node is indistinguishable from any other node in the tree.

5.2 Tree Optimisation Example

To demonstrate the use of higher order attributes in Saiga, we will consider an example creating an optimised version of a simple addition tree. Let's say we have a simple expression language consisting of a sequence of additions. A context free grammar for such a language might be as follows.

```
E  ::= int '+' E
   ::= int
```

We will represent trees from such a concrete grammar according to the following abstract grammar.

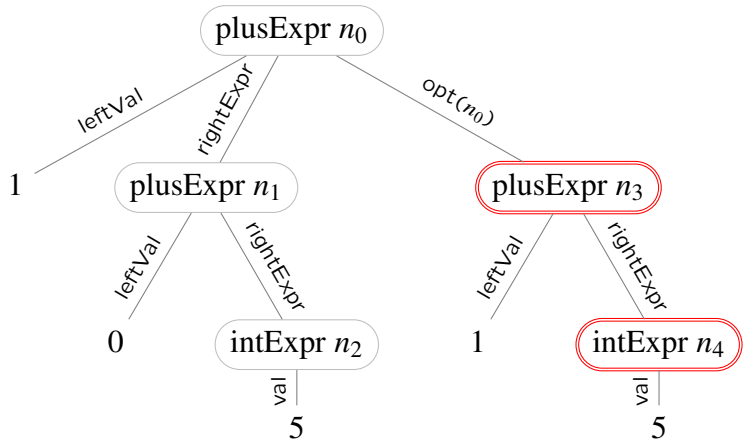
```
plusExpr ::= leftVal(integer), rightExpr(node)
intExpr ::= val(integer)
```

To evaluate such a tree, we could define the eval attribute as follows.

$$\sigma(n,\mathsf{eval}) = \begin{cases} [\![n]\!].\mathsf{val} & \text{(intExpr)} \\ [\![plus]\!]([\![n]\!].\mathsf{leftVal})([\![n]\!].\mathsf{rightExpr.eval}) & \text{(plusExpr)} \end{cases}$$

Now let's say we want to define an attribute opt which constructs and returns an optimised version of a subtree. We will achieve this by creating a clone of the subtree, but replacing any *plusExpr* expressions containing a zero with their right-hand child only. This will essentially remove all zeros from the expression, except for a potential zero on the far-right of the expression. It is possible that a subtree can optimise to a single integer node, as it would for a sequence of additions such as `0 + 0 + 5`.

For example, we would want the tree below, representing the sequence of additions `1 + 0 + 5` to optimise to the sequence of additions `1 + 5`, also shown in the tree below, with the optimised subtree given by nodes with a double red border.



In the tree above, the node n_3 is evaluated from $[\![n_0]\!]$.opt($[\![n_0]\!]$), and the node n_4 is evaluated from both $[\![n_1]\!]$.opt($[\![n_3]\!]$) and $[\![n_2]\!]$.opt($[\![n_3]\!]$). We can define the higher order attribute opt as follows.

$$\sigma(n, \mathsf{opt}, n_p) = \begin{cases} \texttt{MK}\lambda\_,[(\mathsf{parent}, [\![n_p]\!]), & \text{(intExpr)} \\ \quad (\mathsf{nodeType}, [\![\mathit{intExpr}]\!]), & \\ \quad (\mathsf{val}, [\![n]\!].\mathsf{val})] & \\ \texttt{IF } [\![\mathit{isZero}]\!]([\![n]\!].\mathsf{leftVal}) & \text{(plusExpr)} \\ \texttt{ THEN } [\![n]\!].\mathsf{rightExpr}.\mathsf{opt}([\![n_p]\!]) & \\ \texttt{ ELSE } [\![n]\!].\mathsf{clonePlus}([\![n_p]\!]) & \end{cases}$$

Notice that the opt attribute has a parameter n_p. We implement opt this way so that a newly created node can be anchored to the existing subtree via its parent attribute. This may or may not be necessary for all higher order attributes, but we implement it here as a demonstration. When opt is called on some node, the parameter given is a node value which will be written to the parent attribute on the newly constructed node. For example if we want to optimise the node n_0 with the new subtree being rooted under n_0, we evaluate $[\![n_0]\!]$.opt($[\![n_0]\!]$).

When opt is called on an *intExpr* node, a MK expression is returned (as shown above). The function in this MK expression returns a static list, ignoring its parameter. This is because an *intExpr* node will have no children, so we will not need access to the *intExpr*'s node reference to construct it. The parameter n_p is used as the value for the parent attribute. The nodeType attribute is set to the value *intExpr*, and the val attribute is given the *non-value* expression $[\![n]\!]$.val, specifying that the new node's val attribute will be derived from n's val attribute when it is evaluated.

When opt is called on a *plusExpr* node, a conditional expression is returned which checks the node's leftVal to decide whether to return a recursively optimised version of the right child, or to clone the node with clonePlus, which is defined below. Note that if the former path is taken, the opt attribute on the child is called with a parameter taken directly from the parent call (underlined above). This is because if the current addition "fork" contains a zero, it is to be ignored entirely, and the parent of the right child should be linked to this fork's parent, bypassing it in the new tree. You can see an example of this in the tree given above, where the new node n_4 (the optimised version of n_2) is rooted under the node

n_3 (the optimised version of n_0), and n_1 is bypassed entirely.

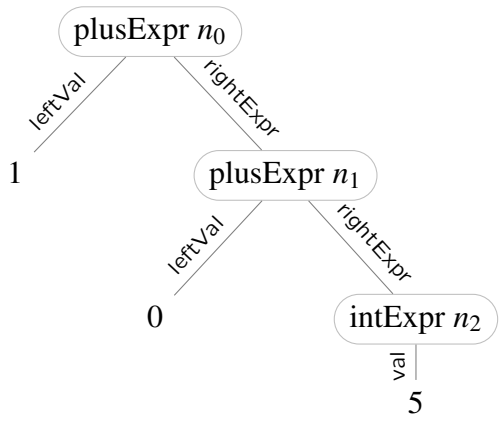
$$\sigma(n, \mathsf{clonePlus}, n_p) = \begin{cases} \texttt{MK}\lambda n_1, [(\mathsf{parent}, [\![n_p]\!]), \\ \quad (\mathsf{nodeType}, [\![\mathit{plusExpr}]\!]), \\ \quad (\mathsf{leftVal}, [\![n]\!].\mathsf{leftVal}), \\ \quad (\mathsf{rightExpr}, [\![n]\!].\mathsf{rightExpr}.\mathsf{opt}([\![n_1]\!]))] \end{cases}$$

The clonePlus attribute is always called on a *plusExpr* node, and always returns a `MK` expression. Note that in the above definition there are three nodes listed: n, n_p, and n_1. n is the node that clonePlus is being evaluated upon. n_p is the parameter given for that evaluation, and in this case represents the desired parent reference for the new node. n_1 is the newly created node, which exists now as the parameter of a lambda expression, but will be replaced with the appropriate node when the higher order evaluation takes place.

The list produced for a clonePlus construction uses the parameter n_p directly for the parent attribute, and the value *plusExpr* for nodeType. The leftVal attribute is copied directly, as val was in the opt attribute. The rightExpr attribute is given an expression that recursively optimises the rightExpr of the given node, using n_1 (the newly created node) as the parameter, meaning n_1 will be its parent.

Once clonePlus is evaluated under σ on some node n, the value of $\sigma(n, \mathsf{rightExpr}, \_)$ will be the *non-value* expression underlined above. Higher order construction happens here recursively, but only on-demand. If $[\![n]\!].\mathsf{rightExpr}$ is never evaluated, the higher order node associated with that attribute will never be constructed, but when it is evaluated, it will be constructed transparently, creating a new node whose parent will reference the new node n_1. Saiga uses *on-demand recursive higher order construction*. For further discussion of this design decision and its alternatives, see Section 5.3.3.

To demonstrate how such an attribute will evaluate, we will evaluate an optimisation of the following tree, which represents the addition expression `1 + 0 + 5`.



To optimise this tree, we evaluate the expression $[\![n_0]\!].\mathsf{opt}([\![n_0]\!])$, which will create an optimised version of the n_0 subtree, whose parent will be the original node n_0.

$\sigma \vdash [\![n_0]\!].\mathsf{opt}([\![n_0]\!])$

(1) $\longrightarrow$ (AttrFetchCached)

$\sigma \vdash n_0.\mathsf{opt}(n_0) :=$
$\mathtt{IF}\ [\![isZero]\!]([\![n_0]\!].\mathsf{leftVal})$
$\mathtt{THEN}\ [\![n_0]\!].\mathsf{rightExpr}.\mathsf{opt}([\![n_0]\!])$
$\mathtt{ELSE}\ [\![n_0]\!].\mathsf{clonePlus}([\![n_0]\!])$

(2) $\longrightarrow$ (CacheStep, CondStep, FunParStep, AttrFetchValue)

$\sigma \vdash n_0.\mathsf{opt}(n_0) :=$
$\mathtt{IF}\ [\![isZero]\!]([\![1]\!])$
$\mathtt{THEN}\ [\![n_0]\!].\mathsf{rightExpr}.\mathsf{opt}([\![n_0]\!])$
$\mathtt{ELSE}\ [\![n_0]\!].\mathsf{clonePlus}([\![n_0]\!])$

(3) $\longrightarrow$ (CacheStep, CondStep, FunApp)

$\sigma \vdash n_0.\mathsf{opt}(n_0) :=$
$\mathtt{IF}\ [\![false]\!]$
$\mathtt{THEN}\ [\![n_0]\!].\mathsf{rightExpr}.\mathsf{opt}([\![n_0]\!])$
$\mathtt{ELSE}\ [\![n_0]\!].\mathsf{clonePlus}([\![n_0]\!])$

(4) $\longrightarrow$ (CacheStep, CondFalse)

$\sigma \vdash n_0.\mathsf{opt}(n_0) :=$
$[\![n_0]\!].\mathsf{clonePlus}([\![n_0]\!])$

(5) $\longrightarrow$ (CacheStep, AttrFetchHO, as $n_3 \notin \sigma$)

$\sigma_2 \vdash n_0.\mathsf{opt}(n_0) :=\ [\![n_3]\!]$

(6) $\longrightarrow$ (CacheWrite)

$\sigma_3 \vdash [\![n_3]\!]$

The full definitions of σ_2 and σ_3 are shown below. We give the definition of σ_2 using the $\otimes$ operator instead of its expanded form using $\oplus$ operators, as it is a more concise notation. The $\otimes$ operation is of course equivalent to repeated uses of the $\oplus$ operation.

$$\sigma_2 = \sigma \oplus \{(n_0, \mathsf{clonePlus}, n_0) \mapsto [\![n_3]\!]\} \otimes n_3/[(\mathsf{parent}, [\![n_0]\!]),\ (\mathsf{nodeType}, [\![plusExpr]\!]),\ (\mathsf{leftVal}, [\![n_0]\!].\mathsf{leftVal}),\ (\mathsf{rightExpr}, [\![n_0]\!].\mathsf{rightExpr}.\mathsf{opt}([\![n_3]\!]))]$$

$$\sigma_3 = \sigma_2 \oplus \{(n_0, \mathsf{opt}, n_0) \mapsto [\![n_3]\!]\}$$

At the end of the above evaluation, the value n_3 is reached. However n_3 is the only node that has been constructed during this evaluation. The other two nodes n_1 and n_2 have not been considered for cloning yet, due to the on-demand nature of Saiga's higher order node

construction. To test out the full semantics of the opt attribute, we will need to force evaluation of these attributes via some traversal. To this end, we will evaluate $[\![n_0]\!]$.opt.eval under σ_3.

$\sigma_3 \vdash [\![n_0]\!]$.opt.eval

(1) $\longrightarrow$ (AttrNodeStep, AttrFetchValue)

$\sigma_3 \vdash [\![n_3]\!]$.eval

(2) $\longrightarrow$ (AttrFetchCached)

$\sigma_3 \vdash n_3$.eval:= $[\![plus]\!]([\![n_3]\!]$.leftVal$)([\![n_3]\!]$.rightExpr.eval$)$

(3) $\longrightarrow$ (CacheStep, FunStep, FunParStep, AttrFetchCached)

$\sigma_3 \vdash n_3$.eval:= $[\![plus]\!]($$n_3$.leftVal:= $[\![n_0]\!]$.leftVal$)([\![n_3]\!]$.rightExpr.eval$)$

(4) $\longrightarrow$ (CacheStep, FunStep, FunParStep, CacheStep, AttrFetchValue)

$\sigma_3 \vdash n_3$.eval:= $[\![plus]\!]($$n_3$.leftVal:= $[\![1]\!]$$)([\![n_3]\!]$.rightExpr.eval$)$

(5) $\longrightarrow$ (CacheStep, FunStep, FunParStep, CacheWrite)

$\sigma_4 \vdash n_3$.eval:= $[\![plus]\!]([\![1]\!])([\![n_3]\!]$.rightExpr.eval$)$

(6) $\longrightarrow$ (CacheStep, FunStep, FunApp)

$\sigma_4 \vdash n_3$.eval:= $[\![plus(1)]\!]([\![n_3]\!]$.rightExpr.eval$)$

(7) $\longrightarrow$ (CacheStep, FunParStep, AttrNodeStep, AttrFetchCached)

$\sigma_4 \vdash n_3$.eval:= $[\![plus(1)]\!]($$n_3$.rightExpr:= $[\![n_0]\!]$.rightExpr.opt$([\![n_3]\!])$$)$.eval$)$

(8) $\longrightarrow$ (CacheStep, FunParStep, CacheStep, AttrNodeStep, AttrNodeStep, AttrFetchValue)

$\sigma_4 \vdash n_3$.eval:= $[\![plus(1)]\!]($$n_3$.rightExpr:= $[\![n_1]\!]$.opt$([\![n_3]\!])$$)$.eval$)$

(9) $\longrightarrow$ (CacheStep, FunParStep, CacheStep, AttrNodeStep, AttrFetchCached)

$\sigma_4 \vdash n_3$.eval:= $[\![plus(1)]\!]($$n_3$.rightExpr:= n_1.opt(n_3):=
IF $[\![isZero]\!]([\![n_1]\!]$.leftVal$)$
THEN $[\![n_1]\!]$.rightExpr.opt$([\![n_3]\!])$
ELSE $[\![n_1]\!]$.clonePlus$([\![n_3]\!])$$)$.eval$)$

(10) $\longrightarrow$ (CacheStep, FunParStep, CacheStep, CacheStep, CondStep, FunParStep, AttrFetchValue)

$\sigma_4 \vdash n_3$.eval:= $[\![plus(1)]\!]($$n_3$.rightExpr:= n_1.opt(n_3):=
IF $[\![isZero]\!]([\![0]\!])$
THEN $[\![n_1]\!]$.rightExpr.opt$([\![n_3]\!])$
ELSE $[\![n_1]\!]$.clonePlus$([\![n_3]\!])$$)$.eval$)$

(11) $\longrightarrow$ (CacheStep, FunParStep, CacheStep, CacheStep, CondStep, FunApp)

$\sigma_4 \vdash n_3$.eval:= $[\![plus(1)]\!]($$n_3$.rightExpr:= n_1.opt(n_3):=
IF $[\![true]\!]$
THEN $[\![n_1]\!]$.rightExpr.opt$([\![n_3]\!])$
ELSE $[\![n_1]\!]$.clonePlus$([\![n_3]\!])$$)$.eval$)$

(12) $\longrightarrow$ (CacheStep, FunParStep, CacheStep, CacheStep, CondTrue)

$\sigma_4 \vdash n_3.\mathsf{eval} := [\![plus(1)]\!]((n_3.\mathsf{rightExpr} := n_1.\mathsf{opt}(n_3)) := [\![n_1]\!].\mathsf{rightExpr}.\mathsf{opt}([\![n_3]\!])).\mathsf{eval})$

(13) $\longrightarrow$ (CacheStep, FunParStep, CacheStep, CacheStep, AttrNodeStep, AttrFetchValue)

$\sigma_4 \vdash n_3.\mathsf{eval} := [\![plus(1)]\!]((n_3.\mathsf{rightExpr} := n_1.\mathsf{opt}(n_3)) := [\![n_2]\!].\mathsf{opt}([\![n_3]\!])).\mathsf{eval})$

(14) $\longrightarrow$ (CacheStep, FunParStep, CacheStep, CacheStep, AttrFetchHO, as $n_4 \notin \sigma_4$)

$\sigma_5 \vdash n_3.\mathsf{eval} := [\![plus(1)]\!]((n_3.\mathsf{rightExpr} := n_1.\mathsf{opt}(n_3)) := \overline{[\![n_4]\!]}).\mathsf{eval})$

(15) $\longrightarrow$ (CacheStep, FunParStep, CacheStep, CacheWrite)

$\sigma_6 \vdash n_3.\mathsf{eval} := [\![plus(1)]\!]((n_3.\mathsf{rightExpr} := \overline{[\![n_4]\!]}).\mathsf{eval})$

(16) $\longrightarrow$ (CacheStep, FunParStep, CacheWrite)

$\sigma_7 \vdash n_3.\mathsf{eval} := [\![plus(1)]\!](\overline{[\![n_4]\!]}.\mathsf{eval})$

(17) $\longrightarrow$ (CacheStep, FunParStep, AttrFetchCached)

$\sigma_7 \vdash n_3.\mathsf{eval} := [\![plus(1)]\!](\overline{n_4.\mathsf{eval} := [\![n_4]\!].\mathsf{val}})$

(18) $\longrightarrow$ (CacheStep, FunParStep, CacheStep, AttrFetchCached)

$\sigma_7 \vdash n_3.\mathsf{eval} := [\![plus(1)]\!](\overline{n_4.\mathsf{eval} := n_4.\mathsf{val} := [\![n_2]\!].\mathsf{val}})$

(19) $\longrightarrow$ (CacheStep, FunParStep, CacheStep, CacheStep, AttrFetchValue)

$\sigma_7 \vdash n_3.\mathsf{eval} := [\![plus(1)]\!](n_4.\mathsf{eval} := n_4.\mathsf{val} := \overline{[\![5]\!]})$

(20) $\longrightarrow$ (CacheStep, FunParStep, CacheStep, CacheWrite)

$\sigma_8 \vdash n_3.\mathsf{eval} := [\![plus(1)]\!](n_4.\mathsf{eval} := \overline{[\![5]\!]})$

(21) $\longrightarrow$ (CacheStep, FunParStep, CacheWrite)

$\sigma_9 \vdash n_3.\mathsf{eval} := [\![plus(1)]\!](\overline{[\![5]\!]})$

(22) $\longrightarrow$ (CacheStep, FunApp)

$\sigma_9 \vdash n_3.\mathsf{eval} := \overline{[\![6]\!]}$

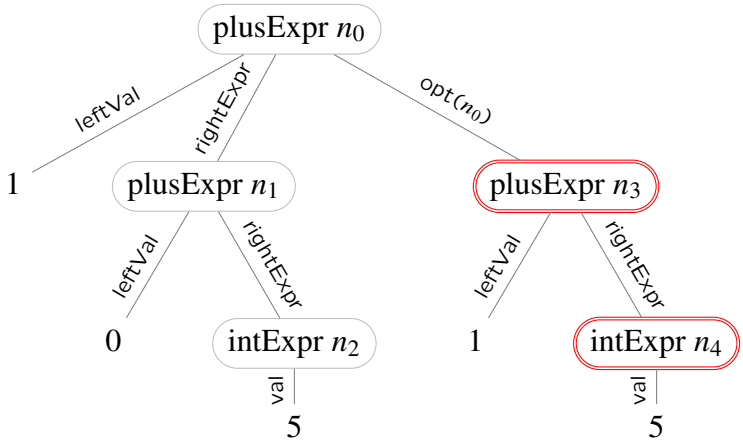
(23) $\longrightarrow$ (CacheWrite)

$\sigma_{10} \vdash [\![6]\!]$

The full definitions of σ_4 to σ_{10} are shown below.

$$\begin{aligned}
\sigma_4 &= \sigma_3 \oplus \{(n_3, \mathsf{leftVal}, _) \mapsto [\![1]\!]\} \\
\sigma_5 &= \sigma_4 \oplus \{(n_2, \mathsf{opt}, n_3) \mapsto [\![n_4]\!]\} \otimes n_4/[(\mathsf{parent}, [\![n_3]\!]), \\
&\qquad\qquad (\mathsf{nodeType}, [\![\mathit{intExpr}]\!]), \\
&\qquad\qquad (\mathsf{val}, [\![n_2]\!].\mathsf{val})] \\
\sigma_6 &= \sigma_5 \oplus \{(n_1, \mathsf{opt}, n_3 \mapsto [\![n_4]\!]\} \\
\sigma_7 &= \sigma_6 \oplus \{(n_3, \mathsf{rightExpr}, _) \mapsto [\![n_4]\!]\} \\
\sigma_8 &= \sigma_7 \oplus \{(n_4, \mathsf{val}, _) \mapsto [\![5]\!]\} \\
\sigma_9 &= \sigma_8 \oplus \{(n_4, \mathsf{eval}, _) \mapsto [\![5]\!]\} \\
\sigma_{10} &= \sigma_9 \oplus \{(n_3, \mathsf{eval}, _) \mapsto [\![6]\!]\}
\end{aligned}$$

The expression $[\![n_0]\!]$.opt.eval, evaluated under σ_3, yielded the value $[\![6]\!]$ in 23 steps. During evaluation, the higher order node n_4 was constructed, which is a clone of n_2. Below we repeat the tree presented earlier in this section, which is a representation of σ_{10}, the output context function of the above evaluation.



The higher order subtree rooted at n_3 contains only the integers 1 and 5, and represents an optimised version of the tree rooted at n_0.

5.3 Discussion

In this section we will discuss some of the details, design decisions, and caveats of the higher order calculus presented in Section 5.1.

5.3.1 User-Level Expressions

In Section 4.3.6 we explained that cache expressions are not intended to be evaluated directly, but only exist as an intermediate expression to keep track of attribute evaluation. The new `MK` expression is similar in that it is not intended for wide use like other expression productions. Indeed, a `MK` expression cannot be evaluated directly – notice in Section 5.1.3 that there is no derivation of the step relation that begins with a `MK` expression. This also means that no expression that has a `MK` expression as a subexpression will be able to fully evaluate.

The MK expression is processed only when it is the expression returned by a context function. Therefore we will say that the MK expression is also not a *user-level expression*, unless it is being returned from the context function.

The reason for this is that higher order attributes are designed to be *attributes*. Our aim is not to add the ability for our expression language to create a subtree, but for the evaluation of a particular attribute to create a subtree. We specify node construction using an expression production (the MK expression) because a context function always returns an expression, and we need to encode the properties of a newly created node somehow. Therefore we only evaluate a MK expression when it is being returned from the context function.

One of the reasons we choose to closely couple attribute fetching and node construction is to provide a strict place to anchor a newly constructed node. When we say "anchor", we are not talking about the parent attribute; we are talking about a newly constructed node necessarily existing in a new context function. If some node n_1 does not exist in some context function σ_1, then there is no set of inputs (n, a, p) such that $\sigma_1(n, a, p) = [\![n_1]\!]$. However if we evaluate $[\![n]\!].\mathsf{HOattr}([\![p]\!])$ under σ_1, which constructs the new node n_1 in the new context function σ_2, then we know that n_1 definitely does exist in σ_2. We know this because $\sigma_2(n, \mathsf{HOattr}, p) = [\![n_1]\!]$.

If we used the same semantics but allowed a MK expression to be evaluated arbitrarily, then it would be possible to "create" a new node and define some of its attributes, but for that node still not to exist in the new context function. Caching the newly constructed node to the attribute that created it forces there to be some relationship between new nodes and the attribute that created them, and behaves well with our concept of node existence.

5.3.2 Intrinsic Attributes on Higher Order Nodes

In Section 3.1.6 we introduced the concept of intrinsic attributes: attributes which represent information about a node that would be present before decoration in a traditional attribute grammar context. It stands to reason that all intrinsic attributes will be fully computed at the start of decoration, so we can always expect a value expression for an intrinsic attribute.

This changes when we start to build new nodes using higher order attributes. Consider the following snippet from a context function constructed during evaluation in Section 5.2.

$$\begin{aligned}\sigma_5 = \sigma_4 \oplus \{(n_2, \mathsf{opt}, n_3) \mapsto [\![n_4]\!]\} \otimes n_4/[&(\mathsf{parent}, [\![n_3]\!]),\\ &(\mathsf{nodeType}, [\![\mathit{intExpr}]\!]),\\ &(\mathsf{val}, [\![n_2]\!].\mathsf{val})]\end{aligned}$$

According to the above definition, the output of $\sigma_5(n_4, \mathsf{val}, ())$ is the expression $[\![n_2]\!].\mathsf{val}$. The val attribute is intrinsic, but the context function now contains a non-value expression to compute it.

This is due to the lazy evaluation strategy we use to implement higher order attributes. In most circumstances, this will not be an issue. When $[\![n_4]\!].\mathsf{val}$ is evaluated, an extra step will be taken to determine its value, which will be cached for future use.

The only potential issue here is in the reflection that can happen during attribute selection, if that is how we choose to define our context function. For example, if we were to write the expression $[\![n_2]\!].\mathsf{nodeType}$ as the equation for the attribute $[\![n_4]\!].\mathsf{nodeType}$, then the following selector would not perform how we might like it to, as the context function could

not always reflect on the value of a node's nodeType attribute.

$$\sigma(n, \mathsf{attr}, \_) = \begin{cases} e_1 & \text{(typeOne)} \\ e_2 & \text{otherwise} \end{cases}$$

As discussed in Section 3.2.2, the selector notation we use above will only work on attributes that always return a value expression. This is a shortcoming of our selector notations – that they may not play nicely with nodes created by higher order attributes. However, if an attribute such as nodeType is always written as a value during node construction, as it is in every example given in this thesis, then such a selector will still work as expected, even on higher order nodes. Further, assuming that all higher order construction happens from top down, the parent structural attribute will always hold a value, even on higher order nodes.

Almost all of the selectors presented in Section 3.2.2 are limited to using the parent and nodeType attributes, except for the selectors that check that a node is a particular child of its parent. In this case, we are guaranteed that the "downward" structural attribute that is being checked against will always be fully computed at the time of selection, as these attribution requests are always based at the child itself, which must necessarily be evaluated before having its attributes queried.

Finally, we will restate that the selector notations presented in Section 3.2.2 and used throughout this thesis are merely suggestions on how a context function might be described, and are not strictly part of the Saiga calculus. Saiga requires only that the context function is a mathematical function[1] and therefore will return an expression for any set of inputs.

5.3.3 Building Initial Trees

In a normal situation, attribute grammar evaluation begins with a tree, and decorates it using the existing structure and contents of the tree. Since higher order attributes introduce new tree structure in such a way that it is indistinguishable from original tree structure, there is no difference between "initial" tree construction and higher order evaluation. Therefore Saiga can be used to model not only attribution, but also tree construction.

If we consider the example presented in Section 5.2 of a tree representing a sequence of additions, we can define an attribute build, which will take some external tree structure as an argument and modify a context function to represent that tree, returning the node at its root. Let us assume some type t_t exists that represents a sequence of additions. We assume the underlying boolean functions *isPlus* and *isInt* which determine the shape of the root of t_t, and functions *getValue*, *getLeftValue*, and *getRightTree* which return a value of type integer, integer, and t_t respectively. We can therefore define the attribute build, which will "build" a

[1]However, for evaluation to proceed, some further conditions must be met, as discussed above.

t_t tree in a context function.

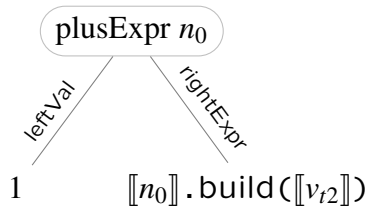
$$\sigma(n, \mathsf{build}, v_t) = \begin{cases} \mathtt{MK}\lambda n_1, [(\mathsf{nodeType}, [\![\mathit{plusExpr}]\!]), & \mathit{isPlus}(v_t) \\ \qquad (\mathsf{parent}, [\![n]\!]), & \\ \qquad (\mathsf{leftVal}, [\![\mathit{getLeftValue}(v_t)]\!]), & \\ \qquad (\mathsf{rightExpr}, [\![n_1]\!].\mathsf{build}([\![\mathit{getRightTree}(v_t)]\!]))] & \\ \mathtt{MK}\lambda \_, [(\mathsf{nodeType}, [\![\mathit{intExpr}]\!]), & \mathit{isInt}(v_t) \\ \qquad (\mathsf{parent}, [\![n]\!]), & \\ \qquad (\mathsf{val}, [\![\mathit{getValue}(v_t)]\!])] & \end{cases}$$

Note that the definition above uses the first parameter of the context function – the node that the attribute is being evaluated upon – to determine the parent of the newly created node. If we had some value v_{t1} of type t_t, we could evaluate $[\![n_{null}]\!].\mathsf{build}([\![v_{t1}]\!])$ under any σ_1, which would return some new node n_r and context function σ_2. n_r would represent the root of the tree described by t_t, and σ_2 would contain the full tree of t_t. The changes made to σ_1 would not override any existing nodes in σ_1.

For example, if v_{t1} represented the addition `1 + 2` such that $\mathit{isPlus}(v_{t1}) = \mathit{true}$, $\mathit{getLeftValue}(v_{t1}) = 1$, and $\mathit{getRightTree}(v_{t1}) = v_{t2}$, where v_{t2} represents the integer tree holding `2`, then evaluating $[\![n_{null}]\!].\mathsf{build}([\![v_{t1}]\!])$ under σ_1 would result in σ_2 as described below.

$$\begin{aligned} \sigma_2 = \sigma_1 \oplus \{(n_{null}, \mathsf{build}, v_{t1}) \mapsto [\![n_0]\!]\} \otimes n_0 / [&(\mathsf{nodeType}, [\![\mathit{plusExpr}]\!]), \\ &(\mathsf{parent}, [\![n_{null}]\!]), \\ &(\mathsf{leftVal}, [\![1]\!]), \\ &(\mathsf{rightExpr}, [\![n_0]\!].\mathsf{build}([\![v_{t2}]\!]))] \end{aligned}$$

The tree below represents the tree rooted at n_0 in the above context function.



Tree Construction and Parsing

Given that the semantics of the underlying functions *isPlus* and *isInt* are unspecified, they could just as easily be performing parsing operations as some object-oriented lookup. If the type t_t is actually the string type, we would only need *isPlus* to check a string for the contents of the "+" operator to return true or false, and *isInt* to be the negation of *isPlus* (as a node can only be of type *intExpr* or *plusExpr*). If *getValue* parses a string as an integer, *getLeftValue* returns the integer parse of the string before the first "+", and *getRightTree* returns everything after the first "+", then the exact same `build` attribute that is presented above would "build" a tree in a Saiga context function from a string.

Evaluating an expression such as $[\![n_{null}]\!].\mathsf{build}([\![\text{"1 + 2 + 3"}]\!])$ under any context σ_1 would, in a single step, produce the context function σ_2, defined as follows.

$$\sigma_2 = \sigma_1 \oplus \{(n_{null}, \mathsf{build}, \text{“1 + 2 + 3”} \mapsto [\![n_1]\!]\} \otimes n_1/[(\mathsf{nodeType}, [\![plusExpr]\!]),$$
$$(\mathsf{parent}, [\![n_{null}]\!]),$$
$$(\mathsf{leftVal}, [\![1]\!]),$$
$$(\mathsf{rightExpr}, [\![n_1]\!].\mathsf{build}([\![\text{“2 + 3”}]\!]))]$$

Note that we already have a context function and nodes that we can begin to evaluate attributes on, but parsing is not yet completed. The lazy nature of higher order construction means that nodes will only be constructed (in this case parsed and constructed) when they are queried.

Forcing Full Construction

We implement lazy construction because it is the most general approach. We can start with lazy construction and force a full higher order evaluation, but we could not start with static construction and force it to evaluate lazily. Continuing with the example of a sequence of additions, we could define the attribute finish which recursively forces the full construction of a tree returned by build.

$$\sigma(n, \mathsf{finish}) = [\![n]\!].\mathsf{finishAndReturn}([\![n]\!])$$
$$\sigma(n, \mathsf{finishAndReturn}, n_r) = \begin{cases} [\![n]\!].\mathsf{rightExpr}.\mathsf{finishAndReturn}([\![n_r]\!]) & \text{(plusExpr)} \\ [\![n_r]\!] & \text{otherwise} \end{cases}$$

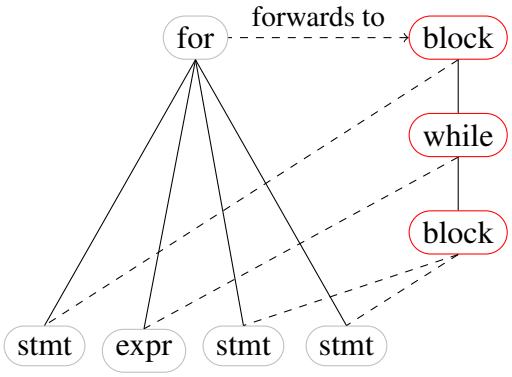
The finish attribute will recursively traverse the tree, evaluating the rightExpr attribute, and will finally return the root node. Semantically this attribute is the identity function, but its evaluation forces caching of the only non-value attribute constructed by the higher order build. If we want to start with a context function that describes a fully parsed tree for this problem, we can evaluate $[\![n_{null}]\!].\mathsf{build}([\![\text{“1 + 2 + 3”}]\!).\mathsf{finish}$ to obtain the root node of the constructed (and fully cached) tree.

5.3.4 Forwarding

Attribute forwarding, as discussed in Section 2.3.5, is an extension that is closely related to higher order attributes. According to Van Wyk “... the only substantial difference [between forwarding and higher order attributes] is that here the ‘copy rules’ for all relevant attributes are automatically generated” [41]. Our interpretation of the difference between forwarding and “normal” higher order attribution is that forwarding provides the notational convenience of forwarding all attributes to a higher order node (except for attributes that are explicitly defined on it).

To account for a range of different notations for defining attribute equations, we have relegated equation selection to the context function, which can be defined however we want it to be defined. Consider that not a single semantic rule presented in this thesis has been interested in the notations used to define an attribute; the semantics are only interested in the output of a context function. By abstracting equation notation away from attribute evaluation, we have created a calculus whose semantics are orthogonal to notation-focused extensions such as forwarding.

This said, we will show how a forwarding example such as the one presented in Section 2.3.5 might be implemented in Saiga. The example we refer to involves implementing the "for" iteration structure by forwarding it to an equivalent "while" structure. The diagram below from Section 2.3.5 demonstrates the desired forwarding pattern.



We can define a context function that provides the semantics of such a forwarding as follows. We assume `nodeType` may have at least the values *whileStmt*, *forStmt*, and *blockStmt*.

$$\sigma(n, \texttt{forwardsTo}) = \begin{cases} \texttt{MK}\lambda n_1, [(\texttt{stmt1}, [\![n]\!].\texttt{initStmt}), \ldots] & \text{(forStmt)} \\ [\![n]\!] & \text{otherwise} \end{cases}$$

$$\sigma(n, \texttt{eval}) = \begin{cases} \ldots & \text{(whileStmt)} \\ [\![n]\!].\texttt{forwardsTo}.\texttt{eval} & \text{(forStmt)} \end{cases}$$

$$\sigma(n, \texttt{pp}) = \begin{cases} [\![concat(\text{“while”})]\!](\ldots) & \text{(whileStmt)} \\ [\![concat(\text{“for”})]\!](\ldots) & \text{(forStmt)} \end{cases}$$

The above definitions will implement the same semantics as the example shown in Section 2.3.5, except that we have manually "forwarded" the `eval` attribute in the underlined attribute equation. The `pp` attribute is still defined explicitly for both *whileStmt* and *forStmt* nodes. As the context function can be defined however we want, we could "automate" this process with a definition like the following, assuming only the definition of `forwardsTo` from above.

$$\sigma(n, a, p) = \begin{cases} [\![concat(\text{“for”})]\!](\ldots) & \text{(forStmt) and } a = \texttt{pp} \\ [\![n]\!].\texttt{forwardsTo}.a & \text{(forStmt)} \\ \ldots & \text{otherwise} \end{cases}$$

Note that the above definition is not defining the output for any single attribute, but is defining the behaviour of the context function for any inputs. The first selector defines specific pretty-printing behaviour for the `pp` attribute, as in the example in Section 2.3.5. The second selector forwards any *other* attribute requests on a *forStmt* node to the higher order subtree returned by `forwardsTo`. Thus we have implemented notational convenience similar to that provided by traditional forwarding.

5.3.5 The Multistep Relation

While the semantics of the step relation have changed, the notation remains the same as in Chapter 4. We define the multistep relation again below, but it appears identical to the multistep semantics shown in Section 4.3.4. Of course the step relation we refer to in the following semantics is the higher order step relation, defined in Section 5.1.3.

$$\frac{}{\sigma \vdash e \longrightarrow^* \sigma \vdash e} \text{ (MultiRefl)}$$

$$\frac{\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2 \qquad \sigma_2 \vdash e_2 \longrightarrow^* \sigma_3 \vdash e_3}{\sigma_1 \vdash e_1 \longrightarrow^* \sigma_3 \vdash e_3} \text{ (MultiStep)}$$

5.3.6 The Big Step Relation

We extend the big step semantics presented in Sections 3.2.9 and 4.3.5 to include the new semantics presented in this chapter.

$$\frac{}{\sigma \vdash [\![v]\!] \twoheadrightarrow \sigma \vdash v} \text{ (BRefl)} \qquad \frac{\sigma_1 \vdash e_1 \twoheadrightarrow \sigma_2 \vdash \mathit{true} \qquad \sigma_2 \vdash e_2 \twoheadrightarrow \sigma_3 \vdash v}{\sigma_1 \vdash \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \twoheadrightarrow \sigma_3 \vdash v} \text{ (BCondTrue)}$$

$$\frac{\sigma_1 \vdash e_1 \twoheadrightarrow \sigma_2 \vdash \mathit{false} \qquad \sigma_2 \vdash e_3 \twoheadrightarrow \sigma_3 \vdash v}{\sigma_1 \vdash \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \twoheadrightarrow \sigma_3 \vdash v} \text{ (BCondFalse)}$$

$$\frac{\sigma_1 \vdash e_1 \twoheadrightarrow \sigma_2 \vdash v_1 \qquad \sigma_2 \vdash e_2 \twoheadrightarrow \sigma_3 \vdash v_2}{\sigma_1 \vdash e_1(e_2) \twoheadrightarrow \sigma_3 \vdash v_1(v_2)} \text{ (BFun)}$$

$$\frac{\sigma_1 \vdash e_1 \twoheadrightarrow \sigma_2 \vdash n \qquad \sigma_2 \vdash e_2 \twoheadrightarrow \sigma_3 \vdash v_1 \qquad \sigma_3(n,a,v_1) = [\![v_2]\!]}{\sigma_1 \vdash e_1.a(e_2) \twoheadrightarrow \sigma_3 \vdash v_2} \text{ (BAttrValue)}$$

$$\frac{\begin{array}{c}\sigma_1 \vdash e_1 \twoheadrightarrow \sigma_2 \vdash n \qquad \sigma_2 \vdash e_2 \twoheadrightarrow \sigma_3 \vdash v_1 \qquad \sigma_3 \vdash \sigma_3(n,a,v_1) \twoheadrightarrow \sigma_4 \vdash v_2 \\ \sigma_3(n,a,v_1) \text{ is not a value or a } \texttt{MK} \text{ expression}\end{array}}{\sigma_1 \vdash e_1.a(e_2) \twoheadrightarrow \sigma_4 \oplus \{(n,a,v_1) \mapsto [\![v_2]\!]\} \vdash v_2} \text{ (BAttrCached)}$$

$$\frac{\begin{array}{c}\sigma_1 \vdash e_1 \twoheadrightarrow \sigma_2 \vdash n \qquad \sigma_2 \vdash e_2 \twoheadrightarrow \sigma_3 \vdash v_1 \qquad \sigma_3(n,a,v_1) = \texttt{MK} f_l \\ n_2 \text{ does not exist in } \sigma_3\end{array}}{\sigma_1 \vdash e_1.a(e_2) \twoheadrightarrow \sigma_3 \oplus \{(n,a,v_1) \mapsto [\![n_2]\!]\} \otimes n_2/f_l(n_2) \vdash n_2} \text{ (BAttrHO)}$$

$$\frac{\sigma_1 \vdash e \twoheadrightarrow \sigma_2 \vdash v_2}{\sigma_1 \vdash n.a(v_1) := e \twoheadrightarrow \sigma_2 \oplus \{(n,a,v_1) \mapsto [\![v_2]\!]\} \vdash v_2} \text{ (BCache)}$$

Once again these semantics mirror those presented in Section 5.1.3. Proof of the equivalence of these two semantics is given in Section 6.7.

A good beer can be judged in only one sip...
but it's better to be thoroughly sure.

Czech Proverb

6 Metatheoretic Properties

In this chapter we present proofs for a number of metatheoretic properties about the Saiga calculus. As the calculus has been split into three stages: core, extended and higher order, we prove many of our theorems in three stages. For most theorems, the proof we present for the core calculus covers most cases for the extended calculus, and the extra proof steps provided for the extended calculus covers most cases for the higher order calculus. As the syntax for the step relation in the extended and higher order calculi are identical, we sometimes combine theorems for these two calculi together, if the proofs are the same.

The primary theorems we prove in this chapter are determinism of the step relation (Theorem 6), type determinism (Theorem 9), type preservation (Theorem 12), progress (Theorem 15), equivalence between the big step and multistep relations (Theorem 25), and cache irrelevance (Theorem 33). Most of these theorems relate to the step relation, whose semantics are given in Sections 3.1.5, 4.2.3 and 5.1.3. We use the terminology of "stepping" in this chapter. We say that some expression e_1 steps to some expression e_2 if the step relation $e_1 \longrightarrow e_2$ holds.

6.1 Axioms

We present a simple axiom about the underlying type system, which we use in some of the proofs presented in this chapter.

Axiom 1. Axiom 1 is a simple axiom of the underlying type system which states that if two function types are equal, and their first component type is equal, then their second component type is also equal.

$$\forall(t_1, t_2, t_3 \in \boldsymbol{T}),$$
$$t_1 \to t_2 = t_1 \to t_3 \tag{1.1}$$
$$\implies t_3 = t_2 \tag{1.2}$$

6.2 Values Can Not Step

Theorems 2 and 3 show that it is not possible for a step relation to have a value expression on its left. Theorem 2 states this theorem for the core calculus, and Theorem 3 states this theorem for the extended and higher order calculi.

Theorem 2.

$$\begin{aligned} &\forall (t \in \boldsymbol{T}), (v \in t), (e \in \boldsymbol{E}), \\ &[\![v]\!] \longrightarrow e \implies \mathit{False} \end{aligned} \tag{2.1}$$

Proof. Proof for this theorem is straightforward by examining all derivations of the step relation in the core calculus, given in Chapter 3. There are no step rules that allow a step relation with a value expression on the left. Therefore, such a relation is not possible. □

Theorem 3.

$$\forall (t \in \boldsymbol{T}), (v \in t), (e \in \boldsymbol{E}),$$

$$\forall \sigma_1, \sigma_2 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}, \tag{3.1}$$

$$\sigma_1 \vdash [\![v]\!] \longrightarrow \sigma_2 \vdash e \implies \mathit{False} \tag{3.2}$$

Proof. Proof of this theorem for the extended and higher order semantics is the same as in Theorem 2. There is no rule that allows a value expression on the left of a step relation. □

6.3 Step Determinism

We will prove determinism of the step relation with all three presented versions of our calculus. In all cases, the property we explore is that for any expression, if it can step into a new expression, there is only one expression that can be stepped to. We will prove this property first with the core semantics in Theorem 4, then with the extended and higher order semantics in Theorems 5 and 6.

6.3.1 Proof for the Core Calculus

Theorem 4. Assuming all evaluations are operating under the same context function σ, we have the following.

$$\forall (e_1, e_2, e_3 \in \boldsymbol{E}),$$

$$e_1 \longrightarrow e_2 \tag{4.1}$$

$$e_1 \longrightarrow e_3 \tag{4.2}$$

$$\implies e_3 = e_2 \tag{4.3}$$

Proof. We introduce the quantified variables e_1, e_2, and e_3, and proceed by structural induction on the derivation of $e_1 \longrightarrow e_2$ (4.1), generalising e_3. Each step rule will provide information about the forms of e_1 and e_2, so for each case examined we will present updated

versions of (4.1) to (4.3) to account for these transformations. If the derivation requires some secondary step relation, this will also be included.

Our induction principle is that Theorem 4 is given for any subrelation of (4.1). As there are eight derivations of the step relation in the core calculus, there are eight cases for us to consider.

Case 4.1 (CondStep)**.** If $e_1 \longrightarrow e_2$ is an instance of CondStep, then we know that e_1 has the form $\texttt{IF}\ e_{1a}\ \texttt{THEN}\ e_{1b}\ \texttt{ELSE}\ e_{1c}$ and e_2 has the form $\texttt{IF}\ e'_{1a}\ \texttt{THEN}\ e_{1b}\ \texttt{ELSE}\ e_{1c}$, and $e_{1a} \longrightarrow e'_{1a}$ (4.5).

$$\texttt{IF}\ e_{1a}\ \texttt{THEN}\ e_{1b}\ \texttt{ELSE}\ e_{1c} \longrightarrow \texttt{IF}\ e'_{1a}\ \texttt{THEN}\ e_{1b}\ \texttt{ELSE}\ e_{1c} \tag{4.4}$$
$$e_{1a} \longrightarrow e'_{1a} \tag{4.5}$$
$$\texttt{IF}\ e_{1a}\ \texttt{THEN}\ e_{1b}\ \texttt{ELSE}\ e_{1c} \longrightarrow e_3 \tag{4.6}$$
$$\Longrightarrow e_3 = \texttt{IF}\ e'_{1a}\ \texttt{THEN}\ e_{1b}\ \texttt{ELSE}\ e_{1c} \tag{4.7}$$

Now, there are three possible derivations for (4.6); CondStep, CondTrue, and CondFalse. CondTrue and CondFalse can only apply if e_{1a} is a value expression containing *true* or *false*, but (4.5) tells us that e_{1a} can step, so it must not be a value, according to Theorem 2. Therefore its derivation came from CondStep. (4.6) coming from CondStep tells us that e_3 has the form $\texttt{IF}\ e''_{1a}\ \texttt{THEN}\ e_{1b}\ \texttt{ELSE}\ e_{1c}$ and provides the additional relation $e_{1a} \longrightarrow e''_{1a}$ (4.10).

$$\texttt{IF}\ e_{1a}\ \texttt{THEN}\ e_{1b}\ \texttt{ELSE}\ e_{1c} \longrightarrow \texttt{IF}\ e'_{1a}\ \texttt{THEN}\ e_{1b}\ \texttt{ELSE}\ e_{1c} \tag{4.8}$$
$$e_{1a} \longrightarrow e'_{1a} \tag{4.9}$$
$$e_{1a} \longrightarrow e''_{1a} \tag{4.10}$$
$$\texttt{IF}\ e_{1a}\ \texttt{THEN}\ e_{1b}\ \texttt{ELSE}\ e_{1c} \longrightarrow \texttt{IF}\ e''_{1a}\ \texttt{THEN}\ e_{1b}\ \texttt{ELSE}\ e_{1c} \tag{4.11}$$
$$\Longrightarrow \texttt{IF}\ e''_{1a}\ \texttt{THEN}\ e_{1b}\ \texttt{ELSE}\ e_{1c} = \texttt{IF}\ e'_{1a}\ \texttt{THEN}\ e_{1b}\ \texttt{ELSE}\ e_{1c} \tag{4.12}$$

We can now use our inductive hypothesis on (4.9) and (4.10) to produce $e''_{1a} = e'_{1a}$. Using this equality to rewrite terms in (4.12) produces a reflexive equality.

Case 4.2 (CondTrue and CondFalse)**.** If $e_1 \longrightarrow e_2$ is an instance of CondTrue, then we know that e_1 has the form $\texttt{IF}\ [\![\mathit{true}]\!]\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_{1a}$.

$$\texttt{IF}\ [\![\mathit{true}]\!]\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_{1a} \longrightarrow e_2 \tag{4.13}$$
$$\texttt{IF}\ [\![\mathit{true}]\!]\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_{1a} \longrightarrow e_3 \tag{4.14}$$
$$\Longrightarrow e_3 = e_2 \tag{4.15}$$

The subexpression $[\![\mathit{true}]\!]$ cannot step, according to Theorem 2, and is trivially not equal to $[\![\mathit{false}]\!]$, so (4.14) must be derived from CondTrue. This means that e_3 must be equal to e_2, which is the goal. The same strategy is used to solve the CondFalse case.

Case 4.3 (FunStep, FunParStep, and AttrNodeStep)**.** The steps FunStep, FunParStep, and AttrNodeStep all involve stepping some subexpression, which requires recursive use of the step relation. In all of these cases, the approach used for Case 4.1 will similarly solve the goal.

Case 4.4 (FunApp)**.** If $e_1 \longrightarrow e_2$ is an instance of FunApp, then we know that e_1 has the form $[\![v_1]\!]([\![v_2]\!])$ and e_2 has the form $[\![v_1(v_2)]\!]$.

$$[\![v_1]\!]([\![v_2]\!]) \longrightarrow [\![v_1(v_2)]\!] \tag{4.16}$$
$$[\![v_1]\!]([\![v_2]\!]) \longrightarrow e_3 \tag{4.17}$$
$$\implies e_3 = [\![v_1(v_2)]\!] \tag{4.18}$$

Now, the only possible derivation of (4.17) is FunApp, as both the function and parameter subexpressions are values, so FunStep and FunParStep are not possible derivations. This immediately tells us that $e_3 = [\![v_1(v_2)]\!]$, which solves the goal.

Case 4.5 (AttrFetch)**.** If $e_1 \longrightarrow e_2$ is an instance of AttrFetch, then we know that e_1 has the form $[\![n_1]\!].a_1$ and e_2 has the form $\sigma(n_1, a_1)$.

$$[\![n_1]\!].a \longrightarrow \sigma(n_1, a_1) \tag{4.19}$$
$$[\![n_1]\!].a \longrightarrow e_3 \tag{4.20}$$
$$\implies e_3 = \sigma(n_1, a_1) \tag{4.21}$$

(4.20) can not be derived from AttrNodeStep, as this would require $[\![n_1]\!]$ to step, which is not possible. Therefore (4.20) is derived from AttrFetch, which tells us that $e_3 = \sigma(n_1, a_1)$, which is the goal.

All possible derivations of $e_1 \longrightarrow e_2$ have been considered, and the target has been shown in each case. Therefore Theorem 4 is proven. □

6.3.2 Proof for the Extended Calculus

Theorem 5 presents step determinism for the extended calculus. Since the extended calculus involves a context function on both sides of the step relation, step determinism states that not only the output expression is deterministic, but so is the output context function.

Theorem 5.

$$\forall(\sigma_1, \sigma_2, \sigma_3 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),$$
$$\forall(e_1, e_2, e_3 \in \boldsymbol{E}),$$
$$\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2 \tag{5.1}$$
$$\sigma_1 \vdash e_1 \longrightarrow \sigma_3 \vdash e_3 \tag{5.2}$$
$$\implies e_3 = e_2 \wedge \sigma_3 = \sigma_2 \tag{5.3}$$

Proof. As in the core theorem, we will proceed by structural induction on the derivation of $\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2$ (5.1), generalising both e_3 and σ_3.

The proofs for the derivations CondStep, CondTrue, CondFalse, FunApp, FunStep, FunParStep, and AttrNodeStep follow the same process as in Theorem 4, except that facts about the context function changing are also produced by inductive hypotheses and used for rewriting. Proof for the AttrParStep case also follows very similar logic. As such, the only cases examined here are AttrFetchValue, AttrFetchCached, CacheStep, and CacheWrite.

Case 5.1 (AttrFetchValue)**.** If $\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2$ is derived from AttrFetchValue, then we know that e_1 has the form $[\![n_1]\!].a_1([\![v_1]\!])$ and e_2 is $\sigma_1(n_1, a_1, v_1)$, and $\sigma_2 = \sigma_1$. Also we know that $\sigma_1(n_1, a_1, v_1) = [\![v_2]\!]$ for some v_2.

$$\sigma_1 \vdash [\![n_1]\!].a_1([\![v_1]\!]) \longrightarrow \sigma_1 \vdash [\![v_2]\!] \tag{5.4}$$
$$\sigma_1(n_1, a_1, v_1) = [\![v_2]\!] \tag{5.5}$$
$$\sigma_1 \vdash [\![n_1]\!].a_1([\![v_1]\!]) \longrightarrow \sigma_3 \vdash e_3 \tag{5.6}$$
$$\Longrightarrow e_3 = [\![v_2]\!] \wedge \sigma_3 = \sigma_1 \tag{5.7}$$

Given that $\sigma_1(n_1, a_1, v_1)$ is a value, (5.6) must also be derived from AttrFetchValue. Satisfying such a derivation requires σ_3 and e_3 to take forms that make the two equalities in the goal reflexive, after rewriting with (5.5).

Case 5.2 (AttrFetchCached)**.** If $\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2$ is derived from AttrFetchCached, then we know that e_1 has the form $[\![n_1]\!].a_1([\![v_1]\!])$ and e_2 has the form $n_1.a_1(v_1) := \sigma_1(n_1, a_1, v_1)$, and $\sigma_2 = \sigma_1$. We also know that $\sigma_1(n_1, a_1, v_1)$ is *not* a value.

$$\sigma_1 \vdash [\![n_1]\!].a_1([\![v_1]\!]) \longrightarrow \sigma_1 \vdash n_1.a_1(v_1) := \sigma_1(n_1, a_1, v_1) \tag{5.8}$$
$$\sigma_1(n_1, a_1, v_1) \text{ is not a value} \tag{5.9}$$
$$\sigma_1 \vdash [\![n_1]\!].a_1([\![v_1]\!]) \longrightarrow \sigma_3 \vdash e_3 \tag{5.10}$$
$$\Longrightarrow e_3 = n_1.a_1(v_1) := \sigma_1(n_1, a_1, v_1) \wedge \sigma_3 = \sigma_1 \tag{5.11}$$

Given that $\sigma_1(n_1, a_1, v_1)$ is not a value, (5.10) must be also derived from AttrFetch-Cached. Satisfying such a derivation requires σ_3 and e_3 to take forms that make the two equalities in the goal reflexive.

Case 5.3 (CacheStep)**.** If $\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2$ is derived from CacheStep, then we know that e_1 has the form $n_1.a_1(v_1) := e_{1a}$ and e_2 has the form $n_1.a_1(v_1) := e'_{1a}$. We also have $\sigma_1 \vdash e_{1a} \longrightarrow \sigma_2 \vdash e'_{1a}$ (5.13).

$$\sigma_1 \vdash n_1.a_1(v_1) := e_{1a} \longrightarrow \sigma_2 \vdash n_1.a_1(v_1) := e'_{1a} \tag{5.12}$$
$$\sigma_1 \vdash e_{1a} \longrightarrow \sigma_2 \vdash e'_{1a} \tag{5.13}$$
$$\sigma_1 \vdash n_1.a_1(v_1) := e_{1a} \longrightarrow \sigma_3 \vdash e_3 \tag{5.14}$$
$$\Longrightarrow e_3 = n_1.a_1(v_1) := e'_{1a} \wedge \sigma_3 = \sigma_2 \tag{5.15}$$

(5.13) means that e_{1a} is not a value (Theorem 3), so (5.14) cannot be derived from CacheWrite, and must be derived from CacheStep. Such a derivation assigns necessary values to σ_3 and e_3 as shown below.

$$\sigma_1 \vdash n_1 . a_1 (v_1) := e_{1a} \longrightarrow \sigma_2 \vdash n_1 . a_1 (v_1) := e'_{1a} \tag{5.16}$$
$$\sigma_1 \vdash e_{1a} \longrightarrow \sigma_2 \vdash e'_{1a} \tag{5.17}$$
$$\sigma_1 \vdash n_1 . a_1 (v_1) := e_{1a} \longrightarrow \sigma_3 \vdash n_1 . a_1 (v_1) := e''_{1a} \tag{5.18}$$
$$\sigma_1 \vdash e_{1a} \longrightarrow \sigma_3 \vdash e''_{1a} \tag{5.19}$$
$$\implies n_1 . a_1 (v_1) := e''_{1a} = n_1 . a_1 (v_1) := e'_{1a} \wedge \sigma_3 = \sigma_2 \tag{5.20}$$

The inductive hypothesis, with (5.17) and (5.19), produces equalities that solve the goal.

Case 5.4 (CacheWrite)**.** If $\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2$ is derived from CacheWrite, then we know that e_1 has the form $n_1 . a_1 (v_1) := [\![v_2]\!]$ and e_2 has the form $[\![v_2]\!]$. We also know that $\sigma_2 = \sigma_1 \oplus \{(n_1, a_1, v_1 \mapsto [\![v_2]\!]\}$.

$$\sigma_1 \vdash n_1 . a_1 (v_1) := [\![v_2]\!] \longrightarrow \sigma_1 \oplus \{(n_1, a_1, v_1 \mapsto [\![v_2]\!]\} \vdash [\![v_2]\!] \tag{5.21}$$
$$\sigma_1 \vdash n_1 . a_1 (v_1) := [\![v_2]\!] \longrightarrow \sigma_3 \vdash e_3 \tag{5.22}$$
$$\implies e_3 = [\![v_2]\!] \wedge \sigma_3 = \sigma_1 \oplus \{(n_1, a_1, v_1 \mapsto [\![v_2]\!]\} \vdash [\![v_2]\!] \tag{5.23}$$

(5.22) can only be derived from CacheWrite, as a value cannot step. By examining the necessary values of e_3 and σ_3 under CacheWrite, the goal becomes reflexive.

As all cases have been satisfied, Theorem 5 is proven. □

6.3.3 Proof for the Higher Order Calculus

Theorem 6 appears identical to Theorem 5, but is framed in the higher order context, so the expressions and type rules in question include the new forms described in Chapter 5.

Theorem 6.

$$\forall(\sigma_1, \sigma_2, \sigma_3 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),$$
$$\forall(e_1, e_2, e_3 \in \boldsymbol{E}),$$
$$\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2 \tag{6.1}$$
$$\sigma_1 \vdash e_1 \longrightarrow \sigma_3 \vdash e_3 \tag{6.2}$$
$$\implies e_3 = e_2 \wedge \sigma_3 = \sigma_2 \tag{6.3}$$

Proof. We proceed by structural induction on the derivation of the step relation (6.1), generalising e_3 and σ_3, which produces 13 cases. All cases bar AttrFetchHO are proved in the same fashion as in Theorem 5, so will be omitted here. We need only consider the case for AttrFetchHO.

Case 6.1 (AttrFetchHO)**.** If $\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2$ is derived from AttrFetchHO, then we know that e_1 has the form $[\![n_1]\!] . a_1 ([\![v_1]\!])$, we know $\sigma_1(n_1, a_1, v_1) = \texttt{MK}f_l$ and we know e_2 has the form $[\![n_1]\!]$, as well as the form of σ_2, as shown below.

$$\sigma_1 \vdash [\![n_1]\!].a_1([\![v_1]\!]) \longrightarrow \sigma_1 \oplus \{(n_1, a_1, v_1) \mapsto [\![n_2]\!]\} \otimes n_2/f_l(n_2) \vdash [\![n_2]\!] \tag{6.4}$$
$$\sigma_1(n_1, a_1, v_1) = \texttt{MK}f_l \tag{6.5}$$
$$n_2 \text{ does not exist in } \sigma_1 \tag{6.6}$$
$$\sigma_1 \vdash [\![n_1]\!].a_1([\![v_1]\!]) \longrightarrow \sigma_3 \vdash e_3 \tag{6.7}$$
$$\implies e_3 = [\![n_2]\!] \wedge \sigma_3 = \sigma_1 \oplus \{(n_1, a_1, v_1) \mapsto [\![n_2]\!]\} \otimes n_2/f_l(n_2) \tag{6.8}$$

Given that $\sigma_1(n_1, a_1, v_1)$ is a MK expression, (6.7) must be also derived from AttrFetchHO. Satisfying such a derivation requires σ_3 and e_3 to take forms that make the two equalities in the goal reflexive.

Theorem 6 is now proven, and the step relation is proven deterministic in the core, extended, and higher order calculi. □

6.4 Type Determinism

We want to prove that the type inference rules for Saiga expressions are deterministic. That is to say that each expression can have at most one type. We prove this first with core Saiga semantics in Theorem 7, then with extended semantics and higher order semantics in Theorems 8 and 9.

6.4.1 Proof for the Core Calculus

Theorem 7.

$$\forall(t_1, t_2 \in \boldsymbol{T}), (e \in \boldsymbol{E}),$$
$$e : t_1 \tag{7.1}$$
$$e : t_2 \tag{7.2}$$
$$\implies t_2 = t_1 \tag{7.3}$$

Proof. We proceed by structural induction on e. The inductive hypothesis is that the type rules are deterministic for any subexpression of e. This produces four cases, which we will examine individually. These cases will make use of core Saiga's type inference rules, which are shown in Section 3.1.4.

Case 7.1 (e is a value)**.**

$$[\![v]\!] : t_1 \tag{7.4}$$
$$[\![v]\!] : t_2 \tag{7.5}$$
$$\implies t_2 = t_1 \tag{7.6}$$

Both (7.4) and (7.5) can only be derived from TypeVal, which means both t_1 and t_2 are equal to whatever type v holds in the underlying system. Therefore the goal is reflexive.

Case 7.2 (e is a conditional expression)**.**

$$\texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3 : t_1 \tag{7.7}$$
$$\texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3 : t_2 \tag{7.8}$$
$$\implies t_2 = t_1 \tag{7.9}$$

Both (7.7) and (7.8) can only be derived from TypeCond, which means, for both cases, the first subexpression e_1 is of type *boolean*, and that e_2 and e_3 are both some type t_1 and t_2. Expanding these derivations yields the following.

$$\texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3 : t_1 \tag{7.10}$$
$$e_1 : \text{boolean} \tag{7.11}$$
$$e_2 : t_1 \tag{7.12}$$
$$e_3 : t_1 \tag{7.13}$$
$$\texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3 : t_2 \tag{7.14}$$
$$e_1 : \text{boolean} \tag{7.15}$$
$$e_2 : t_2 \tag{7.16}$$
$$e_3 : t_2 \tag{7.17}$$
$$\implies t_2 = t_1 \tag{7.18}$$

Applying the inductive hypothesis to (7.12) and (7.16) will yield the goal exactly.

Case 7.3 (e is a function application expression)**.**

$$e_1(e_2) : t_1 \tag{7.19}$$
$$e_1(e_2) : t_2 \tag{7.20}$$
$$\implies t_2 = t_1 \tag{7.21}$$

Both (7.19) and (7.20) can only be derived from TypeFun, which means, for both cases, that the first subexpression is of some function type, and the second subexpression is of the respective input type.

$$e_1(e_2) : t_{1b} \tag{7.22}$$
$$e_1 : t_{1a} \to t_{1b} \tag{7.23}$$
$$e_2 : t_{1a} \tag{7.24}$$
$$e_1(e_2) : t_{2b} \tag{7.25}$$
$$e_1 : t_{2a} \to t_{2b} \tag{7.26}$$
$$e_2 : t_{2a} \tag{7.27}$$
$$\implies t_{2b} = t_{1b} \tag{7.28}$$

The inductive hypothesis on (7.24) and (7.27) yields $t_{2a} = t_{1a}$. Rewriting this, and then applying the inductive hypothesis to (7.23) and (7.26) yields $t_{1a} \to t_{2b} = t_{1a} \to t_{1b}$. We can then apply Axiom 1, which provides $t_{2b} = t_{1b}$, which is the goal.

Case 7.4 (e is an attribution expression)**.**

$$e_1 \,.\, a : t_1 \tag{7.29}$$
$$e_1 \,.\, a : t_2 \tag{7.30}$$
$$\implies t_2 = t_1 \tag{7.31}$$

Both (7.29) and (7.30) can only be derived from TypeAttr, which means, for both cases, that their subexpression is of type $\boldsymbol{N}$, and both t_1 and t_2 are $\tau(a)$.

$$e_1 \,.\, a : \tau(a) \tag{7.32}$$
$$e_1 : \boldsymbol{N} \tag{7.33}$$
$$e_1 \,.\, a : \tau(a) \tag{7.34}$$
$$e_1 : \boldsymbol{N} \tag{7.35}$$
$$\implies \tau(a) = \tau(a) \tag{7.36}$$

The goal is now reflexive.

All subcases are shown, so Theorem 7 is proven. □

6.4.2 Proof for the Extended Calculus

Theorem 8 presents type determinism for the extended calculus. Theorem 8 looks identical to Theorem 7, but the expression e here is from the extended calculus, so can include the new productions introduced in Section 4.1.1.

Theorem 8.

$$\forall (t_1, t_2 \in \boldsymbol{T}), (e \in \boldsymbol{E}),$$
$$e : t_1 \tag{8.1}$$
$$e : t_2 \tag{8.2}$$
$$\implies t_2 = t_1 \tag{8.3}$$

Proof. Recall the type rules for extended Saiga (including both parameterisation and caching) from Section 4.2.2. The only new rule is TypeCache, and TypeAttr has been updated.

As in Theorem 7, we proceed by structural induction on e, generalising t_1 and t_2. Value, conditional, and function application expressions are typed the same way as in the core calculus, and their proofs for this theorem are the same as in Theorem 7, so will be omitted here.

Case 8.1 (e is an attribution expression)**.**

$$e_1 \,.\, a_1 (e_2) : t_1 \tag{8.4}$$
$$e_1 \,.\, a_1 (e_2) : t_2 \tag{8.5}$$
$$\implies t_2 = t_1 \tag{8.6}$$

Both (8.4) and (8.5) can only be derived from TypeAttr, which tells us that both t_1 and t_2 are equal to $\tau(a_1)$. The goal is therefore reflexive.

Case 8.2 (e is a cache expression)**.**

$$n_1 . a_1 (v_1) := e_1 : t_1 \tag{8.7}$$
$$n_1 . a_1 (v_1) := e_1 : t_2 \tag{8.8}$$
$$\implies t_2 = t_1 \tag{8.9}$$

Both (8.7) and (8.8) can only be derived from TypeCache, which tells us that both t_1 and t_2 are equal to $\tau(a_1)$. The goal is therefore reflexive.

The cases for attribution and cache expressions have been proven here, and all other expression forms have been covered in Theorem 7. Therefore Theorem 8 is proven. □

6.4.3 Proof for the Higher Order Calculus

Theorem 9 presents type determinism for the higher order calculus. Theorem 9 looks identical to Theorems 7 and 8, but the expression e here is from the higher order calculus, so can include the new productions introduced in Section 5.1.1.

Theorem 9.

$$\forall (t_1, t_2 \in \boldsymbol{T}), (e \in \boldsymbol{E}),$$
$$e : t_1 \tag{9.1}$$
$$e : t_2 \tag{9.2}$$
$$\implies t_2 = t_1 \tag{9.3}$$

Proof. Recall the type inference rules for higher order Saiga from Section 5.1.2. The only new rule is TypeCstr. As in Theorems 7 and 8, we proceed by structural induction on e. Value, conditional, function application, attribution, and caching expressions are typed the same way as in the extended calculus, and their proofs here are the same as in Theorem 8, so will not be covered again here. This leaves only the case for `MK` expressions.

Case 9.1 (e is a `MK` expression)**.**

$$\texttt{MK}\lambda n, f_l : \boldsymbol{N} \tag{9.4}$$
$$\texttt{MK}\lambda n, f_l : t_2 \tag{9.5}$$
$$\implies t_2 = \boldsymbol{N} \tag{9.6}$$

The only possible derivation of (9.5) is TypeCstr, which tells us that $t_2 = \boldsymbol{N}$, which is exactly our goal.

Theorem 9 is now proven, and Saiga's type inference rules are now proven deterministic in the core, extended, and higher order calculi. □

6.5 Type Preservation over the Step Relation

Here we want to prove that the "output" (right hand side) expression of a step relation is always the same type as the "input' (left hand side) expression. In other words, if e_1 steps into e_2, and e_1 has the type t, then e_2 also has the type t. For this property to hold, the context

function under which evaluation occurs must be *type safe*. Type safety for context functions is defined below.

Definition 6.5.1. A context function σ is type safe iff for any set of inputs n_1 and a_1, (and v_1 in the extended calculus), it produces an expression of type $\tau(a_1)$. We use this definition for context functions from both the core and extended calculi, where the parameter variable v_1 is ignored in the core case.

$$\begin{aligned}&\forall(n_1 \in \boldsymbol{N}), (a_1 \in \boldsymbol{A}), (v_1 \in \rho(a)),\\ &\sigma(n_1, a_1, v_1) : \tau(a_1)\end{aligned}$$

We will prove type preservation first with the core calculus in Theorem 10, and then with the extended and higher order calculi in Theorems 11 and 12.

6.5.1 Proof for the Core Calculus

Theorem 10.

$$\begin{aligned}&\forall(\sigma \in \boldsymbol{N} \to \boldsymbol{A} \to \boldsymbol{E}), (e_1, e_2 \in \boldsymbol{E}), (t_1 \in \boldsymbol{T}),\\ &\quad \sigma \text{ is type safe} && (10.1)\\ &\quad e_1 : t_1 && (10.2)\\ &\quad e_1 \longrightarrow e_2 && (10.3)\\ &\implies e_2 : t_1 && (10.4)\end{aligned}$$

Proof. We will proceed by induction on the derivation of the step relation (10.3), generalising t_1. The inductive hypothesis is that Theorem 10 holds for any substeps required by the step relation (10.3). In each case, the derivation requires e to take some particular form, and often there are related steps that are required for the derivation. The proof state after rewriting such terms and including new predicates will be shown for each subcase below. Further, for each case below there is only one possible derivation of (10.2). This derivation will be expanded before showing the proof state for each case. Type safety (10.1) will be ignored in all cases except Case 10.7, which is the only case that needs this result.

Case 10.1 (CondStep)**.**

$$\begin{aligned}&\texttt{IF } e_{1a} \texttt{ THEN } e_{1b} \texttt{ ELSE } e_{1c} : t_1 && (10.5)\\ &e_{1a} : \mathit{boolean} && (10.6)\\ &e_{1b} : t_1 && (10.7)\\ &e_{1c} : t_1 && (10.8)\\ &\texttt{IF } e_{1a} \texttt{ THEN } e_{1b} \texttt{ ELSE } e_{1c} \longrightarrow \texttt{IF } e'_{1a} \texttt{ THEN } e_{1b} \texttt{ ELSE } e_{1c} && (10.9)\\ &e_{1a} \longrightarrow e'_{1a} && (10.10)\\ \implies &\texttt{IF } e'_{1a} \texttt{ THEN } e_{1b} \texttt{ ELSE } e_{1c} : t_1 && (10.11)\end{aligned}$$

The inductive hypothesis, with (10.10) and (10.6) tells us that $e'_{1a} : \mathit{boolean}$. This fact, along with (10.7) and (10.8), are the prerequisites needed for TypeCond to solve the goal.

Case 10.2 (CondTrue and CondFalse)**.** In the CondTrue case we have the following:

$$\texttt{IF } [\![true]\!] \texttt{ THEN } e_{1b} \texttt{ ELSE } e_{1c} : t_1 \tag{10.12}$$
$$[\![true]\!] : \text{boolean} \tag{10.13}$$
$$e_{1b} : t_1 \tag{10.14}$$
$$e_{1c} : t_1 \tag{10.15}$$
$$\texttt{IF } [\![true]\!] \texttt{ THEN } e_{1b} \texttt{ ELSE } e_{1c} \longrightarrow e_{1b} \tag{10.16}$$
$$\Longrightarrow e_{1b} : t_1 \tag{10.17}$$

The target here is given immediately by (10.14). The same logic is used to solve the CondFalse case.

Case 10.3 (FunApp)**.**

$$[\![v_1]\!]([\![v_2]\!]) : t_1 \tag{10.18}$$
$$[\![v_1]\!] : t_2 \to t_1 \tag{10.19}$$
$$[\![v_2]\!] : t_2 \tag{10.20}$$
$$[\![v_1]\!]([\![v_2]\!]) \longrightarrow [\![v_1(v_2)]\!] \tag{10.21}$$
$$\Longrightarrow [\![v_1(v_2)]\!] : t_1 \tag{10.22}$$

We can immediately solve the goal using TypeFun with (10.19) and (10.20).

Case 10.4 (FunStep)**.**

$$e_{1a}(e_{1b}) : t_1 \tag{10.23}$$
$$e_{1a} : t_2 \to t_1 \tag{10.24}$$
$$e_{1b} : t_2 \tag{10.25}$$
$$e_{1a}(e_{1b}) \longrightarrow e'_{1a}(e_{1b}) \tag{10.26}$$
$$e_{1a} \longrightarrow e'_{1a} \tag{10.27}$$
$$\Longrightarrow e'_{1a}(e_{1b}) : t_1 \tag{10.28}$$

To derive the target using TypeFun, we need to know $e'_{1a} : t_2 \to t_1$, which can be obtained by applying the inductive hypothesis to (10.24) and (10.27). We also need to know $e_{1b} : t_2$, which is given by (10.25).

Case 10.5 (FunParStep)**.**

$$[\![v_1]\!](e_{1a}) : t_1 \tag{10.29}$$
$$[\![v_1]\!] : t_2 \to t_1 \tag{10.30}$$
$$e_{1a} : t_2 \tag{10.31}$$
$$[\![v_1]\!](e_{1a}) \longrightarrow v_1(e'_{1a}) \tag{10.32}$$
$$e_{1a} \longrightarrow e'_{1a} \tag{10.33}$$
$$\Longrightarrow [\![v_1]\!](e'_{1a}) : t_1 \tag{10.34}$$

To derive the target using TypeFun, we need to know (10.30) and $e'_{1a} : t_2$, which can be obtained by applying the inductive hypothesis to (10.31) and (10.33).

Case 10.6 (AttrNodeStep)**.**

$$e_{1a}.a : \tau(a) \tag{10.35}$$
$$e_{1a} : \boldsymbol{N} \tag{10.36}$$
$$e_{1a}.a \longrightarrow e'_{1a}.a \tag{10.37}$$
$$e_{1a} \longrightarrow e'_{1a} \tag{10.38}$$
$$\Longrightarrow e'_{1a}.a : \tau(a) \tag{10.39}$$

To derive the target using TypeAttr, we need only to know $e'_{1a} : \boldsymbol{N}$, which can be obtained by applying the inductive hypothesis to (10.36) and (10.38).

Case 10.7 (AttrFetch)**.**

$$\sigma \text{ is type safe} \tag{10.40}$$
$$[\![n_1]\!].a : \tau(a) \tag{10.41}$$
$$[\![n_1]\!].a \longrightarrow \sigma(n,a) \tag{10.42}$$
$$\Longrightarrow \sigma(n,a) : \tau(a) \tag{10.43}$$

We know that σ is type safe from (10.40). The definition of type safety is exactly the goal here. If σ was not type safe, then Theorem 7 would not hold, as it would be possible for evaluation to step into a new expression with a different type via the context function.

Each case has been satisfied, so Theorem 7 is proven. □

6.5.2 Proof for the Extended Calculus

Theorem 11 is similar to Theorem 10, but expressed for the extended calculus.

Theorem 11.

$$\forall(\sigma_1,\sigma_2 \in \boldsymbol{N} \to \boldsymbol{A} \to \boldsymbol{E}),(e_1,e_2 \in \boldsymbol{E}),(t_1 \in \boldsymbol{T}),$$
$$\sigma_1 \text{ is type safe} \tag{11.1}$$
$$e_1 : t_1 \tag{11.2}$$
$$\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2 \tag{11.3}$$
$$\Longrightarrow e_2 : t_1 \tag{11.4}$$

Proof. As in the proof for Theorem 10, we proceed by induction on the derivation of the step relation (11.3), generalizing t_1, and extract what information can be extracted from the produced terms. The result of this initial analysis will be shown as a starting point for each case below.

The proofs for the cases CondStep, CondTrue, CondFalse, FunStep, FunParStep, FunApp, and AttrNodeStep are similar enough from the same cases in the proof for Theorem 10 that they will be omitted in this section. The remaining cases starting from AttrParStep are shown below.

Case 11.1 (AttrParStep)**.**

$$\llbracket n_1 \rrbracket . a_1 (e_{1a}) : \tau(a_1) \tag{11.5}$$
$$\llbracket n_1 \rrbracket : \boldsymbol{N} \tag{11.6}$$
$$e_{1a} : \rho(a_1) \tag{11.7}$$
$$\sigma_1 \vdash \llbracket n_1 \rrbracket . a_1 (e_{1a}) \longrightarrow \sigma_2 \vdash \llbracket n_1 \rrbracket . a_1 (e'_{1a}) \tag{11.8}$$
$$\sigma_1 \vdash e_{1a} \longrightarrow \sigma_2 \vdash e'_{1a} \tag{11.9}$$
$$\Longrightarrow \llbracket n_1 \rrbracket . a_1 (e'_{1a}) : \tau(a_1) \tag{11.10}$$

To derive the goal using TypeAttr, we need (11.6) and $e'_{1a} : \rho(a_1)$, which can be obtained by applying the inductive hypothesis to (11.7) and (11.9).

Case 11.2 (AttrFetchValue and AttrFetchCached)**.** In the AttrFetchValue case we have the following:

$$\sigma_1 \text{ is type safe} \tag{11.11}$$
$$\llbracket n_1 \rrbracket . a_1 (\llbracket v_1 \rrbracket) : \tau(a_1) \tag{11.12}$$
$$\llbracket v_1 \rrbracket : \rho(a_1) \tag{11.13}$$
$$\llbracket n_1 \rrbracket : \boldsymbol{N} \tag{11.14}$$
$$\sigma_1 \vdash n_1 . a_1 (v_1) \longrightarrow \sigma_1 \vdash \llbracket v_2 \rrbracket \tag{11.15}$$
$$\sigma_1 (n_1, a_1, v_1) = \llbracket v_2 \rrbracket \tag{11.16}$$
$$\Longrightarrow \llbracket v_2 \rrbracket : \tau(a_1) \tag{11.17}$$

The target here is given by the definition of type safety, which we have from (11.11), after rewriting (11.16). The AttrFetchCached case is also solved via type safety, along with TypeCache.

Case 11.3 (CacheStep)**.**

$$n_1 . a_1 (v_1) := e_{1a} : \tau(a_1) \tag{11.18}$$
$$e_{1a} : \tau(a_1) \tag{11.19}$$
$$\sigma_1 \vdash n_1 . a_1 (v_1) := e_{1a} \longrightarrow \sigma_2 \vdash n_1 . a_1 (v_1) := e'_{1a} \tag{11.20}$$
$$\sigma_1 \vdash e_{1a} \longrightarrow \sigma_2 \vdash e'_{1a} \tag{11.21}$$
$$\Longrightarrow n_1 . a_1 (v_1) := e'_{1a} : \tau(a_1) \tag{11.22}$$

To derive the goal using TypeCache, we need $e'_{1a} : \tau(a_1)$, which can be obtained by applying the inductive hypothesis to (11.19) and (11.21).

Case 11.4 (CacheWrite)**.**

$$n_1 . a_1 (v_1) := \llbracket v_2 \rrbracket : \tau(a_1) \tag{11.23}$$
$$\llbracket v_2 \rrbracket : \tau(a_1) \tag{11.24}$$
$$\sigma_1 \vdash n_1 . a_1 (v_1) := \llbracket v_2 \rrbracket \longrightarrow \sigma_2 \vdash \llbracket v_2 \rrbracket \tag{11.25}$$
$$\Longrightarrow \llbracket v_2 \rrbracket : \tau(a_1) \tag{11.26}$$

In this case the goal is known immediately from (11.24).

All cases have been satisfied, so Theorem 8 is proven. □

6.5.3 Proof for the Higher Order Calculus

Theorem 12 appears identical to Theorem 11, but is framed in the higher order calculus, so expressions and type rules include the productions introduced in Chapter 5.

Theorem 12.

$$\begin{aligned}
&\forall(\sigma_1, \sigma_2 \in \boldsymbol{N} \to \boldsymbol{A} \to \boldsymbol{E}), (e_1, e_2 \in \boldsymbol{E}), (t_1 \in \boldsymbol{T}), \\
&\quad \sigma_1 \text{ is type safe} && (12.1)\\
&\quad e_1 : t_1 && (12.2)\\
&\quad \sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2 && (12.3)\\
&\implies e_2 : t_1 && (12.4)
\end{aligned}$$

Proof. As for Theorems 10 and 11, we will proceed by induction on the derivation of the step relation (12.3) generalising t_1, and extract what information can be extracted from the produced terms. The result of this initial analysis will be shown as a starting point for each case below.

The proofs for all cases except AttrFetchCached and AttrFetchHO are identical to the proofs presented in Theorem 11, so will be omitted here. The proof for the AttrFetchCached case is also functionally the same as the proof for AttrFetchCached shown in Theorem 11, so will also be omitted. Here we consider the only new case AttrFetchHO.

Case 12.1 (AttrFetchHO)**.**

$$\begin{aligned}
&\sigma_1 \text{ is type safe} && (12.5)\\
&[\![n_1]\!].a_1([\![v_1]\!]) : \tau(a_1) && (12.6)\\
&[\![v_1]\!] : \rho(a_1) && (12.7)\\
&[\![n_1]\!] : \boldsymbol{N} && (12.8)\\
&\sigma_1 \vdash n_1.a_1(v_1) \longrightarrow \sigma_1 \oplus \{(n_1, a_1, v_1) \mapsto [\![n_2]\!]\} \otimes n_2/f_l(n_2) \vdash [\![n_2]\!] && (12.9)\\
&\sigma_1(n_1, a_1, v_1) = \mathtt{MK}f_l && (12.10)\\
&n_2 \text{ does not exist in } \sigma_1 && (12.11)\\
&\sigma_1(n_1, a_1, v_1) : \tau(a_1) && (12.12)\\
&\mathtt{MK}f_l : \tau(a_1) && (12.13)\\
\implies &[\![n_2]\!] : \tau(a_1) && (12.14)
\end{aligned}$$

We produce (12.12) from (12.5), and we rewrite (12.10) in (12.12) to create (12.13). The only possible derivation of (12.13) is TypeCstr, which means that $\tau(a) = \boldsymbol{N}$. Rewriting this fact in the goal gives us $[\![n_2]\!] : \boldsymbol{N}$, which is derived trivially from TypeVal.

Theorem 12 is now proven, and type preservation over the step relation has been proven in the core, extended, and higher order calculi. □

6.6 Progress

Here we prove that for any well-typed expression, either that expression can step into another expression, or it is a value expression. For the extended case, we say for any pair of context function and expression, either the expression is a value expression or the pair can step to some other context/expression pair. For the higher order case, we say that either the context and expression can step, the expression is a value expression, or the expression contains a `MK` expression.

6.6.1 Proof for the Core Calculus

For the core calculus, we assert that for any well-typed expression e_1, either e_1 is a value, or there exists some e_2 such that $e_1 \longrightarrow e_2$. When we say "e_1 is a value", we mean there is some v such that $e_1 = [\![v]\!]$.

Theorem 13.

$$\forall (e_1 \in \boldsymbol{E}), (t_1 \in \boldsymbol{T}),$$
$$e_1 : t_1 \tag{13.1}$$
$$\implies (\exists e_2, e_1 \longrightarrow e_2) \vee (e_1 \text{ is a value}) \tag{13.2}$$

Proof. We proceed by structural induction on e_1, generalising t_1. This produces four cases for the four production rules for expressions: value, conditional, function application, and attribution.

Case 13.1 (e_1 is a value expression)**.** If e_1 is a value expression, the right branch is trivially true. All following cases will consider the left branch.

Case 13.2 (e_1 is a conditional expression)**.**

$$\texttt{IF } e_{1a} \texttt{ THEN } e_{1b} \texttt{ ELSE } e_{1c} : t_1 \tag{13.3}$$
$$e_{1a} : \text{boolean} \tag{13.4}$$
$$e_{1b} : t_1 \tag{13.5}$$
$$e_{1c} : t_1 \tag{13.6}$$
$$\implies \exists e_2, \texttt{IF } e_{1a} \texttt{ THEN } e_{1b} \texttt{ ELSE } e_{1c} \longrightarrow e_2 \tag{13.7}$$

First, we can apply the inductive hypothesis to e_{1a}, which tells us that it is either a value expression or must step to another expression. If e_{1a} can step to some other expression e'_{1a}, we can solve the goal with $\texttt{IF } e'_{1a} \texttt{ THEN } e_{1b} \texttt{ ELSE } e_{1c}$, which is derived from CondStep. If e_{1a} is a value expression, (13.4) tells us it must be a boolean value. If its value is *true*, we can solve the goal with e_{1b}, derived from CondTrue. If its value is *false*, we can solve the goal with e_{1c}, derived from CondFalse.

Case 13.3 (e_1 is a function application expression)**.**

$$e_{1a}(e_{1b}) : t_{1b} \tag{13.8}$$
$$e_{1a} : t_{1a} \to t_{1b} \tag{13.9}$$
$$e_{1b} : t_{1a} \tag{13.10}$$
$$\implies \exists e_2, e_{1a}(e_{1b}) \longrightarrow e_2 \tag{13.11}$$

By induction, we know that both e_{1a} and e_{1b} are either values or step into some other expressions. If e_{1a} steps into some expression e'_{1a}, we can solve the goal with $e_{1a}(e_{1b}) \longrightarrow e'_{1a}(e_{1b})$, which is derived from FunStep. If e_{1a} is a value and e_{1b} steps into some expression e'_{1b}, we can solve the goal with $e_{1a}(e_{1b}) \longrightarrow e_{1a}(e'_{1b})$, derived from FunParStep. If both e_{1a} and e_{1b} are the values v_1 and v_2, we can solve the goal with $[\![v_1]\!]([\![v_2]\!]) \longrightarrow [\![v_1(v_2)]\!]$, derived from FunApp.

Case 13.4 (e_1 is an attribution expression)**.**

$$e_{1a}.a : \tau(a) \tag{13.12}$$
$$e_{1a} : \boldsymbol{N} \tag{13.13}$$
$$\implies \exists e_2, e_{1a}.a \longrightarrow e_2 \tag{13.14}$$

Inductively, we know that e_{1a} either is a value or steps into some other expression. If e_{1a} is a value, it must be some $n_1 \in \boldsymbol{N}$ according to (13.13), and we can solve the goal with $[\![n_1]\!].a \longrightarrow \sigma(n_1, a)$, derived from AttrFetch. If e_{1a} steps into some expression e'_{1a}, we can solve the goal with $e_{1a}.a \longrightarrow e'_{1a}.a$, derived from AttrNodeStep.

All cases have been considered, so Theorem 13 is proven. □

6.6.2 Proof for the Extended Calculus

For the extended calculus, we assert that for any context function σ_1 and well-typed expression e_1, either e_1 is a value, or there exists some σ_2 and e_2 such that $\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2$.

Theorem 14.

$$\forall(\sigma_1 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}), (e_1 \in \boldsymbol{E}),$$
$$e_1 : t_1 \tag{14.1}$$
$$\implies (\exists e_2, \sigma_2 . \sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2) \vee (e_1 \text{ is a value}) \tag{14.2}$$

Proof. As in the case for the core calculus, we proceed by structural induction on the expression e_1, generalising t_1. This time there are five cases to consider. The value, conditional, and function application expressions are unchanged, so their proofs will not be shown here. We will examine the attribution and caching cases below. In both of these cases, we prove the left branch of the disjunction in the goal.

Case 14.1 (e_1 is an attribution expression)**.**

$$e_{1a}.a_1(e_{1b}) : \tau(a_1) \tag{14.3}$$
$$e_{1a} : \boldsymbol{N} \tag{14.4}$$
$$e_{1b} : \rho(a_1) \tag{14.5}$$
$$\implies \exists e_2, \sigma_2 . \sigma_1 \vdash e_{1a}.a_1(e_{1b}) \longrightarrow \sigma_2 \vdash e_2 \tag{14.6}$$

There are two subexpressions to the attribution expression, e_{1a} and e_{1b}. Inductively, we know that each of these must either be a value expression or step to some other context/expression pair. This creates a number of subcases to consider.

- If $\sigma_1 \vdash e_{1a}$ steps to some $\sigma_2 \vdash e'_{1a}$, we can solve the goal with $\sigma_1 \vdash e_{1a}.a_1(e_{1b}) \longrightarrow \sigma_2 \vdash e'_{1a}.a_1(e_{1b})$, derived from AttrNodeStep.

- If e_{1a} is some value $[\![n_1]\!]$ but $\sigma_1 \vdash e_{1b}$ steps to some $\sigma_2 \vdash e'_{1b}$, we can solve the goal with $\sigma_1 \vdash [\![n_1]\!].a_1(e_{1b}) \longrightarrow \sigma_2 \vdash [\![n_1]\!].a_1(e'_{1b})$, derived from AttrParStep.

- If e_{1a} is some value expression $[\![n_1]\!]$ and e_{1b} is some value expression $[\![v_1]\!]$, there are two possibilities still.

  - If $\sigma_1(n_1, a_1, v_1)$ is a value expression $[\![v_2]\!]$, we can solve the goal with $\sigma_1 \vdash [\![n_1]\!].a_1([\![v_1]\!]) \longrightarrow \sigma_1 \vdash [\![v_2]\!]$, derived from AttrFetchValue.
  - If $\sigma_1(n_1, a_1, v_1)$ is *not* a value expression, we can solve the goal with $\sigma_1 \vdash [\![n_1]\!].a_1([\![v_1]\!]) \longrightarrow \sigma_1 \vdash n_1.a_1(v_1) := \sigma_1(n_1, a_1, v_1)$, derived from AttrFetchCached.

Case 14.2 (e_1 is a caching expression)**.**

$$n_1.a_1(v_1) := e_{1a} : \tau(a_1) \tag{14.7}$$
$$e_{1a} : \tau(a_1) \tag{14.8}$$
$$\implies \exists e_2, \sigma_2 . \sigma_1 \vdash n_1.a_1(v_1) := e_{1a} \longrightarrow \sigma_2 \vdash e_2 \tag{14.9}$$

We know by induction that e_{1a} is a value expression or steps to some e'_{1a}. If e_{1a} is some value expression $[\![v_2]\!]$, we can solve the goal with $\sigma_1 \vdash n_1.a_1(v_1) := [\![v_2]\!] \longrightarrow \sigma_1 \oplus \{(n_1, a_1, v_1) \mapsto [\![v_2]\!]\} \vdash [\![v_2]\!]$, derived from CacheWrite. If $\sigma_1 \vdash e_{1a}$ steps to some $\sigma_2 \vdash e'_{1a}$, we can solve the goal with $\sigma_1 \vdash n_1.a_1(v_1) := e_{1a} \longrightarrow \sigma_2 \vdash n_1.a_1(v_1) := e'_{1a}$, derived from CacheStep.

All cases have been satisfied, so Theorem 14 is proven. □

6.6.3 Proof for the Higher Order Calculus

For the higher order calculus, we assert that for any context function σ_1 and well-typed expression e_1, either e_1 is a value, e_1 contains a `MK` expression, or there exists some σ_2 and e_2 such that $\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2$. When we say "$e_1$ contains a `MK` expression", we mean that there is some f_l such that $e_1 = \texttt{MK}\, f_l$, or that this is true for some subexpression of e_1.

Theorem 15.

$$\forall(\sigma_1 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}), (e_1 \in \boldsymbol{E}),$$
$$e_1 : t_1 \tag{15.1}$$
$$\implies (\exists e_2, \sigma_2 . \sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2) \vee (e_1 \text{ is a value}) \vee (e_1 \text{ contains a } \texttt{MK} \text{ expression}) \tag{15.2}$$

Proof. Theorem 15 is similar to Theorems 13 and 14, with an extra disjunction: if e_1 contains a `MK` expression, then the expression will not step. As before, we proceed by structural induction on the expression e_1, generalising t_1. The only cases with any new proof work are the attribution and `MK` expression cases. We will not examine in detail the case for the `MK` expression, as in this case the third disjunction in the goal is immediately satisfied. We examine here only the attribution expression case.

Case 15.1 (e_1 is an attribution expression)**.**

$$e_{1a}.a_1(e_{1b}) : \tau(a_1) \tag{15.3}$$
$$e_{1a} : \boldsymbol{N} \tag{15.4}$$
$$e_{1b} : \rho(a_1) \tag{15.5}$$
$$\implies (\exists e_2, \sigma_2 . \sigma_1 \vdash e_{1a}.a_1(e_{1b}) \longrightarrow \sigma_2 \vdash e_2) \vee (e_1 \text{ is a value}) \vee (e_1 \text{ is a } \texttt{MK} \text{ expression}) \tag{15.6}$$

There are two subexpressions to the attribution expression, e_{1a} and e_{1b}. Inductively, we know that each of these must either step to some other context/expression pair, be a value expression, or contain MK expression. This creates a number of subcases to consider.

- If e_1 steps, we can satisfy the goal using the first branch, as in Theorem 14.
- If e_1 is a MK expression, we can satisfy the goal trivially with the third branch.
- If e_1 is a value expression and e_2 steps, we can prove the theorem using the first branch, as in Theorem 14.
- If e_1 is a value expression and e_2 is a MK expression, we can satisfy the goal trivially with the third branch.
- If e_1 is a value expression $[\![n_1]\!]$ and e_2 is a value expression $[\![v_1]\!]$, there are three ways the first branch can be satisfied. We are not using the inductive hypothesis on $\sigma_1(n_1, a_1, v_1)$, as this is not a subexpression of e_1. However, we consider the three following possibilities for this expression.
  - If $\sigma_1(n_1, a_1, v_1)$ is a value expression, we satisfy the goal using the first branch, derived using AttrFetchValue.
  - If $\sigma_1(n_1, a_1, v_1)$ is a MK expression, we satisfy the goal using the first branch, derived using AttrFetchHO.
  - If $\sigma_1(n_1, a_1, v_1)$ is any other expression form, we satisfy the goal using the first branch, derived using AttrFetchCached.

Theorem 15 is now proven, and progress has now been proven in the core, extended, and higher order calculi. □

6.7 Big Step Multistep Equivalence

In Sections 3.2.9, 4.3.5 and 5.3.6, we presented a big step semantics for Saiga. Here we will prove that multistep, as presented in Sections 3.2.8, 4.3.4 and 5.3.5, is equivalent to big step, in that the same expression and context function will always reach the same final value and output context function. First we prove that the multistep relation implies the big step relation, then we will prove that the big step relation implies the multistep relation.

For previous theorems, we have presented proofs separately for the core, extended, and higher order calculi. Here we will present only the higher order proof, as proof for the higher order calculus is the most difficult, and covers all cases from the other calculi.

6.7.1 Multistep Implies Big Step

To prove that the multistep relation implies the big step relation, we must first prove Lemma 16, which states that a single step followed by a big step implies a big step from the source of the single step.

Lemma 16.

$$\begin{aligned}
&\forall(e_1, e_2 \in \boldsymbol{E}), (\sigma_1, \sigma_2, \sigma_3 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}), (t \in \boldsymbol{T}), (v \in t), \\
&\quad \sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2 && (16.1)\\
&\quad \sigma_2 \vdash e_2 \twoheadrightarrow \sigma_3 \vdash v && (16.2)\\
&\implies \sigma_1 \vdash e_1 \twoheadrightarrow \sigma_3 \vdash v && (16.3)
\end{aligned}$$

Proof. We proceed by induction on the derivation of the single step relation (16.1). This provides one case for each of the 13 step rules in the higher order single step semantics presented in Section 5.1.3. For each case, we will examine the possible derivations of (16.2). For most cases there will only be one possible derivation, but we will consider each case separately when there are multiple. Usually identifying the derivation will provide some further hypotheses, which will be shown in the proof state for each following case.

Case 16.1 (CondStep)**.** The big step relation as part of this case has two subcases: BCondTrue and BCondFalse.

Case 16.1.1 (CondStep followed by BCondTrue)**.**

$$\begin{aligned}
&\sigma_1 \vdash \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \longrightarrow \sigma_2 \vdash \texttt{IF}\ e_1'\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 && (16.4)\\
&\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_1' && (16.5)\\
&\sigma_2 \vdash \texttt{IF}\ e_1'\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \twoheadrightarrow \sigma_3 \vdash v && (16.6)\\
&\sigma_2 \vdash e_1' \twoheadrightarrow \sigma_4 \vdash \mathit{true} && (16.7)\\
&\sigma_4 \vdash e_2 \twoheadrightarrow \sigma_3 \vdash v && (16.8)\\
\implies &\sigma_1 \vdash \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \twoheadrightarrow \sigma_3 \vdash v && (16.9)
\end{aligned}$$

The goal here is satisfied using BCondTrue with (16.8), and the inductive hypothesis applied to (16.5) and (16.7).

Case 16.1.2 (BCondFalse)**.** This case is proved in the same manner as in case 16.1.1.

Case 16.2 (CondStep followed by CondTrue)**.**

$$\begin{aligned}
&\sigma_1 \vdash \texttt{IF}\ [\![\mathit{true}]\!]\ \texttt{THEN}\ e_1\ \texttt{ELSE}\ e_2 \longrightarrow \sigma_1 \vdash e_1 && (16.10)\\
&\sigma_1 \vdash e_1 \twoheadrightarrow \sigma_2 \vdash v && (16.11)\\
\implies &\sigma_1 \vdash \texttt{IF}\ [\![\mathit{true}]\!]\ \texttt{THEN}\ e_1\ \texttt{ELSE}\ e_2 \twoheadrightarrow \sigma_2 \vdash v && (16.12)
\end{aligned}$$

This goal is given from (16.11) and from BRefl on $[\![\mathit{true}]\!]$.

Case 16.3 (CondFalse)**.** This case is proved in the same manner as in Case 16.2.

Case 16.4 (FunStep)**.** The big step relation here can only be derived from BFun.

$$\sigma_1 \vdash e_1(e_2) \longrightarrow \sigma_2 \vdash e_1'(e_2) \tag{16.13}$$
$$\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_1' \tag{16.14}$$
$$\sigma_2 \vdash e_1'(e_2) \twoheadrightarrow \sigma_3 \vdash f(p) \tag{16.15}$$
$$\sigma_2 \vdash e_1' \twoheadrightarrow \sigma_4 \vdash f \tag{16.16}$$
$$\sigma_4 \vdash e_2 \twoheadrightarrow \sigma_3 \vdash p \tag{16.17}$$
$$\Longrightarrow \sigma_1 \vdash e_1(e_2) \twoheadrightarrow \sigma_3 \vdash f(p) \tag{16.18}$$

The goal here can be satisfied with BFun, using (16.17) and the inductive hypothesis applied to (16.14) and (16.16).

Case 16.5 (FunParStep)**.** The big step relation here can also only be derived from BFun, which yields (16.22) and (16.23).

$$\sigma_1 \vdash [\![f]\!](e_1) \longrightarrow \sigma_2 \vdash [\![f]\!](e_1') \tag{16.19}$$
$$\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_1' \tag{16.20}$$
$$\sigma_2 \vdash [\![f]\!](e_1') \twoheadrightarrow \sigma_3 \vdash f(p) \tag{16.21}$$
$$\sigma_2 \vdash [\![f]\!] \twoheadrightarrow \sigma_2 \vdash f \tag{16.22}$$
$$\sigma_2 \vdash e_1' \twoheadrightarrow \sigma_3 \vdash p \tag{16.23}$$
$$\Longrightarrow \sigma_1 \vdash [\![f]\!](e_1) \twoheadrightarrow \sigma_3 \vdash f(p) \tag{16.24}$$

The goal here can be satisfied with BFun, using (16.22) and the inductive hypothesis applied to (16.20) and (16.23).

Case 16.6 (FunApp)**.**

$$\sigma_1 \vdash [\![f]\!]([\![p]\!]) \longrightarrow \sigma_1 \vdash [\![f(p)]\!] \tag{16.25}$$
$$\sigma_1 \vdash [\![f(p)]\!] \twoheadrightarrow \sigma_1 \vdash f(p) \tag{16.26}$$
$$\Longrightarrow \sigma_1 \vdash [\![f]\!]([\![p]\!]) \twoheadrightarrow \sigma_1 \vdash [\![f(p)]\!] \tag{16.27}$$

The goal here is satisfied immediately by BFun, with each of its requisites provided by BRefl.

Case 16.7 (AttrNodeStep)**.** The big step relation shown by (16.30) below can be derived in three ways: BAttrValue, BAttrCached, and BAttrHO. We consider these three subcases separately.

Case 16.7.1 (AttrNodeStep followed by BAttrValue)**.**

$$\sigma_1 \vdash e_1.a(e_2) \longrightarrow \sigma_2 \vdash e_1'.a(e_2) \tag{16.28}$$
$$\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_1' \tag{16.29}$$
$$\sigma_2 \vdash e_1'.a(e_2) \twoheadrightarrow \sigma_3 \vdash v \tag{16.30}$$
$$\sigma_2 \vdash e_1' \twoheadrightarrow \sigma_4 \vdash n \tag{16.31}$$
$$\sigma_4 \vdash e_2 \twoheadrightarrow \sigma_3 \vdash p \tag{16.32}$$
$$\sigma_3(n,a,p) = [\![v]\!] \tag{16.33}$$
$$\Longrightarrow \sigma_1 \vdash e_1.a(e_2) \twoheadrightarrow \sigma_3 \vdash v \tag{16.34}$$

Here we can satisfy the goal using BAttrValue. The first requisite can be provided by the inductive hypothesis applied to (16.29) and (16.31). The other requisites to BAttrValue are provided by (16.32) and (16.33).

Case 16.7.2 (AttrNodeStep followed by BAttrCached)**.**

$$\sigma_1 \vdash e_1.a(e_2) \longrightarrow \sigma_2 \vdash e_1'.a(e_2) \tag{16.35}$$
$$\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_1' \tag{16.36}$$
$$\sigma_2 \vdash e_1'.a(e_2) \twoheadrightarrow \sigma_6 \oplus \{(n,a,p) \mapsto [\![v]\!]\} \vdash v \tag{16.37}$$
$$\sigma_2 \vdash e_1' \twoheadrightarrow \sigma_4 \vdash n \tag{16.38}$$
$$\sigma_4 \vdash e_2 \twoheadrightarrow \sigma_5 \vdash p \tag{16.39}$$
$$\sigma_5 \vdash \sigma_5(n,a,p) \twoheadrightarrow \sigma_6 \vdash v \tag{16.40}$$
$$\sigma_5(n,a,p) \text{ is not a value or } \texttt{MK} \text{ expression} \tag{16.41}$$
$$\Longrightarrow \sigma_1 \vdash e_1.a(e_2) \twoheadrightarrow \sigma_6 \oplus \{(n,a,p) \mapsto [\![v]\!]\} \vdash v \tag{16.42}$$

Here we can satisfy the goal using BAttrCached. The first requisite can be provided by the inductive hypothesis applied to (16.36) and (16.38). The other requisites to BAttrCached are provided by (16.39) and (16.41).

Case 16.7.3 (AttrNodeStep followed by BAttrHO)**.**

$$\sigma_1 \vdash e_1.a(e_2) \longrightarrow \sigma_2 \vdash e_1'.a(e_2) \tag{16.43}$$
$$\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_1' \tag{16.44}$$
$$\sigma_2 \vdash e_1'.a(e_2) \twoheadrightarrow \sigma_4 \oplus \{(n,a,p) \mapsto [\![n_2]\!]\} \otimes n_2/f_l(n_2) \vdash n_2 \tag{16.45}$$
$$\sigma_2 \vdash e_1' \twoheadrightarrow \sigma_3 \vdash n \tag{16.46}$$
$$\sigma_3 \vdash e_2 \twoheadrightarrow \sigma_4 \vdash p \tag{16.47}$$
$$\sigma_4(n,a,p) = \texttt{MK}f_l \tag{16.48}$$
$$n_2 \text{ does not exist in } \sigma_4 \tag{16.49}$$
$$\Longrightarrow \sigma_1 \vdash e_1.a(e_2) \twoheadrightarrow \sigma_4 \oplus \{(n,a,p) \mapsto [\![n_2]\!]\} \otimes n_2/f_l(n_2)v \vdash n_2 \tag{16.50}$$

Here we can satisfy the goal using BAttrHO. The first requisite can be provided by the inductive hypothesis applied to (16.44) and (16.46). The other requisites to BAttrHO are provided by (16.47), (16.48), and (16.49).

Case 16.8 (AttrParStep)**.** As in Case 16.7, the big step relation for this case can have the three derivations BAttrValue, BAttrCached, and BAttrHO. Proofs for these three cases follow the same pattern as for Case 16.7, so will not be shown here.

Case 16.9 (AttrFetchValue)**.**

$$\sigma_1 \vdash [\![n]\!].a([\![p]\!]) \longrightarrow \sigma_1 \vdash [\![v]\!] \tag{16.51}$$
$$\sigma_1(n,a,p) = [\![v]\!] \tag{16.52}$$
$$\sigma_1 \vdash [\![v]\!] \twoheadrightarrow \sigma_1 \vdash v \tag{16.53}$$
$$\Longrightarrow \sigma_1 \vdash [\![n]\!].a([\![p]\!]) \twoheadrightarrow \sigma_1 \vdash v \tag{16.54}$$

Here the goal can be satisfied immediately by BAttrValue, using (16.52).

Case 16.10 (AttrFetchCached)**.**

$$\sigma_1 \vdash [\![n]\!].a([\![p]\!]) \longrightarrow \sigma_1 \vdash n.a(p) := \sigma_1(n,a,p) \tag{16.55}$$
$$\sigma_1(n,a,p) \text{ is not a value or } \texttt{MK} \text{ expression} \tag{16.56}$$
$$\sigma_1 \vdash n.a(p) := \sigma_1(n,a,p) \twoheadrightarrow \sigma_2 \oplus \{(n,a,p) \mapsto [\![v]\!]\} \vdash v \tag{16.57}$$
$$\sigma_1 \vdash \sigma_1(n,a,p) \twoheadrightarrow \sigma_2 \vdash v \tag{16.58}$$
$$\Longrightarrow \sigma_1 \vdash [\![n]\!].a([\![p]\!]) \twoheadrightarrow \sigma_2 \oplus \{(n,a,p) \mapsto [\![v]\!]\} \vdash v \tag{16.59}$$

Here the goal can be satisfied immediately by BAttrCached, using (16.58) and (16.56)

Case 16.11 (AttrFetchHO)**.**

$$\sigma_1 \vdash [\![n]\!].a([\![p]\!]) \longrightarrow \sigma_1 \oplus \{(n,a,p) \mapsto [\![n_2]\!]\} \otimes n_2/f_l \vdash [\![n_2]\!] \tag{16.60}$$
$$\sigma_1(n,a,p) = \texttt{MK}f_l \tag{16.61}$$
$$n_2 \text{ does not exist in } \sigma_1 \tag{16.62}$$
$$\sigma_1 \oplus \{(n,a,p) \mapsto [\![n_2]\!]\} \otimes n_2/f_l \vdash [\![n_2]\!] \twoheadrightarrow \sigma_1 \oplus \{(n,a,p) \mapsto [\![n_2]\!]\} \otimes n_2/f_l \vdash n_2 \tag{16.63}$$
$$\Longrightarrow \sigma_1 \vdash [\![n]\!].a([\![p]\!]) \twoheadrightarrow \sigma_1 \oplus \{(n,a,p) \mapsto [\![n_2]\!]\} \otimes n_2/f_l \vdash n_2 \tag{16.64}$$

Here the goal can be satisfied immediately by BAttrHO, using (16.61) and (16.62).

Case 16.12 (CacheStep)**.**

$$\sigma_1 \vdash n.a(p) := e \longrightarrow \sigma_2 \vdash n.a(p) := e' \tag{16.65}$$
$$\sigma_1 \vdash e \longrightarrow \sigma_2 \vdash e' \tag{16.66}$$
$$\sigma_2 \vdash n.a(p) := e' \twoheadrightarrow \sigma_3 \oplus \{(n,a,p) \mapsto [\![v]\!]\} \vdash v \tag{16.67}$$
$$\sigma_2 \vdash e' \twoheadrightarrow \sigma_3 \vdash v \tag{16.68}$$
$$\Longrightarrow \sigma_1 \vdash n.a(p) := e \twoheadrightarrow \sigma_3 \oplus \{(n,a,p) \mapsto [\![v]\!]\} \vdash v \tag{16.69}$$

Here the goal can be satisfied by BCache, and with the inductive hypothesis applied to (16.66) and (16.68).

Case 16.13 (CacheWrite)**.**

$$\sigma_1 \vdash n.a(p) := [\![v]\!] \longrightarrow \sigma_1 \oplus \{(n,a,p) \mapsto [\![v]\!]\} \vdash [\![v]\!] \tag{16.70}$$
$$\sigma_1 \oplus \{(n,a,p) \mapsto [\![v]\!]\} \vdash [\![v]\!] \twoheadrightarrow \sigma_1 \oplus \{(n,a,p) \mapsto [\![v]\!]\} \vdash v \tag{16.71}$$
$$\Longrightarrow \sigma_1 \vdash n.a(p) := [\![v]\!] \twoheadrightarrow \sigma_1 \oplus \{(n,a,p) \mapsto [\![v]\!]\} \vdash v \tag{16.72}$$

Here the goal can be satisfied immediately by BCache.

Every case has been satisfied, so Lemma 16 is proven. □

Now that we have proven Lemma 16, we can prove that the multistep relation implies the big step relation, in Theorem 17.

Theorem 17.

$$\forall(\sigma_1,\sigma_2 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}), (e \in \boldsymbol{E}), (t \in \boldsymbol{T}), (v \in t),$$
$$\sigma_1 \vdash e \longrightarrow^* \sigma_2 \vdash [\![v]\!] \tag{17.1}$$
$$\Longrightarrow \sigma_1 \vdash e \twoheadrightarrow \sigma_2 \vdash v \tag{17.2}$$

Proof. We proceed by structural induction on the derivation of the multistep relation (17.1). This produces two cases: one for the reflection case MultiRefl, and one for the step case MultiStep.

Case 17.1 (MultiRefl)**.**

$$\sigma_1 \vdash [\![v]\!] \longrightarrow^* \sigma_1 \vdash [\![v]\!] \tag{17.3}$$
$$\implies \sigma_1 \vdash [\![v]\!] \twoheadrightarrow \sigma_1 \vdash v \tag{17.4}$$

Here the goal is satisfied immediately with BRefl.

Case 17.2 (MultiStep)**.**

$$\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_2 \tag{17.5}$$
$$\sigma_2 \vdash e_2 \longrightarrow^* \sigma_3 \vdash [\![v]\!] \tag{17.6}$$
$$\sigma_2 \vdash e_2 \twoheadrightarrow \sigma_3 \vdash v \tag{17.7}$$
$$\implies \sigma_1 \vdash e_1 \twoheadrightarrow \sigma_3 \vdash v \tag{17.8}$$

The big step relation (17.7) is given by induction. Lemma 16 solves this goal, using (17.5) and (17.7).

Both cases are satisfied, so Theorem 17 is proven. □

6.7.2 Big Step Implies Multistep

To prove that the big step relation implies the multistep relation, we must first prove six related lemmas (Lemmas 18 to 23). These lemmas are similar enough to each other in their structure and proof strategy that we will only show one of their proofs here.

Lemma 18.

$$\forall(\sigma_1,\sigma_2,\sigma_3 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),$$
$$\forall(e_1,e_2,e_3 \in \boldsymbol{E}),(t_1,t_2 \in \boldsymbol{T}),(v_1 \in t_1),(v_2 \in t_2),$$
$$\sigma_1 \vdash e_1 \longrightarrow^* \sigma_2 \vdash [\![v_1]\!] \tag{18.1}$$
$$\sigma_2 \vdash \texttt{IF}\ [\![v_1]\!]\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \longrightarrow^* \sigma_3 \vdash [\![v_2]\!] \tag{18.2}$$
$$\implies \sigma_1 \vdash \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \longrightarrow^* \sigma_3 \vdash [\![v_2]\!] \tag{18.3}$$

Proof. We proceed by structural induction on the derivation of the multistep relation (18.1), generalising σ_3 and $[\![v_2]\!]$. This produces two cases, one for the MultiRefl derivation and one for the MultiStep derivation of the multistep relation. For the MultiStep case, we have the inductive hypothesis that this lemma holds for the nested multistep relation.

Case 18.1 (MultiRefl)**.**

$$\sigma_1 \vdash [\![v_1]\!] \longrightarrow^* \sigma_1 \vdash [\![v_1]\!] \tag{18.4}$$
$$\sigma_1 \vdash \texttt{IF}\ [\![v_1]\!]\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \longrightarrow^* \sigma_3 \vdash [\![v_2]\!] \tag{18.5}$$
$$\implies \sigma_1 \vdash \texttt{IF}\ [\![v_1]\!]\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \longrightarrow^* \sigma_3 \vdash [\![v_2]\!] \tag{18.6}$$

Here the goal is satisfied immediately by (18.5).

Case 18.2 (MultiStep)**.**

$$\sigma_1 \vdash e_1 \longrightarrow \sigma_2 \vdash e_1' \tag{18.7}$$
$$\sigma_2 \vdash e_1' \longrightarrow^* \sigma_3 \vdash [\![v_1]\!] \tag{18.8}$$
$$\sigma_3 \vdash \texttt{IF } [\![v_1]\!] \texttt{ THEN } e_2 \texttt{ ELSE } e_3 \longrightarrow^* \sigma_4 \vdash [\![v_2]\!] \tag{18.9}$$
$$\sigma_2 \vdash \texttt{IF } e_1' \texttt{ THEN } e_2 \texttt{ ELSE } e_3 \longrightarrow^* \sigma_4 \vdash [\![v_2]\!] \tag{18.10}$$
$$\implies \sigma_1 \vdash \texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3 \longrightarrow^* \sigma_4 \vdash [\![v_2]\!] \tag{18.11}$$

(18.10) is provided by the inductive hypothesis applied to (18.9). The goal is satisfied by MultiStep, with (18.7) and (18.10).

Lemma 18 is proven. □

Lemma 19.

$$\forall(\sigma_1,\sigma_2,\sigma_3 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),$$
$$\forall(e_1,e_2 \in \boldsymbol{E}),(t_1,t_2 \in \boldsymbol{T}),(v_1 \in t_1),(v_2 \in t_2),$$
$$\sigma_1 \vdash e_1 \longrightarrow^* \sigma_2 \vdash [\![v_1]\!] \tag{19.1}$$
$$\sigma_2 \vdash [\![v_1]\!](e_2) \longrightarrow^* \sigma_3 \vdash [\![v_2]\!] \tag{19.2}$$
$$\implies \sigma_1 \vdash e_1(e_2) \longrightarrow^* \sigma_3 \vdash [\![v_2]\!] \tag{19.3}$$

Proof. This lemma is proven using the same strategy employed to prove Lemma 18. □

Lemma 20.

$$\forall(\sigma_1,\sigma_2,\sigma_3 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),$$
$$\forall(e \in \boldsymbol{E}),(t_1,t_2,t_3 \in \boldsymbol{T}),(v_1 \in t_1),(v_2 \in t_2),(v_3 \in t_3),$$
$$\sigma_1 \vdash e \longrightarrow^* \sigma_2 \vdash [\![v_2]\!] \tag{20.1}$$
$$\sigma_2 \vdash [\![v_1]\!]([\![v_2]\!]) \longrightarrow^* \sigma_3 \vdash [\![v_3]\!] \tag{20.2}$$
$$\implies \sigma_1 \vdash [\![v_1]\!](e) \longrightarrow^* \sigma_3 \vdash [\![v_3]\!] \tag{20.3}$$

Proof. This lemma is proven using the same strategy employed to prove Lemma 18. □

Lemma 21.

$$\forall(\sigma_1,\sigma_2,\sigma_3 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),$$
$$\forall(e_1,e_2 \in \boldsymbol{E}),(t_2 \in \boldsymbol{T}),(v_2 \in t_2),(n_1 \in \boldsymbol{N}),(a \in \boldsymbol{A}),$$
$$\sigma_1 \vdash e_1 \longrightarrow^* \sigma_2 \vdash [\![n_1]\!] \tag{21.1}$$
$$\sigma_2 \vdash [\![n_1]\!].a(e_2) \longrightarrow^* \sigma_3 \vdash [\![v_2]\!] \tag{21.2}$$
$$\implies \sigma_1 \vdash e_1.a(e_2) \longrightarrow^* \sigma_3 \vdash [\![v_2]\!] \tag{21.3}$$

Proof. This lemma is proven using the same strategy employed to prove Lemma 18. □

Lemma 22.

$$\begin{aligned}
&\forall(\sigma_1,\sigma_2,\sigma_3 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),\\
&\forall(e_1 \in \boldsymbol{E}),(t_1,t_2 \in \boldsymbol{T}),(v_1 \in t_1),(v_2 \in t_2),(n_1 \in \boldsymbol{N}),(a \in \boldsymbol{A}),\\
&\quad \sigma_1 \vdash e_1 \longrightarrow^* \sigma_2 \vdash [\![v_1]\!] && (22.1)\\
&\quad \sigma_2 \vdash [\![n_1]\!].a([\![v_1]\!]) \longrightarrow^* \sigma_3 \vdash [\![v_2]\!] && (22.2)\\
&\implies \sigma_1 \vdash [\![n_1]\!].a(e_1) \longrightarrow^* \sigma_3 \vdash [\![v_2]\!] && (22.3)
\end{aligned}$$

Proof. This lemma is proven using the same strategy employed to prove Lemma 18. □

Lemma 23.

$$\begin{aligned}
&\forall(\sigma_1,\sigma_2,\sigma_3 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),\\
&\forall(e_1 \in \boldsymbol{E}),(n_1 \in \boldsymbol{N}),(a \in \boldsymbol{A}),(v_1 \in \rho(a)),(t_1 \in \boldsymbol{T}),(v_1 \in t_1),\\
&\quad \sigma_1 \vdash e_1 \longrightarrow^* \sigma_2 \vdash [\![v_1]\!] && (23.1)\\
&\quad \sigma_2 \vdash n_1.a(v_1) := [\![v_1]\!] \longrightarrow^* \sigma_3 \vdash [\![v_1]\!] && (23.2)\\
&\implies \sigma_1 \vdash n_1.a(v_1) := e_1 \longrightarrow^* \sigma_3 \vdash [\![v_1]\!] && (23.3)
\end{aligned}$$

Proof. This lemma is proven using the same strategy employed to prove Lemma 18. □

Theorem 24.

$$\begin{aligned}
&\forall(\sigma_1,\sigma_2 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),(e \in \boldsymbol{E}),(t \in \boldsymbol{T}),(v \in t),\\
&\quad \sigma_1 \vdash e \twoheadrightarrow \sigma_2 \vdash v && (24.1)\\
&\implies \sigma_1 \vdash e \longrightarrow^* \sigma_2 \vdash [\![v]\!] && (24.2)
\end{aligned}$$

Proof. We proceed by induction on the derivation of the big step relation (24.1). This produces eight cases, one for each of the derivations of the big step relation. We will not examine the details of these cases, as they matched to Lemmas 18 to 23. The eight cases are satisfied as follows.

- The BRefl case is solved by MultiRefl.
- The BCondTrue case is solved by Lemma 18 applied to a multistep relation given by induction, MultiStep and CondTrue.
- The BCondFalse case is solved by Lemma 18 applied to a multistep relation given by induction, MultiStep and CondFalse.
- The BFun case is solved by Lemmas 19 and 20 applied to two multistep relations given by induction, MultiStep, and FunApp.
- The BAttrValue case is solved by Lemmas 21 and 22 applied to two multistep relations given by induction, MultiStep, AttrFetchValue, and the fact that the expression returned by the context function is a value.
- The BAttrCached case is solved by Lemmas 21 and 22 applied to two multistep relations given by induction, MultiStep, AttrFetchCached, and the fact that the expression returned by the context function is not a value.

- The BAttrHO case is solved by Lemmas 21 and 22 applied to two multistep relations given by induction, MultiStep, AttrFetchHO, and multiple facts provided by the expansion of BAttrHO.
- The BCache case is solved by Lemma 23 applied to a multistep relation given by induction, MultiStep, and CacheWrite.

Theorem 24 is proven. □

Through Theorems 17 and 24, we have proven that the multistep and big step relations given in this thesis are equivalent. We combine these results into one with Theorem 25.

Theorem 25.

$$\begin{aligned}&\forall(\sigma_1 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}), (e \in \boldsymbol{E}), (t \in \boldsymbol{T}), (v \in t),\\ &\quad \sigma_1 \vdash e \twoheadrightarrow \_ \vdash v\\ &\iff \sigma_1 \vdash e \longrightarrow^* \_ \vdash [\![v]\!]\end{aligned}$$

Proof. Theorem 25 is given by Theorems 17 and 24. □

6.7.3 Big Step Induction

Now that the big step and multistep relations are proven equivalent, we can choose between two kinds of induction for all future proofs. Consider the following multistep relation.

$$\sigma_1 \vdash e \longrightarrow^* \sigma_2 \vdash [\![v]\!]$$

If we were to perform induction on the above multistep relation, we would need to satisfy two cases based on the MultiRefl and MultiStep derivations. Now consider an equivalent big step relation.

$$\sigma_1 \vdash e \twoheadrightarrow \sigma_2 \vdash v$$

If we were to perform induction on the above big step relation, we would need to satisfy the eight derivations of the relation. We express most theorems from this point on using the multistep relation, as it is the most closely linked to the single step relation, which is the core of our calculus. However, we often say that we use "big step induction" on a multistep relation, which is equivalent to using Theorem 17 to transform the relation into a big step relation, performing induction on that big step relation, and transforming all terms and hypotheses back into multistep relations using Theorem 24. This approach can be useful as the big step semantics capture the "big picture" semantics of a total evaluation, which is sometimes a focus that makes our proof easier to frame.

6.8 Cache Irrelevance

Caching is a feature designed for efficiency; when an attribute's value is computed, it is stored for later use, so that it does not need to be recomputed. It is expected that implementing caching will never change the output of any attributes. However, it is useful to not assume this property, but to prove it. It turns out that proving *cache irrelevance* (the property described above) is quite complex in Saiga.

Cache irrelevance does not apply to the core calculus, which does not implement caching. A proof of cache irrelevance has been mechanised in Lean for the extended calculus, after considerable effort. Cache irrelevance for the higher order calculus has a partial proof mechanised in Lean, but is a more complex problem.

In this section we will not explore the low-level details of this proof, as our mechanisation lends us the confidence to explore the proof on a higher level. We start with a definition of *context equivalence* (Definition 6.8.1), a property that states that two context functions, while perhaps not equal, will always produce the same result after stepping its output expression to a value. Theorem 33 proves that evaluation only ever creates equivalent contexts, which implies that caching (and higher order construction) do not change the values that will be evaluated from a context function. This is how we phrase and prove cache irrelevance for Saiga.

For simplicity we work with the big step relation here - big step has fewer semantic rules than small step, and breaks evaluation down into useful milestone points. Also, big step is proven to be equivalent to multistep, so proving cache irrelevance for the big step semantics is equivalent to proving cache irrelevance for the small step semantics.

Definition 6.8.1 (Context Equivalence)**.** Context implication (expressed using the $\equiv\rangle$ operator) means that evaluation of any context output from the left context implies an equivalent evaluation in the right context.

$$\begin{gathered}\forall(\sigma_1,\sigma_2 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),\\ \sigma_1 \equiv\rangle\ \sigma_2\\ \iff \forall(n \in \boldsymbol{N}),(a \in \boldsymbol{A}),(p \in \rho(a)),(t \in \boldsymbol{T}),(v \in t),({\sigma_1}' \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),\\ (\sigma_1 \vdash \sigma_1(n,a,p) \twoheadrightarrow {\sigma_1}' \vdash v \implies \exists {\sigma_2}', \sigma_2 \vdash \sigma_2(n,a,p) \twoheadrightarrow {\sigma_2}' \vdash v)\end{gathered}$$

Context equivalence (expressed using the $\equiv$ operator) is the bidirectional version of context implication.

$$\begin{gathered}\forall(\sigma_1,\sigma_2 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),\\ \sigma_1 \equiv \sigma_2 \iff \sigma_1 \equiv\rangle\ \sigma_2 \wedge \sigma_2 \equiv\rangle\ \sigma_1\end{gathered}$$

Proofs of transitivity, reflexivity, and symmetry of the context equivalence relation are trivial, and will not be shown here. The primary task in proving cache irrelevance is proving that evaluations during normal evaluation will only ever create equivalent contexts; this is to say that the transformations of a context function never change the semantics of its outputs. To prove this we must work through a number of related lemmas.

Lemma 26.

$$\begin{gathered}\forall(\sigma,\sigma' \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),\\ \forall(n \in \boldsymbol{N}),(a \in \boldsymbol{A}),(p \in \rho(a)),(t \in \boldsymbol{T}),(v \in \boldsymbol{T}),(e \in \boldsymbol{E}),\\ \sigma(n,a,p) \text{ is a value} \qquad (26.1)\\ \sigma \vdash e \twoheadrightarrow \sigma' \vdash v \qquad (26.2)\\ \implies \sigma'(n,a,p) = \sigma(n,a,p) \qquad (26.3)\end{gathered}$$

Proof. This theorem states that once a context function returns a value expression, evaluation will never change that particular output. This is observable by examining the rules that can change the context function: AttrFetchCached, AttrFetchHO, and CacheWrite. AttrFetchCached and AttrFetchHO will only occur when the context function returns a non-value expression, so these will never overwrite a value expression in the context. CacheWrite is not a user-level expression, and is only ever created by requesting an attribute, which creates a caching expression containing the expression taken directly from the context function – in this case, a value. Therefore the value may be written 'back' into the context function, but this will not change the context function at all. □

Lemma 27.

$$\begin{aligned} &\forall(\sigma \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}), \\ &\forall(n \in \boldsymbol{N}), (a \in \boldsymbol{A}), (p \in \rho(a)), (t \in \boldsymbol{T}), (v \in \boldsymbol{T}), \\ &\quad \sigma \vdash \sigma(n,a,p) \twoheadrightarrow \sigma \vdash v && (27.1) \\ &\implies \sigma(n,a,p) \text{ only requests valued attributes} && (27.2) \end{aligned}$$

Proof. This lemma states that if an evaluation terminates without changing the context function, any attributes requested during this evaluation are valued. We use the term *valued attribute* to refer to an attribute that a particular context function will return a value expression for. We can observe this property intuitively: if evaluation requests an attribute that is higher order, then that attribute will be immediately overwritten by its new node label, therefore changing the context (by replacing a higher-order expression with a node value expression). If evaluation requests a non-value non-higher order attribute, it will be evaluated and a value will be written in its place to the context function, therefore also changing the context (by replacing a non-value expression with a value expression). Therefore if the context has not changed, then only valued attributes have been requested. □

Lemma 28.

$$\begin{aligned} &\forall(\sigma, \sigma' \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}), \\ &\quad \forall(e \in \boldsymbol{E}), (t \in \boldsymbol{T}), (v \in t), \\ &\quad \sigma \vdash e \twoheadrightarrow \sigma' \vdash v && (28.1) \\ &\implies \sigma' \vdash e \twoheadrightarrow \sigma' \vdash v && (28.2) \end{aligned}$$

Proof. This lemma states that once some expression e has been evaluated, producing some new context, evaluating e under the new context will result in the same value, and an unchanged context. The intuition behind this proof is simple: any attributes that are requested during the evaluation of e (including any that are called recursively from a called attribute) will be cached into the new context σ'. Evaluating e under σ' will result in only calling attributes that are cached and therefore valued attributes. Since calling values attributes will not change a context function, the output context of the second evaluation will be the same as the input. □

Lemma 29.

$$\begin{aligned}
&\forall(\sigma,\sigma' \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),\\
&\forall(n \in \boldsymbol{N}),(a \in \boldsymbol{A}),(p \in \rho(a)),(t \in \boldsymbol{T}),(v \in t),\\
&\quad \sigma \vdash \sigma(n,a,p) \twoheadrightarrow \sigma' \vdash v && (29.1)\\
&\implies \sigma'(n,a,p) = \sigma(n,a,p) && (29.2)
\end{aligned}$$

Proof. This lemma states that if the returned expression from a context function is evaluated to a value, the resulting context function will not have changed its output for the attribute in question. The intuition behind the proof of this lemma comes in two parts. Firstly, we assert that the evaluation (29.1) will never request the attribute (n,a,p). We know this because if it did, an infinite loop would occur, and the relation would not be possible, as a value would never be reached. The fact that (29.1) is a terminating evaluation means that it does not self-reference.

Secondly, we assert that an evaluation that does not request an attribute will never change the context's return value for that attribute. We know this because every semantic rule that changes the value of an attribute (AttrFetchCached, AttrFetchHO, CacheWrite) only changes the value of an attribute that has been requested. Therefore we know that the context's output expression will not be changed during the evaluation of that expression. □

Lemma 30.

$$\begin{aligned}
&\forall(\sigma \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),\\
&\forall(n \in \boldsymbol{N}),(a \in \boldsymbol{A}),(p \in \rho(a)),(t \in \boldsymbol{T}),(v \in t),\\
&\quad \sigma \vdash \sigma(n,a,p) \twoheadrightarrow \sigma \vdash v && (30.1)\\
&\implies \sigma \equiv \sigma \oplus \{(n,a,p) \mapsto [\![v]\!]\} && (30.2)
\end{aligned}$$

Proof. This lemma states that if the output of a context function is evaluated to some value v without changing the context function, then the context function is equivalent to itself with the value v cached to it. In this proof we will call σ the left context and $\sigma \oplus \{(n,a,p) \mapsto [\![v]\!]\}$ the right context.

To prove that the left and right contexts are equivalent, we need to show that evaluating any attribute under each context will yield the same value. Consider the evaluation of some attribute under the left context. Either this evaluation will request the attribute described by (n,a,p) or it will not. In the case where this attribute is never requested, we can intuitively see that the single change to the right context will not affect the output value, as this change will not be accessed, and all other outputs are the same. In the case where the attribute is requested, we know from Lemma 27 that it will only request valued expressions during its evaluation. Even if some other evaluation had occurred before requesting (n,a,p) producing some new context function σ', we know that all outputs will be identical, as all requested attributes were values, from Lemma 26.

Now consider evaluating some expression under the right context. Again, if the attribute (n,a,p) is never requested, evaluation will be identical to evaluation under the left. If the attribute (n,a,p) is requested, the cached version will be used, producing the same result immediately. □

Lemma 31. For this lemma we consider only the extended calculus, with no higher order expressions or semantics.

$$\forall(\sigma_1, \sigma_2 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),$$
$$\forall(e \in \boldsymbol{E}), (t \in \boldsymbol{T}), (v \in t),$$
$$\sigma_1 \vdash e \twoheadrightarrow \sigma_2 \vdash v \tag{31.1}$$
$$\implies \sigma_1 \equiv \sigma_2 \tag{31.2}$$

Proof. The hard part of this proof is provided by Lemma 30. We proceed by induction on the derivation of the big step relation (31.1). Most cases are proved using the inductive hypotheses and with the use of the transitivity of context equivalence. We will show the working for the AttrFetchCached case only, as this is the difficult part of the proof.

Case 31.1 (AttrFetchCached)**.**

$$\sigma_1 \vdash e_1.a(e_2) \twoheadrightarrow \sigma_4 \oplus \{(n, a, p) \mapsto [\![v]\!]\} \vdash v \tag{31.3}$$
$$\sigma_3(n, a, p) \text{ is not a value} \tag{31.4}$$
$$\sigma_1 \vdash e_1 \twoheadrightarrow \sigma_2 \vdash n \tag{31.5}$$
$$\sigma_2 \vdash e_2 \twoheadrightarrow \sigma_3 \vdash p \tag{31.6}$$
$$\sigma_3 \vdash \sigma_3(n, a, p) \twoheadrightarrow \sigma_4 \vdash v \tag{31.7}$$
$$\implies \sigma_1 \equiv \sigma_4 \oplus \{(n, a, p) \mapsto [\![v]\!]\} \tag{31.8}$$

The following can be inferred.

$$\sigma_1 \equiv \sigma_2 \tag{31.9}$$
$$\sigma_2 \equiv \sigma_3 \tag{31.10}$$
$$\sigma_3 \equiv \sigma_4 \tag{31.11}$$
$$\sigma_1 \equiv \sigma_4 \tag{31.12}$$
$$\sigma_4 \vdash \sigma_3(n, a, p) \twoheadrightarrow \sigma_4 \vdash v \tag{31.13}$$
$$\sigma_4(n, a, p) = \sigma_3(n, a, p) \tag{31.14}$$
$$\sigma_4 \vdash \sigma_4(n, a, p) \twoheadrightarrow \sigma_4 \vdash v \tag{31.15}$$
$$\implies \sigma_4 \equiv \sigma_4 \oplus \{(n, a, p) \mapsto [\![v]\!]\} \tag{31.16}$$

The inductive hypotheses associated with (31.5) to (31.7) provide the equivalences (31.9) to (31.11). (31.12) is provided by the transitivity of context equivalence, and (31.5) to (31.7). (31.13) is given by applying Lemma 28 to (31.7). (31.14) is given by applying Lemma 29 to (31.7). We obtain (31.15) by rewriting the equality (31.14) in (31.13).

We have obtained a new goal (31.16) through the transitivity of context equivalence with (31.12). The hypothesis (31.15) is the key ingredient for this proof, and can complete the proof using Lemma 30.

□

Lemma 32.

$$\forall(\sigma \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),$$
$$\forall(n \in \boldsymbol{N}), (a \in \boldsymbol{A}), (p \in \rho(a)), (f_l \in \boldsymbol{N} \to (\boldsymbol{A}_u \times \boldsymbol{E})), (n_f \in \boldsymbol{N}),$$
$$\sigma(n,a,p) = \texttt{MK}f_l \tag{32.1}$$
$$n_f \text{ does not exist in } \sigma \tag{32.2}$$
$$\implies \sigma \equiv \sigma \oplus \{(n,a,p) \mapsto [\![n_f]\!]\} \otimes n_f/f_l(n_f) \tag{32.3}$$

Proof. This lemma states that if some particular attribute request (n,a,p) returns a `MK` expression, then the context function returning it is equivalent to the same context function with the contents of the `MK` expression written to it. As in Lemma 30, we consider the evaluation of some attribute request under the left context. First we consider the case where evaluation does not request the attribute (n,a,p). The caching of the higher order attribute (n,a,p) will not have any effect on evaluation, and we also know that no attributes of n_f will be requested, as the only way to obtain the node n_f is by evaluating (n,a,p).

If the attribute (n,a,p) is requested, immediately the changes shown on the right context are applied to the current context, and the value n_f is returned. It does not matter what other changes have been made to the context in the mean time, as higher order creations are completely independent of the state of the context function (by the definition of node existence). From this point on, all evaluations are trivially identical to those under the right context, as the only difference between the contexts has now been matched. It is only possible for attributes of n_f to be requested after the higher order attribute n_f through evaluating (n,a,p) has been created, as creating n_f is the only way to gain access to this node.

Considering the inverse is similar. Evaluating some attribute request under the right context, if never requesting (n,a,p), will yield the same result as under the left context. If evaluation under the right context does request (n,a,p), it will be the same to request the cached node n_f as it would be to create this node and return it.

Let us also consider that some of the properties written to the context function for the newly created higher order node n_f are not values, but non-value expressions. If some evaluation creates the node n_f and then evaluates one of its non-value attributes a_1, then the result is a context function that is not perfectly described by $\_ \oplus \{(n,a,p) \mapsto [\![n_f]\!]\} \otimes n_f/f_l(n_f)$. This case is already covered by Lemma 30.

This case, where higher order context changes interact with caching context changes, has proven to be the most difficult part of cache irrelevance to mechanise a proof for. This lemma, Lemma 32, is the only one of the lemmas presented here that has not been mechanised[1]. The mutual dependence between Lemmas 30 and 32[2] has proven quite difficult to express, especially given the extra complexities needed to mechanise higher order concepts such as the "does not exist" relation between a node and a context function.

Nevertheless, while this lemma's proof is not mechanised, we are confident that it holds and we are confident that a proof for it can be mechanised given enough time. We are happy with our explanation and understanding of the proof's structure. □

Theorem 33. We now present the key theorem of what we call "big step equivalence". This is expressed identically as Lemma 31, but here we consider the full higher-order semantics.

[1]Actually, at the time of this writing, Lemma 29 is also not mechanised in the current version of the Lean calculus, but it has been proven in earlier versions.

[2]Because of the way we implement higher order node creation in our mechanisation, Lemma 30 depends on the result in Lemma 32 for its proof.

$$\forall(\sigma_1,\sigma_2 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),$$
$$\forall(e \in \boldsymbol{E}),(t \in \boldsymbol{T}),(v \in t),$$
$$\sigma_1 \vdash e \twoheadrightarrow \sigma_2 \vdash v \tag{33.1}$$
$$\implies \sigma_1 \equiv \sigma_2 \tag{33.2}$$

Proof. This theorem states that if any expression is evaluated under some context function σ_1, resulting in a new context function σ_2, these two context functions are equivalent. As this theorem has been proven for the extended calculus in Lemma 31, we need only consider the new semantic rule in the higher order calculus; AttrFetchHO. Similarly to our proof for Lemma 31, this proof's heavy lifting is performed by Lemma 32.

Case 33.1 (AttrFetchHO)**.**

$$\sigma_1 \vdash e_1.a(e_2) \twoheadrightarrow \sigma_3 \oplus \{(n,a,p) \mapsto [\![n_f]\!]\} \otimes n_f/f_l(n_f) \vdash n_f \tag{33.3}$$
$$\sigma_3(n,a,p) = \texttt{MK}f_l \tag{33.4}$$
$$n_f \text{ does not exist in } \sigma_3 \tag{33.5}$$
$$\sigma_1 \vdash e_1 \twoheadrightarrow \sigma_2 \vdash n \tag{33.6}$$
$$\sigma_2 \vdash e_2 \twoheadrightarrow \sigma_3 \vdash p \tag{33.7}$$
$$\implies \sigma_1 \equiv \sigma_3 \oplus \{(n,a,p) \mapsto [\![n_f]\!]\} \otimes n_f/f_l(n_f) \tag{33.8}$$

The inductive hypotheses and the transitivity of context equivalence give us the new goal (33.9).

$$\implies \sigma_3 \equiv \sigma_3 \oplus \{(n,a,p) \mapsto [\![n_f]\!]\} \otimes n_f/f_l(n_f) \tag{33.9}$$

Lemma 32, along with (33.4) and (33.5), is sufficient to solve this goal.

Theorem 33 is proven. □

Theorem 34. We proved in Theorem 33 that the big step relation always produces an output context function that is equivalent to the input context function. Here we prove that the multistep relation has the same property.

$$\forall(\sigma_1,\sigma_2 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),$$
$$\forall(e \in \boldsymbol{E}),(t \in \boldsymbol{T}),(v \in t),$$
$$\sigma_1 \vdash e \longrightarrow^* \sigma_2 \vdash v \tag{34.1}$$
$$\implies \sigma_1 \equiv \sigma_2 \tag{34.2}$$

Proof. Proof is trivial from Theorem 25. Since the the multistep and big step relations imply each other, we can use Theorem 33 to prove Theorem 34. □

Theorem 35. We present and prove this theorem, which is really a variation of Theorem 34, so that we can more conveniently use the equivalence property in proofs. We actually present two theorems here, which are expansions of each direction of context equivalence.

$$\begin{aligned}
&\forall(\sigma_1,\sigma_2,\sigma_3 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),\\
&\forall(e_1,e_2 \in \boldsymbol{E}),(t_1,t_2 \in \boldsymbol{T}),(v_1 \in t_1),(v_2 \in t_2)\\
&\qquad \sigma_1 \vdash e_1 \longrightarrow^* \sigma_2 \vdash v_1 && (35.1)\\
&\qquad \sigma_1 \vdash e_2 \longrightarrow^* \sigma_3 \vdash v_2 && (35.2)\\
&\implies \exists\sigma_4,\sigma_2 \vdash e_2 \longrightarrow^* \sigma_4 \vdash v_2 && (35.3)
\end{aligned}$$

$$\begin{aligned}
&\forall(\sigma_1,\sigma_2,\sigma_3 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),\\
&\forall(e_1,e_2 \in \boldsymbol{E}),(t_1,t_2 \in \boldsymbol{T}),(v_1 \in t_1),(v_2 \in t_2)\\
&\qquad \sigma_1 \vdash e_1 \longrightarrow^* \sigma_2 \vdash v_1 && (35.4)\\
&\qquad \sigma_2 \vdash e_2 \longrightarrow^* \sigma_3 \vdash v_2 && (35.5)\\
&\implies \exists\sigma_4,\sigma_1 \vdash e_2 \longrightarrow^* \sigma_4 \vdash v_2 && (35.6)
\end{aligned}$$

For each of the above theorems, the goal can be satisfied by applying Theorem 34 to the first multistep relation to show equivalence, then applying that equivalence to the second multistep relation. Since this is the natural result of context equivalence, and we use this result frequently in our proofs from now on, we will use the phrase "by multistep equivalence" to mean that we are using this result.

Definition 6.8.2 (Alternative Context Equivalence)**.** We also use a more generalised version of context equivalence, represented by the $\equiv'$ operator. The definitions given here are the same as Definition 6.8.1, except expressing that any expression will evaluate to the same value in both contexts, not just any attribute evaluation.

$$\begin{aligned}
&\forall(\sigma_1,\sigma_2 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),\\
\sigma_1 \equiv\rangle' \sigma_2 \iff\ &\forall(e \in \boldsymbol{E}),(t \in \boldsymbol{T}),(v \in t),({\sigma_1}' \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),\\
&(\sigma_1 \vdash e \twoheadrightarrow {\sigma_1}' \vdash v \implies \exists{\sigma_2}',\sigma_2 \vdash e \twoheadrightarrow {\sigma_2}' \vdash v)
\end{aligned}$$

$$\begin{aligned}
&\forall(\sigma_1,\sigma_2 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),\\
&\sigma_1 \equiv' \sigma_2 \iff \sigma_1 \equiv\rangle' \sigma_2 \wedge \sigma_2 \equiv\rangle' \sigma_1
\end{aligned}$$

We now prove that both definitions of context equivalence are equivalent.

Lemma 36.

$$\begin{aligned}
&\forall(\sigma_1,\sigma_2 \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E})\\
&(\sigma_1 \equiv \sigma_2) \iff (\sigma_1 \equiv' \sigma_2) && (36.1)
\end{aligned}$$

Proof. This lemma states that if two contexts are equivalent, they are also alternatively equivalent (equivalent using the $\equiv'$ operator). Proof that alternative equivalence implies equivalence is trivial, as alternative equivalence is a more general case of equivalence.

Proof that equivalence implies alternative equivalence is given from examining the semantics of the multistep relation; considering the evaluation one small step at a time. Some step rules do not use their context function, so are trivially proven. Some step rules only use the context function in their sub steps, so are shown via induction. Any steps that access the context function can use the definition of context equivalence to satisfy their goal. □

6.9 Conclusion

In this chapter we have presented proofs for a number of theorems about the Saiga calculus. While verbose, the proofs for step determinism, type determinism, type preservation, and progress were simple in their approach. Showing equivalence between the big step and multistep relations was also straightforward, and a useful result for future proofs. Cache irrelevance has been by far the most difficult property to prove, with only a high-level summary of its proof given here. This effort is worthwhile, as cache irrelevance is a very important result for further analysis, as it simplifies the analysis of evaluation with a changing context function, showing that when a context function changes during normal evaluation, it remains semantically equivalent.

Overall, we have shown that our operational semantics do hold these important properties, and have shown that our calculus provides a framework for proving these and other properties of attribute grammar evaluation.

To brew is human, to ferment is divine.
Divine Barrel Brewing – Charlotte, NC

7 Example

In this chapter we demonstrate how Saiga can be used to analyse and compare attribute grammar programs written in different real-world attribute grammar platforms. As a basis for this analysis, we consider type analysis for Featherweight Java, based on the calculus presented by Igarashi *et al.* [9].

A working compiler for Featherweight Java programs was written based on Igarashi's calculus, by two experienced attribute grammar researchers, each in their own attribute grammar platform. Tony Sloane, the developer of the Kiama language workbench, provided an implementation written in Kiama. Niklas Fors, who has been involved in the maintenance of the JastAdd attribute grammar platform, provided an implementation written in JastAdd.

As each implementation was written by an author very familiar with their platform, the particulars of these implementations align well with the expected design style for each platform. As both implementations are based on a common calculus (the Featherweight Java calculus presented in [9]), the logical backbone of each implementation aligns with the other. However there is a key difference between the two approaches: the Kiama implementation uses the *environment* method of name analysis, while the JastAdd implementation uses the *lookup* method.

The environment method of name analysis involves compiling a list of all names in each scope for each scope-creating node in a program tree, and performing a lookup in this list to associate name uses with their declarations. An example of this approach is given in Section 3.3. The lookup method of name analysis involves performing a targeted search for a particular name every time a name's declaration is needed. An example of this approach is given in Section 4.1.5.

A repository containing the complete code of the Kiama implementation can be found at `https://bitbucket.org/scottbuckley/fwjava-kiama`, and the JastAdd implementation at `https://bitbucket.org/scottbuckley/fwjava-jastadd`.

The layout of this chapter is as follows. Section 7.1 discusses the abstract grammars of the Kiama and JastAdd implementations of Featherweight Java, finding a common ground between these systems and presenting an abstract grammar in a Saiga-friendly notation. Section 7.2 presents the Saiga specification of the Kiama and JastAdd attributes that makes up

Featherweight Java's type system. Section 7.3 presents proofs for some lemmas that set up a framework for our major proofs for this chapter. Section 7.4 presents a series of lemmas and theorems building to a proof that the type attribute always evaluates to the same value between the Kiama and JastAdd specifications. Section 7.5 explores an quantitative analysis of the two specifications, proving a relationship between the evaluation step counts for identifier lookup in Featherweight Java.

7.1 The Abstract Grammar

The Kiama implementation uses the sbt-rats parser generator [52] to generate a set of Scala case classes that represent an abstract syntax tree for a Featherweight Java program. A sample of this specification is given in Figure 7.1. The JastAdd implementation explicitly defines an abstract syntax tree in their own syntax, a sample of which is given in Figure 7.2. The full text of these specifications in JastAdd and Kiama are given by Appendices A.1.1 and A.1.2 in Appendix A.1.

These two specifications describe compatible trees, which is not surprising as they are both based on the specification given in [9]. In Figure 7.3 we present an abstract grammar in the same form we have presented throughout this thesis. To recap this notation, the first line indicates that *Program* is a possible value of nodeType, there are expected to be sensible values for the attributes classes and expr for a *Program* node, $\tau(\mathsf{classes}) = \mathit{listNode}$, and it is expected that every node contained by a value of classes will have the nodeType of *ClassDecl*.

While we do not list the value *Expr* as a node type, we use this as a catch-all for the types *EIdn*, *EFld*, *ECall*, *ECast*, and *ENew*. Such a strategy has a simple implementation: while we can assume underlying functions *isEIdn etc.* to check for a particular type value, we can similarly assume an underlying function *isExpr* which is returns true if any of the appropriate types are given. Also in our notation above when we write *Expr* in square brackets (as in the *Program* case), we mean that the node returned by expr will be of some type that would match using the *isExpr* function.

Figure 7.4 shows an example of a tree that conforms to our abstract syntax.

7.2 The Attributes

The definition of all relevant attributes in each implementation is given in Appendices A.2.1 and A.2.2. We have translated these attributes into Saiga attributes in the context functions σ_k and σ_j, which we describe below. We have used all of the features described in this thesis: reference attributes, parameterised attributes, and higher order attributes. We have performed this translation as a roughly one-to-one translation between Kiama/JastAdd attributes and Saiga attributes, with some exceptions where we needed to describe auxiliary attributes to implement some of the deeper functionalities available in Kiama and JastAdd. Some attributes are defined identically between σ_k and σ_j, such as _matchName[1], which is not surprising as both the Kiama and JastAdd versions are implementing the same semantics.

[1]We prefix _matchName (and some other attribute names) with an underscore to note that they are performing utility tasks, independent of the semantics of the particular problem.

```
object FWJavaParserSyntax {
  sealed abstract class ASTNode extends Product

  case class Program (optClassDecls : Vector[ClassDecl],
                                expr : Expr) extends ASTNode

  case class ClassDecl (identifier : String,
                            idnUse : IdnUse,
              optFieldOrParamDecls : Vector[FieldOrParamDecl],
                          ctorDecl : CtorDecl,
                   optMethodDecls : Vector[MethodDecl]) extends ASTNode

  case class CtorDecl (idnUse : IdnUse,
         optFieldOrParamDecls : Vector[FieldOrParamDecl],
                   optIdnUses : Vector[IdnUse],
                optFieldInits : Vector[FieldInit]) extends ASTNode

  case class MethodDecl (idnUse : IdnUse,
                      identifier : String,
           optFieldOrParamDecls : Vector[FieldOrParamDecl],
                          expr : Expr) extends ASTNode

  ...
}
```

Figure 7.1: A sample of the abstract grammar used by the Kiama implementation.

```
  Program ::= ClassDecl* [Expr];

  abstract TypeDecl;
  ClassDecl : TypeDecl ::= <Name> Extends:TypeUse
                           FPDecl* CtorDecl MethodDecl*;
  CtorDecl   ::= TypeUse FPDecl* Super:VarUse* FieldInit*;
  MethodDecl ::= TypeUse <Name> FPDecl* Expr;

  ...
```

Figure 7.2: A sample of the abstract grammar used by the JastAdd implementation.

```
  Program ::=  classes(list node)[ClassDecl], expr(node)[Expr]
ClassDecl ::=  name(string), extUse(node)[TypeUse],
               args(list node)[FPDecl], ctor(node)[CtorDecl],
               methods(list node)[MethodDecl]
 CtorDecl ::=  typeUse(node)[TypeUse], args(list node)[FPDecl],
               super(list node)[FPDecl], inits(list node)[FieldInit]
MethodDecl::=  name(string), rtnUse(node)[TypeUse],
               args(list node)[FPDecl], expr(node)[Expr]
     EIdn ::=  varUse(node)[VarUse]
     EFld ::=  expr(node)[Expr], nameUse(node)[VarUse]
    ECall ::=  expr(node)[Expr], nameUse(node)[VarUse],
               exprs(list node)[Expr]
    ECast ::=  typeUse(node)[TypeUse], expr(node)[Expr]
     ENew ::=  typeUse(node)[TypeUse], exprs(list node)[Expr]
   FPDecl ::=  typeUse(node)[TypeUse], nameDef(node)[IdnDef]
FieldInit ::=  leftUse(node)[varUse], rightUse(node)[varUse]
  TypeUse ::=  name(string)
   VarUse ::=  name(string)
   IdnDef ::=  name(string)
```

Figure 7.3: The abstract syntax tree used in this example.

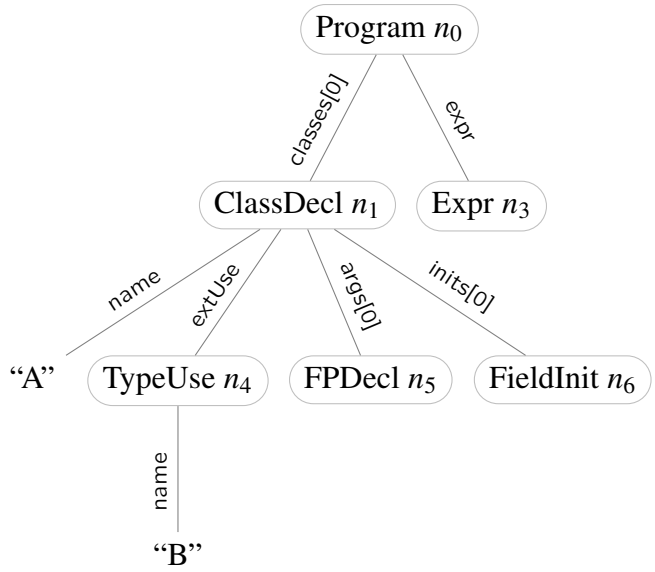


Figure 7.4: An example tree that matches the abstract syntax described in Figure 7.3.

7.2.1 The Kiama Implementation

We show the full definitions here of all attributes defined for the Saiga translation of the Kiama implementation of the Featherweight Java type analysis semantics, as shown in Appendix A.2.1. We will not walk through every attribute's translation into a Saiga specification, but we will explain the translation for the env attribute. In this chapter we use the notation e_1 `IFNULL` e_2 as a shorthand for `IF` *isNull*(e_1) `THEN` e_2 `ELSE` e_1.

env

The env attribute computes a list of all field and parameter declarations visible to a node. The Kiama attribute env is defined as follows.

```
1  val env : ASTNode => Vector[FPDecl] =
2    attr {
3      case l @ ClassDecl(c, _, _, _, _) =>
4        FPDecl(TypeUse(c), IdnDef("this")) +: fieldsRef(l).get
5      case k : CtorDecl =>
6        k.optFPDecls
7      case parent.pair(m : MethodDecl, p) =>
8        m.optFPDecls ++ env(p)
9      case parent(p) =>
10       env(p)
11     case _ =>
12       Vector()
13   }
```

This attribute is translated to the following Saiga specification.

$$\sigma_k(n, \mathsf{env}) = \begin{cases} [\![\mathit{prepend}]\!]([\![n]\!].\mathsf{thisFPDecl})([\![n]\!].\mathsf{fields}) & \text{(ClassDecl)} \\ [\![n]\!].\mathsf{args} & \text{(CtorDecl)} \\ [\![\mathit{concat}]\!]([\![n]\!].\mathsf{args})([\![n]\!].\mathsf{parent.env}) & \text{(MethodDecl)} \\ [\![[]]\!] & \text{(Program)} \\ \text{inherited} \end{cases}$$

The Kiama implementation splits the attribute equation into five parts, using pattern matching on the input node, as is standard in Kiama. The Saiga specification splits the attribute equation into the same five parts, using our selector notation.

Line 3 in the Kiama implementation matches against a node of type *ClassDecl*, binding its name to the variable c. This variable is used in constructing a higher order node of type FPDecl, which is prepended to the list returned by the fieldsRef attribute. The first selector in our Saiga specification delegates creation of the higher order FPDecl node to the thisFPDecl attribute, and uses the underlying *prepend* function to prepend this node to the list returned by the fields attribute. We consider this to be a faithful translation from Kiama to Saiga.

Attribute names are not mapped exactly between the Kiama implementation and its Saiga specification, and our syntax for prepending and concatenation in Saiga is not the same as in Kiama, but otherwise the rest of the env attribute maps directly between Kiama and Saiga.

decl

The decl attribute takes a class name (string parameter), and tries to find a class declaration matching that class name.

$$\sigma_k(n, \mathsf{decl}, s) = \begin{cases} [\![n]\!].\mathsf{objectClassDecl} & s = \text{“Object”} \\ [\![n]\!].\_\mathsf{matchName}((.[\![s]\!], [\![n]\!].\mathsf{classes})) & \text{(Program)} \\ \text{inherited} \end{cases}$$

type

The type attribute returns the type of an expression, as a reference to the node where that type was declared.

$$\sigma_k(n, \mathsf{type}) =$$

$$\begin{cases} [\![n]\!].\mathsf{varUse.varUseType} & \text{(EIdn)} \\ [\![n]\!].\mathsf{findFP}(([\![n]\!].\mathsf{expr.type.fields}, [\![n]\!].\mathsf{nameUse.name})).\mathsf{fpType} & \text{(EFld)} \\ \mathtt{IF}\ [\![\mathit{isNull}]\!]([\![n]\!].\mathsf{methodDecl})\ \mathtt{THEN}\ [\![n_{null}]\!] & \text{(ECall)} \\ \quad \mathtt{ELSE\ IF}\ [\![n]\!].\_\mathsf{exprsMatchFPs}(([\![n]\!].\mathsf{exprs}, [\![n]\!].\mathsf{methodDecl.args})) & \\ \quad \mathtt{THEN}\ [\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{methodDecl.rtnUse.name}) & \\ \quad \mathtt{ELSE}\ [\![n_{null}]\!] & \\ \mathtt{IF}\ [\![\mathit{isNull}]\!]([\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{typeUse.name}))\ \mathtt{THEN}\ [\![n_{null}]\!] & \text{(ENew)} \\ \quad \mathtt{ELSE\ IF}\ [\![n]\!].\_\mathsf{exprsMatchFPs}(& \\ \quad\ ([\![n]\!].\mathsf{exprs}, [\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{typeUse.name}).\mathsf{fields})) & \\ \quad \mathtt{THEN}\ [\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{typeUse.name}) & \\ \quad \mathtt{ELSE}\ [\![n_{null}]\!] & \\ \mathtt{IF}\ [\![\mathit{isNull}]\!]([\![n]\!].\mathsf{expr.type}) & \text{(ECast)} \\ \quad \mathtt{THEN}\ [\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{typeUse.name}) & \\ \quad \mathtt{ELSE}\ [\![n_{null}]\!] & \\ [\![n]\!].\mathsf{fpType} & \text{(FPDecl)} \\ [\![n_{null}]\!] & \text{otherwise} \end{cases}$$

type Aux Attributes

$$\sigma_k(n, \mathsf{fpType}) = \begin{cases} [\![n_{null}]\!] & n = n_{null} \\ [\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{typeUse.name}) & \text{otherwise} \end{cases}$$

$$\sigma_k(n, \mathsf{methodDecl}) = [\![n]\!].\mathsf{expr.type.method}([\![n]\!].\mathsf{nameUse.name})$$

$$\sigma_k(n, \mathsf{varUseType}) = [\![n]\!].\mathsf{findFP}(([\![n]\!].\mathsf{env}, [\![n]\!].\mathsf{name})).\mathsf{fpType}$$

$$\sigma_k(n, \mathsf{findFP}, (l,s)) = \begin{cases} [\![n_{null}]\!] & n = n_{null} \\ [\![n_{null}]\!].\_\mathsf{matchName}([\![(s,l)]\!]) & \text{otherwise} \end{cases}$$

fields

The fields attribute, when called on a node of type *ClassDecl*, returns a list of the fields visible to that class (including those it inherits).

$$\sigma_k(n, \mathsf{fields}) = \begin{cases} \mathtt{IF}\ [\![\mathit{strEq}]\!]([\![n]\!].\mathsf{name})([\![\text{“Object”}]\!]) & \text{(ClassDecl)} \\ \ \mathtt{THEN}\ [\![[]]\!] & \\ \ \mathtt{ELSE}\ [\![\mathit{concat}]\!] & \\ \quad ([\![n]\!].\mathsf{args}) & \\ \quad ([\![n]\!].\mathsf{superClass}.\mathsf{fields}) & \\ & \\ [\![[]]\!] & \text{otherwise} \end{cases}$$

superClass

The superClass attribute returns a reference to a class's superclass.

$$\sigma_k(n, \mathsf{superClass}) = \begin{cases} [\![n_{null}]\!] & n = n_{null} \\ \mathtt{IF}\ [\![\mathit{strEq}(\text{“Object”})]\!]([\![n]\!].\mathsf{name}) & \text{otherwise} \\ \ \mathtt{THEN}\ [\![n_{null}]\!] & \\ \ \mathtt{ELSE}\ [\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{extUse}.\mathsf{name}) & \end{cases}$$

method

The method takes a string parameter and returns a reference to the declaring instance of a method with that name

$$\sigma_k(n, \mathsf{method}, s) = \begin{cases} [\![n_{null}]\!] & s = \text{“Object”} \\ [\![n]\!].\_\mathsf{matchName}(([\![s]\!], [\![n]\!].\mathsf{methods})) & \text{(ClassDecl)} \\ \ \mathtt{IFNULL}\ [\![n]\!].\mathsf{superClass}.\mathsf{method}([\![s]\!]) & \\ [\![n_{null}]\!] & \text{otherwise} \end{cases}$$

subTypeOf

The subTypeOf attribute takes a parameter (the name of a class), and returns a boolean value representing whether or not the current class is a subtype of a class of that name.

$$\sigma_k(n,\mathsf{subTypeOf},s) = \begin{cases} [\![false]\!] & n = n_{null} \\ [\![true]\!] & s = \text{“Object”} \\ \texttt{IF } [\![strEq]\!]([\![n]\!].\mathsf{name})([\![\text{“Object”}]\!]) & \text{otherwise} \\ \texttt{ THEN } [\![false]\!] \\ \texttt{ ELSE IF } [\![strEq]\!]([\![n]\!].\mathsf{name})([\![s]\!]) \\ \texttt{ THEN } [\![true]\!] \\ \texttt{ ELSE } [\![n]\!].\mathsf{superClass}.\mathsf{subTypeOf}([\![s]\!]) \end{cases}$$

thisFPDecl

The thisFPDecl attribute creates a new subtree that is a *FPDecl*, referencing the current class, representing the keyword “this”.

$$\begin{aligned}\sigma_k(n_p,\mathsf{thisFPDecl}) = \texttt{MK}\lambda n,[&(\mathsf{nodeType}, [\![FPDecl]\!]),\\ &(\mathsf{parent}, [\![n_p]\!]),\\ &(\mathsf{nameDef}, [\![n]\!].\mathsf{buildIdnDef}([\![\text{“this”}]\!]))\\ &(\mathsf{typeUse}, [\![n]\!].\mathsf{buildTypeUse}([\![n_p]\!].\mathsf{name}))]\end{aligned}$$

objectClassDecl

The objectClassDecl attribute returns (and maybe constructs) a special “Object” class declaration.

$$\sigma_k(n,\mathsf{objectClassDecl}) = \begin{cases} [\![n]\!].\mathsf{buildObjectClassDecl} & \text{(Program)} \\ \text{inherited} \end{cases}$$

Aux Node Construction Attributes

$\sigma_k(n_p,\mathsf{buildTypeUse},s) = \texttt{MK}\lambda n,[(\mathsf{nodeType}, [\![\text{TypeUse}]\!]), (\mathsf{name}, [\![s]\!]), (\mathsf{parent}, [\![n_p]\!])]$

$\sigma_k(n_p,\mathsf{buildIdnDef},s) = \texttt{MK}\lambda n,[(\mathsf{nodeType}, [\![\text{IdnDef}]\!], (\mathsf{name}, [\![s]\!]), (\mathsf{parent}, [\![n_p]\!])]$

$$\begin{aligned}\sigma_k(n_p,\mathsf{buildObjectClassDecl}) = \texttt{MK}\lambda n,[&(\mathsf{nodeType}, [\![ClassDecl]\!]),\\ &(\mathsf{name}, [\![\text{“Object”}]\!]),\\ &(\mathsf{extUse}, [\![n]\!].\mathsf{buildTypeUse}([\![\text{“”}]\!])),\\ &(\mathsf{ctor}, [\![n]\!].\mathsf{buildObjectCtorDecl}),\\ &(\mathsf{args}, [\![[]]\!]),\\ &(\mathsf{methods}, [\![[]]\!]),\\ &(\mathsf{parent}, [\![n_p]\!])]\end{aligned}$$

$$\sigma_k(n_p, \mathsf{buildObjectCtorDecl}) = \mathtt{MK}\lambda n, [(\mathsf{nodeType}, [\![\mathit{ClassDecl}]\!]),\ (\mathsf{name}, [\![\text{“Object”}]\!]),\ (\mathsf{args}, [\![[]]\!]),\ (\mathsf{supers}, [\![[]]\!]),\ (\mathsf{inits}, [\![[]]\!]),\ (\mathsf{parent}, [\![n_p]\!])]$$

Helper Attributes

$$\sigma_k(n, \_\mathsf{matchName}, (s,l)) = \begin{cases} [\![n_{null}]\!].\_\mathsf{matchName}([\![(s,l)]\!]) & n \neq n_{null} \\ [\![n_{null}]\!] & l = [] \\ \mathtt{IF}\ [\![\mathit{strEq}]\!]([\![\mathit{fst}(l)]\!].\mathsf{getName})([\![s]\!]) & \text{otherwise} \\ \ \mathtt{THEN}\ [\![\mathit{fst}(l)]\!] \\ \ \mathtt{ELSE}\ [\![n]\!].\_\mathsf{matchName}([\![(s, \mathit{rest}(l))]\!]) \end{cases}$$

$$\sigma_k(n, \mathsf{getName}) = \begin{cases} [\![n]\!].\mathsf{nameDef.name} & \text{(FPDecl)} \\ [\![n]\!].\mathsf{name} & \text{otherwise} \end{cases}$$

$$\sigma_k(n, \_\mathsf{exprsMatchFPs}, (es,fps)) = \begin{cases} [\![n_{null}]\!].\_\mathsf{exprsMatchFPs}([\![(es,fps)]\!]) & n \neq n_{null} \\ [\![\mathit{false}]\!] & \mathit{len}(es) \neq \mathit{len}(fps) \\ [\![\mathit{true}]\!] & es = fps = [] \\ \mathtt{IF}\ [\![\mathit{fst}(es)]\!].\mathsf{type.subTypeOf}([\![\mathit{fst}(fps)]\!].\mathsf{typeUse.name}) & \text{otherwise} \\ \ \mathtt{THEN}\ [\![n]\!].\_\mathsf{exprsMatchFPs}([\![(\mathit{rest}(es), \mathit{rest}(fps))]\!]) \\ \ \mathtt{ELSE}\ [\![\mathit{false}]\!] \end{cases}$$

7.2.2 The JastAdd Implementation

We show the full definitions here of all attributes defined for the Saiga translation of the JastAdd implementation of the Featherweight Java type analysis semantics, as shown in Appendix A.2.2. We will not walk through every attribute's translation into a Saiga specification, but we will explain the translation for the decl attribute.

decl

The decl attribute returns a reference to the declaring instance of a type or variable use, or the method declaration associated with a method call expression. The JastAdd attribute decl is defined as follows.

```
1 syn TypeDecl TypeUse.decl() = lookupType(getName());
2 syn FPDecl VarUse.decl() = lookup(getName());
3 syn MethodDecl ECall.decl()
4             = getExpr().type().lookupMethod(getVarUse().getName());
```

This attribute is translated to the following Saiga specification.

$$\sigma_j(n,\mathsf{decl}) = \begin{cases} [\![n]\!].\mathsf{lookupType}([\![n]\!].\mathsf{name}) & \text{(TypeUse)} \\ [\![n]\!].\mathsf{lookup}([\![n]\!].\mathsf{name}) & \text{(VarUse)} \\ [\![n]\!].\mathsf{expr.type.lookupMethod}([\![n]\!].\mathsf{nameUse.name}) & \text{(ECall)} \\ [\![n_{null}]\!] & \text{otherwise} \end{cases}$$

The JastAdd implementation splits the attribute equation into three different parts, matching against nodes of type *TypeUse*, *VarUse*, and *ECall*. The Saiga specification of this attribute uses selectors to match against these same three node types, with a fourth selector used to specify a default value (as Saiga attributes must be defined on all nodes). It is plain from observation that the three attribute expressions returned in our Saiga specification directly match the semantics of the three attribute expressions listed in the JastAdd implementation.

The type attribute (on the next page) returns the type of an expression, as a reference to the node where that type was declared.

type

$$\sigma_j(n,\mathsf{type}) = \begin{cases}
\texttt{IF } [\![isNull]\!]([\![n]\!].\mathsf{typeUse.decl}) & \text{(ENew)} \\
\texttt{THEN } [\![n_{null}]\!] & \\
\texttt{ELSE IF } [\![n]\!]._\mathsf{exprsMatchFPs}(& \\
\quad ([\![n]\!].\mathsf{exprs}, [\![n]\!].\mathsf{typeUse.decl.fields})) & \\
\texttt{THEN } [\![n]\!].\mathsf{typeUse.decl} & \\
\texttt{ELSE } [\![n_{null}]\!] & \\
 & \\
\texttt{IF } [\![isNull]\!]([\![n]\!].\mathsf{decl}) & \text{(ECall)} \\
\texttt{THEN } [\![n_{null}]\!] & \\
\texttt{ELSE IF } [\![n]\!]._\mathsf{exprsMatchFPs}(& \\
\quad ([\![n]\!].\mathsf{exprs}, [\![n]\!].\mathsf{decl.args})) & \\
\texttt{THEN } [\![n]\!].\mathsf{decl.rtnUse.decl} & \\
\texttt{ELSE } [\![n_{null}]\!] & \\
 & \\
[\![n]\!].\mathsf{varUse.decl.type} & \text{(EIdn)} \\
 & \\
[\![n]\!].\mathsf{nameUse.decl.type} & \text{(EFld)} \\
 & \\
\texttt{IF } [\![isNull]\!]([\![n]\!].\mathsf{expr.type}) & \text{(ECast)} \\
\texttt{THEN } [\![n_{null}]\!] & \\
\texttt{ELSE } [\![n]\!].\mathsf{typeUse.decl} & \\
 & \\
[\![n]\!].\mathsf{typeUse.decl} & \text{(FPDecl)} \\
 & \\
[\![n_{null}]\!] & \text{otherwise}
\end{cases}$$

subTypeOf

The subTypeOf attribute takes a parameter (the name of a class), and returns a boolean value representing whether or not the current class is a subtype of a class of that name.

$$\sigma_j(n,\mathsf{subTypeOf},s) = \begin{cases}
[\![false]\!] & n = n_{null} \\
[\![true]\!] & s = \text{“Object”} \\
\texttt{IF } [\![strEq]\!]([\![n]\!].\mathsf{name})([\![\text{“Object”}]\!]) & \text{otherwise} \\
\texttt{THEN } [\![false]\!] & \\
\texttt{ELSE IF } [\![strEq]\!]([\![n]\!].\mathsf{name})([\![s]\!]) & \\
\texttt{THEN } [\![true]\!] & \\
\texttt{ELSE } [\![n]\!].\mathsf{superClass.subTypeOf}([\![s]\!]) &
\end{cases}$$

lookup

The lookup attribute takes a parameter (the name of a variable) and searches up the tree for the defining instance of that variable.

$\sigma_j(n, \mathsf{lookup}, s) =$

$$\begin{cases} \texttt{IF } [\![strEq(s)(\text{“this”})]\!] & \text{(MethodDecl).expr} \\ \quad \texttt{THEN } [\![n]\!].\mathsf{parent.enclosingClassDecl.thisFPDecl} & \\ \quad \texttt{ELSE } ([\![n]\!].\_\mathsf{matchName}([\![s]\!], [\![n]\!].\mathsf{parent.args})) & \\ \quad\quad \texttt{IFNULL } [\![n]\!].\mathsf{parent.lookup}([\![s]\!])) & \\ & \\ [\![n]\!].\mathsf{parent.remoteLookup}([\![s]\!]) & \text{(ClassDecl).methods*} \\ [\![n]\!].\mathsf{parent.expr.type.remoteLookup}([\![s]\!]) & \text{(EFld).nameUse} \\ [\![n_{null}]\!] & \text{(Program).*} \\ \text{inherited} & \end{cases}$$

fields

The fields attribute, when called on a node of type *ClassDecl*, returns a list of the fields visible to that class (including those it inherits).

$$\sigma_j(n, \mathsf{fields}) = \begin{cases} \texttt{IF } [\![strEq]\!]([\![n]\!].\mathsf{name})([\![\text{“Object”}]\!]) & \text{(ClassDecl)} \\ \quad \texttt{THEN } [\![[]]\!] & \\ \quad \texttt{ELSE } [\![concat]\!] & \\ \quad\quad ([\![n]\!].\mathsf{args}) & \\ \quad\quad ([\![n]\!].\mathsf{superClass.fields}) & \\ & \\ [\![[]]\!] & \text{otherwise} \end{cases}$$

lookupType

The lookupType attribute takes a parameter (the name of a class) and searches up the tree for the defining instance of that class.

$\sigma_j(n, \mathsf{lookupType}, s) =$

$$\begin{cases} \texttt{IF } [\![strEq(s)(\text{“Object”})]\!] & \text{(Program).*} \\ \quad \texttt{THEN } [\![n]\!].\mathsf{parent.objectClassDecl} & \\ \quad \texttt{ELSE } ([\![n]\!].\mathsf{parent}.\_\mathsf{matchName}((.[\![s]\!], [\![n]\!].\mathsf{parent.classes})) & \\ \text{inherited} & \end{cases}$$

lookupMethod

The lookupMethod attribute takes a parameter (the name of a method) and searches up the tree for the defining instance of that method.

$$\sigma_j(n,\mathsf{lookupMethod},s) = \begin{cases} [\![n_{null}]\!] & s = \text{“Object”} \\ [\![n]\!].\_\mathsf{matchName}(\mathsf{(}.[\![s]\!]\mathsf{,}\ [\![n]\!].\mathsf{methods)}) & \text{(ClassDecl)} \\ \ \mathtt{IFNULL}\ [\![n]\!].\mathsf{superClass.lookupMethod(}[\![s]\!]\mathsf{)} & \\ [\![n_{null}]\!] & \text{otherwise} \end{cases}$$

remoteLookup

The remoteLookup attribute takes a parameter (the name of a variable) and searches up the inheritance chain to find the defining instance of that variable.

$$\sigma_j(n,\mathsf{remoteLookup},s) = \begin{cases} [\![n_{null}]\!].\_\mathsf{matchName}(\mathsf{(}.[\![s]\!]\mathsf{,}\ [\![n]\!].\mathsf{args)}) & \text{(ClassDecl)} \\ \ \mathtt{IFNULL}\ [\![n]\!].\mathsf{superClass.remoteLookup(}[\![s]\!]\mathsf{)} & \\ [\![n_{null}]\!] & \text{otherwise} \end{cases}$$

superClass

The superClass attribute returns a reference to a class's superclass.

$$\sigma_j(n,\mathsf{superClass}) = \begin{cases} [\![n_{null}]\!] & n = n_{null} \\ \mathtt{IF}\ [\![\mathit{strEq}(\text{“Object”})]\!](\mathsf{[}\![n]\!]\mathsf{.name)} & \text{otherwise} \\ \ \mathtt{THEN}\ [\![n_{null}]\!] & \\ \ \mathtt{ELSE}\ [\![n]\!].\mathsf{extUse.decl} & \end{cases}$$

enclosingClassDecl

The enclosingClassDecl returns a reference to the class that contains the current node.

$$\sigma_j(n,\mathsf{enclosingClassDecl}) = \begin{cases} [\![n]\!].\mathsf{parent} \quad \text{(ClassDecl).methods*} \\ \text{inherited} \end{cases}$$

thisFPDecl

The thisFPDecl attribute creates a new subtree that is a *FPDecl*, referencing the current class, representing the keyword “this”.

$$\begin{aligned} \sigma_j(n_p,\mathsf{thisFPDecl}) = \mathtt{MK}\lambda n,[&(\mathsf{nodeType},[\![\mathit{FPDecl}]\!]),\\ &(\mathsf{parent},[\![n_p]\!]),\\ &(\mathsf{nameDef},[\![n]\!].\mathsf{buildIdnDef(}[\![\text{“this”}]\!]\mathsf{)})\\ &(\mathsf{typeUse},[\![n]\!].\mathsf{buildTypeUse(}[\![n_p]\!].\mathsf{name)})] \end{aligned}$$

objectClassDecl

The objectClassDecl attribute returns (and maybe constructs) a special "Object" class declaration.

$$\sigma_j(n, \mathsf{objectClassDecl}) = [\![n]\!].\mathsf{buildObjectClassDecl}$$

Aux Node Construction Attributes

$$\sigma_j(n_p, \mathsf{buildTypeUse}, s) = \mathtt{MK}\lambda n, [(\mathsf{nodeType}, [\![\mathrm{TypeUse}]\!]), (\mathsf{name}, [\![s]\!]), (\mathsf{parent}, [\![n_p]\!])]$$

$$\sigma_j(n_p, \mathsf{buildIdnDef}, s) = \mathtt{MK}\lambda n, [(\mathsf{nodeType}, [\![\mathrm{IdnDef}]\!], (\mathsf{name}, [\![s]\!]), (\mathsf{parent}, [\![n_p]\!])]$$

$$\begin{aligned}\sigma_j(n_p, \mathsf{buildObjectClassDecl}) = \mathtt{MK}\lambda n, [&(\mathsf{nodeType}, [\![\mathit{ClassDecl}]\!]),\\ &(\mathsf{name}, [\![\text{“Object”}]\!]),\\ &(\mathsf{extUse}, [\![n]\!].\mathsf{buildTypeUse}([\![\text{“”}]\!])),\\ &(\mathsf{ctor}, [\![n]\!].\mathsf{buildObjectCtorDecl}),\\ &(\mathsf{args}, [\![[]]\!]),\\ &(\mathsf{methods}, [\![[]]\!]),\\ &(\mathsf{parent}, [\![n_p]\!])]\end{aligned}$$

$$\begin{aligned}\sigma_j(n_p, \mathsf{buildObjectCtorDecl}) = \mathtt{MK}\lambda n, [&(\mathsf{nodeType}, [\![\mathit{ClassDecl}]\!]),\\ &(\mathsf{name}, [\![\text{“Object”}]\!]),\\ &(\mathsf{args}, [\![[]]\!]),\\ &(\mathsf{supers}, [\![[]]\!]),\\ &(\mathsf{inits}, [\![[]]\!]),\\ &(\mathsf{parent}, [\![n_p]\!])]\end{aligned}$$

Helper Attributes

$$\sigma_j(n, \_\mathsf{matchName}, (s,l)) = \begin{cases} [\![n_{null}]\!].\_\mathsf{matchName}([\![(s,l)]\!]) & n \neq n_{null} \\ [\![n_{null}]\!] & l = [] \\ \\ \mathtt{IF}\ [\![\mathit{strEq}]\!]([\![\mathit{fst}(l)]\!].\mathsf{getName})([\![s]\!]) & \text{otherwise} \\ \ \mathtt{THEN}\ [\![\mathit{fst}(l)]\!] \\ \ \mathtt{ELSE}\ [\![n]\!].\_\mathsf{matchName}([\![(s, \mathit{rest}(l))]\!]) \end{cases}$$

$$\sigma_j(n, \mathsf{getName}) = \begin{cases} [\![n]\!].\mathsf{nameDef}.\mathsf{name} & \text{(FPDecl)} \\ [\![n]\!].\mathsf{name} & \text{otherwise} \end{cases}$$

$$\sigma_j(n, \_\texttt{exprsMatchFPs}, (es,fps)) =$$
$$\begin{cases} [\![n_{null}]\!].\_\texttt{exprsMatchFPs}([\![(es,fps)]\!]) & n \neq n_{null} \\ [\![false]\!] & len(es) \neq len(fps) \\ [\![true]\!] & es = fps = [] \\ \texttt{IF } [\![fst(es)]\!].\texttt{type.subTypeOf}([\![fst(fps)]\!].\texttt{typeUse.name}) & \text{otherwise} \\ \quad \texttt{THEN } [\![n]\!].\_\texttt{exprsMatchFPs}([\![(rest(es), rest(fps))]\!]) & \\ \quad \texttt{ELSE } [\![false]\!] & \end{cases}$$

7.3 Important Lemmas

The key theorem we aim to prove is Theorem 61, which says that the type attribute will evaluate to identical results under σ_k and σ_j. To prove this, we must first prove that a number of other attributes evaluate to identical results under σ_k and σ_j. To make this process smoother, we first prove some related lemmas. We are working with the higher order calculus for this entire chapter.

Definition 7.3.1. We have two context functions σ_a and σ_b. We say there is some set of attributes $A_{a\equiv b}$, for which any attribute is evaluationally equivalent in σ_a and σ_b. This means that for any attribute $a \in A_{a\equiv b}$, we have:

$$\forall(n \in \boldsymbol{N}), (v_1 \in \rho(a)), (t \in \boldsymbol{T}), (v_2 \in t),$$
$$\sigma_a \vdash [\![n]\!].a([\![v_1]\!]) \longrightarrow^* \_ \vdash [\![v_2]\!] \iff \sigma_b \vdash [\![n]\!].a([\![v_1]\!]) \longrightarrow^* \_ \vdash [\![v_2]\!]$$

Lemma 37. If there is some expression e which, when evaluating under either σ_a and σ_b, will only reference attributes in $A_{a\equiv b}$, then we know that this expression will evaluate to the same value under both context functions.

$$\forall(\sigma_a, \sigma_a', \sigma_b \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),$$
$$\forall(e \in \boldsymbol{E}), (t \in \boldsymbol{T}), (v \in t),$$
$$e, \text{ evaluated under } \sigma_a \text{ or } \sigma_b, \text{ only references attributes in } A_{a\equiv b} \tag{37.1}$$
$$\sigma_a \vdash e \longrightarrow^* \sigma_a' \vdash [\![v]\!] \tag{37.2}$$
$$\implies \sigma_b \vdash e \longrightarrow^* \_ \vdash [\![v]\!] \tag{37.3}$$

Proof. We proceed by induction on the expression e. This gives us six cases to prove. The value and node construction cases are trivial. The conditional, function application, and cache cases are proven using induction and multistep equivalence (Theorem 35), and have similar proofs, so we will only show a proof for the function application case. We will also show the attribution case, as this is the crux of the lemma.

For each case below, we give the proof state at the start of an induction case, and we add hypotheses with explanations for their derivations below them, in {braces}[2].

Case 37.1 (e is a function application expression)**.**

$$\sigma_a \vdash e_1(e_2) \longrightarrow^* \sigma_{a3} \vdash [\![v]\!] \tag{37.4}$$

{the above is (37.2), with e expanded}

$$\sigma_a \vdash e_1 \longrightarrow^* \sigma_{a2} \vdash [\![v_f]\!] \tag{37.5}$$

$$\sigma_{a2} \vdash e_2 \longrightarrow^* \sigma_{a3} \vdash [\![v_p]\!] \tag{37.6}$$

$$v = v_f(v_p) \tag{37.7}$$

{the above are given by big step expansion of (37.4)}

$$\sigma_b \vdash e_1 \longrightarrow^* \sigma_{b1} \vdash [\![v_f]\!] \tag{37.8}$$

$$\sigma_b \vdash e_2 \longrightarrow^* \sigma_{b2} \vdash [\![v_p]\!] \tag{37.9}$$

{the above are given by the inductive hypothesis and (37.5) and (37.6)}

$$\sigma_{b1} \vdash e_2 \longrightarrow^* \sigma_{b2}' \vdash [\![v_p]\!] \tag{37.10}$$

{from multistep equivalence for expressions with (37.8) and (37.9)}

$$\sigma_b \vdash e_1(e_2) \longrightarrow^* \sigma_{b2}' \vdash [\![v_f]\!]([\![v_p]\!]) \tag{37.11}$$

{from (37.8) and (37.10)}

$$\sigma_{b2}' \vdash [\![v_f]\!]([\![v_p]\!]) \longrightarrow \sigma_{b2}' \vdash [\![v_f(v_p)]\!] \tag{37.12}$$

{derived from FunApp }

$$\sigma_{b2}' \vdash [\![v_f]\!]([\![v_p]\!]) \longrightarrow \sigma_{b2}' \vdash [\![v]\!] \tag{37.13}$$

{(37.7) rewritten in (37.13)}

$$\sigma_b \vdash e_1(e_2) \longrightarrow^* \sigma_{b2}' \vdash [\![v]\!] \tag{37.14}$$

{by combining (37.11) and (37.13)}

$$\implies \sigma_b \vdash e_1(e_2) \longrightarrow^* \_ \vdash [\![v]\!] \tag{37.15}$$

The goal (37.15) is given by (37.14).

Case 37.2 (e is an attribution expression)**.**

$$\sigma_a \vdash e_1.a(e_2) \longrightarrow^* \sigma_{a4} \vdash [\![v]\!] \tag{37.16}$$

{the above is (37.2), with e expanded}

$$\sigma_a \vdash e_1 \longrightarrow^* \sigma_{a2} \vdash [\![n]\!] \tag{37.17}$$

$$\sigma_{a2} \vdash e_2 \longrightarrow^* \sigma_{a3} \vdash [\![v_p]\!] \tag{37.18}$$

{the above are given by big step expansion of (37.16), independent of whether value/cached/higher order is used}

$$\sigma_{a3} \vdash [\![n]\!].a([\![v_p]\!]) \longrightarrow^* \sigma_{a4} \vdash [\![v]\!] \tag{37.19}$$

{by subbing (37.17) and (37.18) into (37.16)}

$$\sigma_a \vdash [\![n]\!].a([\![v_p]\!]) \longrightarrow^* \sigma_{a4'} \vdash [\![v]\!] \tag{37.20}$$

[2]Note that this is different to the notation shown in Section 3.1.7, which uses (parentheses) and describes the derivation of a particular evaluation step, rather than explaining how a new hypothesis is justified.

{from multistep equivalence for expressions with (37.17), (37.18) and (37.20)}

$$\sigma_b \vdash [\![n]\!].a([\![v_p]\!]) \longrightarrow^* \sigma_{b4'} \vdash [\![v]\!] \tag{37.21}$$

{from (37.20), as a must be in $A_{a\equiv b}$ }

$$\sigma_b \vdash e_1 \longrightarrow^* \sigma_{b1} \vdash [\![n]\!] \tag{37.22}$$

$$\sigma_b \vdash e_2 \longrightarrow^* \sigma_{b2} \vdash [\![v_p]\!] \tag{37.23}$$

{the above are given by the inductive hypothesis and (37.17) and (37.18)}

$$\sigma_{b1} \vdash e_2 \longrightarrow^* \sigma_{b2'} \vdash [\![v_p]\!] \tag{37.24}$$

{from multistep equivalence for expressions with (37.22) and (37.23)}

$$\sigma_{b2'} \vdash [\![n]\!].a([\![v_p]\!]) \longrightarrow^* \sigma_{b4''} \vdash [\![v]\!] \tag{37.25}$$

{from multistep equivalence for expressions with (37.21), (37.22) and (37.24)}

$$\sigma_b \vdash e_1.a(e_2) \longrightarrow^* \sigma_{b4}'' \vdash [\![v]\!] \tag{37.26}$$

{by combining (37.22), (37.24) and (37.25)}

$$\implies \sigma_b \vdash e_1.a(e_2) \longrightarrow^* \_ \vdash [\![v]\!] \tag{37.27}$$

The goal (37.27) is given by (37.26).

All cases are satisfied, so Lemma 37 is proven. □

Lemma 38. We have some attribute a for which two context functions σ_a and σ_b return identical attribute equation expressions for any node and parameter inputs. If the evaluation of this attribute only references attributes in $A_{a\equiv b}$, then $a \in A_{a\equiv b}$.

$$\forall(\sigma_a, \sigma_b \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}), (a \in \boldsymbol{A}),$$

$$(\forall(n \in \boldsymbol{N}), (v_p \in \rho(a)), \quad \sigma_a(n, a, v_p) = \sigma_b(n, a, v_p)) \tag{38.1}$$

$$\sigma_a(\_, a, \_), \text{ evaluated under } \sigma_a \text{ or } \sigma_b, \text{ only references attributes in } A_{a\equiv b} \tag{38.2}$$

$$\implies a \in A_{a\equiv b} \tag{38.3}$$

Proof. Since we are proving a bidirectional implication, we must prove one direction at a time. We expand the bidirectional implication, beginning with the left-to-right case.

$$\forall(\sigma_a, \sigma_a', \sigma_b \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}),$$

$$\forall(n \in \boldsymbol{N}), (a \in \boldsymbol{A}), (v_1 \in \rho(a)), (t \in \boldsymbol{T}), (v_2 \in t),$$

$$(\forall(n \in \boldsymbol{N}), (v_p \in \rho(a)), \quad \sigma_a(n, a, v_p) = \sigma_b(n, a, v_p)) \tag{38.4}$$

$$\sigma_a(\_, a, \_), \text{ evaluated under } \sigma_a \text{ or } \sigma_b, \text{ only references attributes in } A_{a\equiv b} \tag{38.5}$$

$$\sigma_a \vdash [\![n]\!].a([\![v_1]\!]) \longrightarrow^* \sigma_a' \vdash [\![v_2]\!] \tag{38.6}$$

$$\implies \sigma_b \vdash [\![n]\!].a([\![v_1]\!]) \longrightarrow^* \_ \vdash [\![v_2]\!] \tag{38.7}$$

Expanding the first single step in (38.6), there are three possible derivations: AttrFetchValue, AttrFetchCached, and AttrFetchHO. We will consider these three cases individually.

Case 38.1 (The AttrFetchValue case)**.**

$$(\forall(n \in \boldsymbol{N}), (v_p \in \rho(a)), \quad \sigma_a(n,a,v_p) = \sigma_b(n,a,v_p)) \tag{38.8}$$

$$\sigma_a(\_,a,\_), \text{ evaluated under } \sigma_a \text{ or } \sigma_b, \text{ only references attributes in } A_{a\equiv b} \tag{38.9}$$

$$\sigma_a \vdash [\![n]\!].a([\![v_1]\!]) \longrightarrow^* \sigma_a' \vdash [\![v_2]\!] \tag{38.10}$$

{the above are unchanged}

$$\sigma_a \vdash [\![n]\!].a([\![v_1]\!]) \longrightarrow \sigma_a \vdash [\![v]\!] \tag{38.11}$$

{the first single step in (38.10) is derived from AttrFetchValue }

$$\sigma_a(n,a,v_p) = [\![v]\!] \tag{38.12}$$

{a requisite for (38.11)}

$$\sigma_a \vdash [\![v]\!] \longrightarrow^* \sigma_a' \vdash [\![v_2]\!] \tag{38.13}$$

{the rest of (38.10)}

$$v_2 = v \tag{38.14}$$

{as the only derivation for (38.13) is MultiRefl}

$$\sigma_b \vdash [\![n]\!].a([\![v_1]\!]) \longrightarrow \sigma_b \vdash [\![v]\!] \tag{38.15}$$

{derived from AttrFetchValue, using (38.8) rewritten in (38.12)}

$$\sigma_b \vdash [\![n]\!].a([\![v_1]\!]) \longrightarrow \sigma_b \vdash [\![v_2]\!] \tag{38.16}$$

{(38.15), rewritten by (38.14)}

$$\Longrightarrow \sigma_b \vdash [\![n]\!].a([\![v_1]\!]) \longrightarrow^* \_ \vdash [\![v_2]\!] \tag{38.17}$$

The goal (38.17) is provided by (38.16), as a single step can trivially be used to satisfy a multistep relation.

Case 38.2 (The AttrFetchCached case)**.**

$$(\forall(n \in \boldsymbol{N}), (v_p \in \rho(a)), \quad \sigma_a(n,a,v_p) = \sigma_b(n,a,v_p)) \tag{38.18}$$

$$\sigma_a(\_,a,\_), \text{ evaluated under } \sigma_a \text{ or } \sigma_b, \text{ only references attributes in } A_{a\equiv b} \tag{38.19}$$

$$\sigma_a \vdash [\![n]\!].a([\![v_1]\!]) \longrightarrow^* \sigma_a' \vdash [\![v_2]\!] \tag{38.20}$$

{the above are unchanged}

$$\sigma_a \vdash [\![n]\!].a([\![v_1]\!]) \longrightarrow \sigma_a \vdash n.a(v_1) := \sigma_a(n,a,v_p) \tag{38.21}$$

{the first single step in (38.20) is derived from AttrFetchCached }

$$\sigma_a(n,a,v_p) \text{ is not a value or } \texttt{MK} \text{ expression} \tag{38.22}$$

{a requisite for (38.21)}

$$\sigma_a \vdash n.a(v_1) := \sigma_a(n,a,v_p) \longrightarrow^* \sigma_a' \vdash [\![v_2]\!] \tag{38.23}$$

{the rest of (38.20)}

$$\sigma_b \vdash n.a(v_1) := \sigma_b(n,a,v_p) \longrightarrow^* \sigma_b'' \vdash [\![v_2]\!] \tag{38.24}$$

{from Lemma 37, given (38.19) and equality from (38.18)}

$$\sigma_b \vdash [\![n]\!].a([\![v_1]\!]) \longrightarrow \sigma_b \vdash n.a(v_1) := \sigma_b(n,a,v_p) \tag{38.25}$$

{derived from AttrFetchCached, using (38.22)}

$$\sigma_b \vdash [\![n]\!].a([\![v_1]\!]) \longrightarrow^* \sigma_b'' \vdash [\![v_2]\!] \tag{38.26}$$

{by combining (38.24) and (38.25)}

$$\Longrightarrow \sigma_b \vdash [\![n]\!].a([\![v_1]\!]) \longrightarrow^* \_ \vdash [\![v_2]\!] \tag{38.27}$$

The goal (38.27) is given by (38.26).

Case 38.3 (The AttrFetchHO case)**.**

$$(\forall(n \in \boldsymbol{N}), (v_p \in \rho(a)), \quad \sigma_a(n, a, v_p) = \sigma_b(n, a, v_p)) \tag{38.28}$$

$$\sigma_a(\_, a, \_), \text{ evaluated under } \sigma_a \text{ or } \sigma_b, \text{ only references attributes in } A_{a\equiv b} \tag{38.29}$$

$$\sigma_a \vdash [\![n]\!] \,.\, a([\![v_1]\!]) \longrightarrow^* \sigma_a' \vdash [\![v_2]\!] \tag{38.30}$$

{the above are unchanged}

$$\sigma_a \vdash [\![n]\!] \,.\, a([\![v_1]\!]) \longrightarrow \sigma_a \oplus \{(n, a, v_1) \mapsto [\![n_2]\!]\} \otimes n_2/f_l(n_2) \vdash [\![n_2]\!] \tag{38.31}$$

{the first single step in (38.30) is derived from AttrFetchHO }

$$\sigma_a(n, a, v_p) = \texttt{MK}f_l \tag{38.32}$$

$$n_2 \text{ does not exist in } \sigma_a \tag{38.33}$$

{the above are requisites for (38.31)}

$$v_2 = n_2 \tag{38.34}$$

{ as $[\![n_2]\!]$ must now step to $[\![v_2]\!]$}

$$n_3 \text{ does not exist in } \sigma_b \tag{38.35}$$

{by selecting some appropriate n_3}

$$\sigma_b \vdash [\![n]\!] \,.\, a([\![v_1]\!]) \longrightarrow \sigma_b \oplus \{(n, a, v_1) \mapsto [\![n_3]\!]\} \otimes n_3/f_l(n_3) \vdash [\![n_3]\!] \tag{38.36}$$

{derived from AttrFetchHO, using (38.33) and (38.32) rewritten by (38.28)}

$$\implies \sigma_b \vdash [\![n]\!] \,.\, a([\![v_1]\!]) \longrightarrow^* \_ \vdash [\![v_2]\!] \tag{38.37}$$

The goal (38.37) is given by (38.36), if we can prove $v_2 = n_3$. We already know $v_2 = n_2$ from (38.34), so what we need is $n_3 = n_2$. Given that we are satisfied with equality modulo renaming, we can say that n_3 and n_2 are the same node, as each has had its attributes specified using the same list of attribute expressions provided by f_l.

As all cases have been satisfied, Lemma 38 is proven. □

Lemma 39. We have some attribute a, which produces identical expressions in two context functions σ_a and σ_b, for any inputs. If the evaluation of this attribute only references attributes in $A_{a\equiv b}$ *and itself*, then a is also in $A_{a\equiv b}$. This lemma is very similar to Lemma 38, except that we now allow recursive calls to the same attribute.

$$\forall(\sigma_a, \sigma_b \in \boldsymbol{N} \to (a : \boldsymbol{A}) \to \rho(a) \to \boldsymbol{E}), (a \in \boldsymbol{A}),$$

$$(\forall(n \in \boldsymbol{N}), (v_p \in \rho(a)), \quad \sigma_a(n, a, v_p) = \sigma_b(n, a, v_p)) \tag{39.1}$$

$$\sigma_a(\_, a, \_), \text{ evaluated under } \sigma_a \text{ or } \sigma_b, \text{ only references attributes in } A_{a\equiv b} \text{ or } a \tag{39.2}$$

$$\implies a \in A_{a\equiv b} \tag{39.3}$$

Proof. Membership in $A_{a\equiv b}$ requires bidirectional implication of terminating evaluation. If some attribute terminates evaluation in one context, then it must terminate evaluation with the same value in the other context. If (for some input node and parameter) evaluation of the attribute *a always* references itself, then evaluation will never terminate. In this case, the requirement for $A_{a\equiv b}$ is met, via *ex falso quodlibet*. If evaluation does not always reference itself for some input node and parameter, then there is some sequence of self-calls that eventually does not self-call.

We will proceed by induction on this sequence (or tree) of recursive calls. The base case is an evaluation of a that never references a, and the inductive case is an evaluation of a, where the theorem holds for any subsequent evaluation of a.

Case 39.1 (e does not reference a)**.**

$$(\forall(n \in \boldsymbol{N}), (v_p \in \rho(a)), \quad \sigma_a(n, a, v_p) = \sigma_b(n, a, v_p)) \tag{39.4}$$

$$\sigma_a(\_, a, \_), \text{ evaluated under } \sigma_a \text{ or } \sigma_b, \text{ only references attributes in } A_{a\equiv b} \text{ or } a \tag{39.5}$$

$$\sigma_a \vdash [\![n]\!].a(v_1) \longrightarrow^* \_ \vdash [\![v]\!] \tag{39.6}$$

{the above are unchanged}

$$[\![n]\!].a(v_1) \text{ does not reference } a \tag{39.7}$$

{this is the inductive base case}

$$[\![n]\!].a(v_1) \text{ only references attributes in } A_{a\equiv b} \tag{39.8}$$

{(39.5), with the a case eliminated by (39.7)}

$$\implies \sigma_b \vdash [\![n]\!].a(v_1) \longrightarrow^* \_ \vdash [\![v]\!] \tag{39.9}$$

The goal (39.9) is given by Lemma 38 and (39.4), (39.6) and (39.8).

Case 39.2 (e references a)**.**

$$(\forall(n \in \boldsymbol{N}), (v_p \in \rho(a)), \quad \sigma_a(n, a, v_p) = \sigma_b(n, a, v_p)) \tag{39.10}$$

$$\sigma_a(\_, a, \_), \text{ evaluated under } \sigma_a \text{ or } \sigma_b, \text{ only references attributes in } A_{a\equiv b} \text{ or } a \tag{39.11}$$

$$\sigma_a \vdash [\![n]\!].a(v_1) \longrightarrow^* \_ \vdash [\![v]\!] \tag{39.12}$$

{the above are unchanged}

$$\text{further calls to } a \text{ in } [\![n]\!].a(v_1) \text{ will behave identically in } \sigma_a \text{ and } \sigma_b \tag{39.13}$$

{the inductive hypothesis}

$$\implies \sigma_b \vdash [\![n]\!].a(v_1) \longrightarrow^* \_ \vdash [\![v]\!] \tag{39.14}$$

(39.13) means that, for further calls to a, a will act as though it is in $A_{a\equiv b}$. If $a \in A_{a\equiv b}$, Lemma 38 is sufficient to prove the rest of this theorem.

Both cases have been satisfied, so Lemma 39 is proven. □

7.4 Featherweight Java

Now that we have defined the kind of tree we are working with, we specified the attributes in σ_k and σ_j, and we have defined some useful lemmas, we can begin to prove our core theorem: that name and type analysis in each implementation is equivalent. First, we will more strictly define the context functions we start evaluation under.

We begin with a blank context function, which we will call σ_0. For every input, a value is returned, which is the default for the attribute being provided. We don't care what these defaults are, we only care that σ_0 always returns value expressions.

$$\sigma_0 = \text{the blank context}$$

Now we define σ_{tree}, which describes a Featherweight Java program tree, conforming to the abstract grammar given in Figure 7.3, rooted at the node n_0. For attributes and nodes orthogonal to this tree definition, the definition from σ_0 are used.

$$\sigma_{tree} = \text{the tree only}$$

Now we define σ_k, which is the same as σ_{tree} except that all extrinsic attributes described in Section 7.2.1 are also present. We also define σ_j, which is the same as σ_{tree} except that all extrinsic attributes described in Section 7.2.2 are also present. The context functions σ_k and σ_j are the starting points for our analysis.

$$\sigma_k = \text{the tree and Kiama attributes}$$

$$\sigma_k = \text{the tree and JastAdd attributes}$$

Throughout this chapter we will sometimes refer to σ_k as "the Kiama implementation" and σ_j as "the JastAdd implementation".

7.4.1 Attributes in $A_{k\equiv j}$

We know that all attribute described in the abstract syntax tree are intrinsic, and are defined in σ_{tree}. Since none of the attributes defined in Sections 7.2.1 and 7.2.2 override definitions of any of these attributes in σ_k or σ_j, we know that all attributes from the abstract syntax tree return identical value expressions in each implementation. Therefore we know that all of these attributes are in $A_{k\equiv j}$. Formally, we have the following.

$$(\mathsf{name}, \mathsf{classes}, \mathsf{expr}, \mathsf{exprs}, \mathsf{args}, \mathsf{ctor}, \mathsf{methods}, \mathsf{super}, \mathsf{inits}, \mathsf{parent}) \subset A_{k\equiv j}$$
$$(\mathsf{nameDef}, \mathsf{nameUse}, \mathsf{rtnUse}, \mathsf{varUse}, \mathsf{typeUse}, \mathsf{leftUse}, \mathsf{rightUse}, \mathsf{extUse}) \subset A_{k\equiv j}$$

We can now use Lemma 38 to include some more attributes in $A_{k\equiv j}$, as they are defined identically in both implementations, and all attributes they reference are already shown to be in $A_{k\equiv j}$. These attributes are as follows.

- getName, which references nameDef and name.
- buildTypeUse, which does not reference other attributes.
- buildIdnDef, which does not reference other attributes.
- thisFPDecl, which references name, buildIdnDef, and buildTypeUse.
- buildObjectCtorDecl, which does not reference other attributes.
- buildObjectClassDecl, which references buildTypeUse and buildObjectCtorDecl.
- _matchName (using Lemma 39), which references getName and _matchName.

Formally, we have the following.

$$(\mathsf{getName}, \mathsf{buildTypeUse}, \mathsf{buildIdnDef}, \mathsf{thisFPDecl}) \subset A_{k\equiv j}$$
$$(\mathsf{buildObjectCtorDecl}, \mathsf{buildObjectClassDecl}, \mathsf{\_matchName}) \subset A_{k\equiv j}$$

From this point on, we define various lemmas that prove the inclusion of various other attributes in $A_{k\equiv j}$.

Lemma 40 (objectClassDecl for Program nodes)**.** For any node n of type *Program*, we have the following.

$$\begin{aligned} &\forall(n \in \boldsymbol{N}), (t \in \boldsymbol{T}), (v \in t), \\ &\sigma_k \vdash [\![n]\!].\mathsf{objectClassDecl} \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff &\sigma_j \vdash [\![n]\!].\mathsf{objectClassDecl} \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Proof. We proceed by evaluating both expressions.

$$\begin{aligned} &\sigma_k \vdash [\![n]\!].\mathsf{objectClassDecl} \\ &\quad \{\text{as } n \text{ is a Program node}\} \\ &\longrightarrow [\![n]\!].\mathsf{buildObjectClassDecl} \end{aligned}$$

$$\begin{aligned} &\sigma_j \vdash [\![n]\!].\mathsf{objectClassDecl} \\ &\longrightarrow [\![n]\!].\mathsf{buildObjectClassDecl} \end{aligned}$$

As $\mathsf{buildObjectClassDecl} \in A_{k\equiv j}$, we know that both implementations will evaluate from this point to the same value. This satisfies the goal, so Lemma 40 is proven. □

Lemma 41 (_matchName with different nodes)**.** For any n_x and n_y, _matchName, when called with the same parameters, will evaluate to the same result.

$$\begin{aligned} &\forall(n_x, n_y \in \boldsymbol{N}), (t \in \boldsymbol{T}), (v \in t), (v_p \in (\mathit{listNode} \times \mathit{string})), \\ &\sigma_k \vdash [\![n_x]\!].\_\mathsf{matchName}([\![v_p]\!]) \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff &\sigma_j \vdash [\![n_y]\!].\_\mathsf{matchName}([\![v_p]\!]) \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Proof. We proceed by evaluating both expressions.

$$\begin{aligned} &\sigma_k \vdash [\![n_x]\!].\_\mathsf{matchName}([\![v]\!]) \\ &\longrightarrow [\![n_{null}]\!].\_\mathsf{matchName}([\![v]\!]) \end{aligned}$$

$$\begin{aligned} &\sigma_j \vdash [\![n_y]\!].\_\mathsf{matchName}([\![v]\!]) \\ &\longrightarrow [\![n_{null}]\!].\_\mathsf{matchName}([\![v]\!]) \end{aligned}$$

As _matchName $\in A_{k\equiv j}$, we know that both implementations will evaluate from this point to the same value. This satisfies the goal, so Lemma 41 is proven. □

Theorem 42 (decl and lookupType)**.** For any two nodes n_x and n_y, but with the same parameter s, decl in the Kiama implementation and lookupType in the JastAdd implementation will always evaluate to the same value.

$$\begin{aligned} &\forall (n_x, n_y \in \boldsymbol{N}), (s \in \mathit{string}), (t \in \boldsymbol{T}), (v \in t), \\ &\quad \sigma_k \vdash [\![n_x]\!].\mathsf{decl}([\![s]\!]) \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff &\ \sigma_j \vdash [\![n_y]\!].\mathsf{lookupType}([\![s]\!]) \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Proof. We proceed by considering separately the cases where s is and is not equal to the string "Object".

Case 42.1 (s = "Object")**.** We proceed by evaluating both expressions in parallel.

$\sigma_k \vdash [\![n_x]\!].\mathsf{decl}([\![s]\!])$

{as s = "Object"}

$\longrightarrow [\![n_x]\!].\mathsf{objectClassDecl}$

{via inheritance}

$\longrightarrow^* [\![n_0]\!].\mathsf{objectClassDecl}$

$\sigma_j \vdash [\![n_y]\!].\mathsf{lookupType}([\![s]\!])$

{where n_z is the nearest ancestor of n_y whose parent is the Program root n_0}

$\longrightarrow^*$ IF $[\![\mathit{strEq}(s)(\text{"Object"})]\!]$

THEN $[\![n_z]\!].\mathsf{parent}.\mathsf{objectClassDecl}$

ELSE $[\![n_z]\!].\mathsf{parent}.\_\mathsf{matchName}((.[\![s]\!], [\![n_z]\!].\mathsf{parent}.\mathsf{classes}))$

{as s = "Object"}

$\longrightarrow [\![n_z]\!].\mathsf{parent}.\mathsf{objectClassDecl}$

$\longrightarrow [\![n_0]\!].\mathsf{objectClassDecl}$

As objectClassDecl $\in A_{k \equiv j}$, we know that both implementations will evaluate from this point to the same value. This satisfies the goal, so this case is satisfied.

Case 42.2 ($s \neq$ "Object")**.** We proceed by evaluating both expressions in parallel.

$\sigma_k \vdash [\![n_x]\!].\mathsf{decl}([\![s]\!])$

{as $s \neq$ "Object", and via inheritance}

$\longrightarrow^* [\![n_0]\!].\_\mathsf{matchName}((.[\![s]\!], [\![n_0]\!].\mathsf{classes}))$

$\sigma_j \vdash [\![n_y]\!].\mathsf{lookupType}([\![s]\!])$

$$
\begin{aligned}
&\{\text{where } n_z \text{ is the nearest ancestor whose } \texttt{parent} \text{ is the Program root } n_0\}\\
\longrightarrow^* \ &\texttt{IF } [\![strEq(s)(\text{“Object”})]\!]\\
&\quad \texttt{THEN } [\![n_z]\!].\texttt{parent.objectClassDecl}\\
&\quad \texttt{ELSE } [\![n_z]\!].\texttt{parent.\_matchName((.}[\![s]\!], [\![n_z]\!].\texttt{parent.classes))}\\
&\qquad\qquad \{\text{as } s \neq \text{“Object”}\}\\
\longrightarrow\ &[\![n_z]\!].\texttt{parent.\_matchName((.}[\![s]\!], [\![n_0]\!].\texttt{parent.classes))}\\
\longrightarrow\ &[\![n_0]\!].\texttt{\_matchName((.}[\![s]\!], [\![n_z]\!].\texttt{parent.classes))}\\
\longrightarrow\ &[\![n_0]\!].\texttt{\_matchName((.}[\![s]\!], [\![n_0]\!].\texttt{classes))}
\end{aligned}
$$

At this point both implementations have stepped to an identical expression, which only contains the attributes _matchName and classes, which are both in $A_{k\equiv j}$. Therefore we can use Lemma 37 to satisfy this case.

As both cases have been satisfied, Theorem 42 is proven. □

Lemma 43 (decl and typeUse)**.** For any two nodes n_x and n, the particular pattern of uses of decl and typeUse shown below will always evaluate to the same value.

$$
\begin{aligned}
&\forall(n_x, n \in \boldsymbol{N}), (t \in \boldsymbol{T}), (v \in t),\\
&\sigma_k \vdash [\![n_x]\!].\texttt{decl(}[\![n]\!].\texttt{typeUse.name)} \longrightarrow^* \_ \vdash [\![v]\!]\\
\iff\ &\sigma_j \vdash [\![n]\!].\texttt{typeUse.decl} \longrightarrow^* \_ \vdash [\![v]\!]
\end{aligned}
$$

Proof. We proceed by evaluating both expressions. We say that $[\![n]\!].\texttt{typeUse}$ yields $[\![n_t]\!]$ and $[\![n_t]\!].\texttt{name}$ yields $[\![s_t]\!]$. We know this because both typeUse and name are in $A_{k\equiv j}$. We also know that n_t must be of type *TypeUse*, as per the abstract grammar in Figure 7.3.

$$
\begin{aligned}
&\sigma_k \vdash [\![n_x]\!].\texttt{decl(}[\![n]\!].\texttt{typeUse.name)}\\
&\longrightarrow [\![n_x]\!].\texttt{decl(}[\![n_t]\!].\texttt{name)}\\
&\longrightarrow [\![n_x]\!].\texttt{decl(}[\![s_t]\!]\texttt{)}
\end{aligned}
$$

$$
\begin{aligned}
&\sigma_j \vdash [\![n]\!].\texttt{typeUse.decl}\\
&\longrightarrow [\![n_t]\!].\texttt{decl}\\
&\qquad \{\text{as } n_t \text{ must be a } \textit{TypeUse} \text{ node}\}\\
&\longrightarrow [\![n_t]\!].\texttt{lookupType(}[\![n_t]\!].\texttt{name)}\\
&\longrightarrow [\![n_t]\!].\texttt{lookupType(}[\![s_t]\!]\texttt{)}
\end{aligned}
$$

At this point both implementations have stepped to expressions that allow the theorem to be proven by Theorem 42. □

Theorem 44 (superClass is in $A_{k\equiv j}$)**.**

$$\begin{aligned} &\forall (n \in \boldsymbol{N}), (t \in \boldsymbol{T}), (v \in t), \\ &\sigma_k \vdash [\![n]\!].\mathsf{superClass} \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff\ &\sigma_j \vdash [\![n]\!].\mathsf{superClass} \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Proof. We separately consider the cases where n is and is not equal to n_{null}. If n is equal to n_{null}, both context functions will return $[\![n_{null}]\!]$, and the theorem is proven trivially.

We say that $[\![n]\!].\mathsf{name}$ evaluates to some value $[\![s]\!]$. We now consider the cases where s is and is not equal to "Object".

Case 44.1 (s = "Object")**.** We proceed by evaluating both expressions in parallel.

$$\begin{aligned} &\sigma_k \vdash [\![n]\!].\mathsf{superClass} \\ &\qquad\qquad \{\text{as } n \neq n_{null}\} \\ &\longrightarrow \mathtt{IF}\ [\![strEq(\text{“Object”})]\!]([\![n]\!].\mathsf{name}) \\ &\qquad \mathtt{THEN}\ [\![n_{null}]\!] \\ &\qquad \mathtt{ELSE}\ [\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{extUse}.\mathsf{name}) \\ &\longrightarrow^* \mathtt{IF}\ [\![strEq(\text{“Object”})]\!]([\![s]\!]) \\ &\qquad \mathtt{THEN}\ [\![n_{null}]\!] \\ &\qquad \mathtt{ELSE}\ [\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{extUse}.\mathsf{name}) \\ &\qquad\qquad \{\text{as } s = \text{“Object”}\} \\ &\longrightarrow^* [\![n_{null}]\!] \end{aligned}$$

$$\begin{aligned} &\sigma_j \vdash [\![n]\!].\mathsf{superClass} \\ &\qquad\qquad \{\text{as } n \neq n_{null}\} \\ &\longrightarrow \mathtt{IF}\ [\![strEq(\text{“Object”})]\!]([\![n]\!].\mathsf{name}) \\ &\qquad \mathtt{THEN}\ [\![n_{null}]\!] \\ &\qquad \mathtt{ELSE}\ [\![n]\!].\mathsf{extUse}.\mathsf{decl} \\ &\longrightarrow^* \mathtt{IF}\ [\![strEq(\text{“Object”})]\!]([\![s]\!]) \\ &\qquad \mathtt{THEN}\ [\![n_{null}]\!] \\ &\qquad \mathtt{ELSE}\ [\![n]\!].\mathsf{extUse}.\mathsf{decl} \\ &\qquad\qquad \{\text{as } s = \text{“Object”}\} \\ &\longrightarrow^* [\![n_{null}]\!] \end{aligned}$$

Both implementations have stepped to the same value, so this case is satisfied.

Case 44.2 ($s \neq$ "Object")**.** We proceed by evaluating both expressions in parallel. We say that $[\![n]\!].\mathsf{extUse}$ yields $[\![n_2]\!]$ and $[\![n_2]\!].\mathsf{name}$ yields $[\![s_2]\!]$. We know this because both extUse and name are in $A_{k\equiv j}$. We also know that n_2 must be of type *TypeUse*, as per the abstract grammar in Figure 7.3.

$\sigma_k \vdash [\![n]\!].\mathsf{superClass}$

$\{\text{as } n \neq n_{null}\}$

$\longrightarrow \mathtt{IF}\ [\![strEq(\text{“Object”})]\!]([\![n]\!].\mathsf{name})$

$\mathtt{THEN}\ [\![n_{null}]\!]$

$\mathtt{ELSE}\ [\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{extUse}.\mathsf{name})$

$\longrightarrow^* \mathtt{IF}\ [\![strEq(\text{“Object”})]\!]([\![s]\!])$

$\mathtt{THEN}\ [\![n_{null}]\!]$

$\mathtt{ELSE}\ [\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{extUse}.\mathsf{name})$

$\{\text{as } s \neq \text{“Object”}\}$

$\longrightarrow [\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{extUse}.\mathsf{name})$

$\longrightarrow [\![n]\!].\mathsf{decl}([\![n_2]\!].\mathsf{name})$

$\longrightarrow [\![n]\!].\mathsf{decl}([\![s_2]\!])$

$\sigma_j \vdash [\![n]\!].\mathsf{superClass}$

$\{\text{as } n \neq n_{null}\}$

$\longrightarrow \mathtt{IF}\ [\![strEq(\text{“Object”})]\!]([\![n]\!].\mathsf{name})$

$\mathtt{THEN}\ [\![n_{null}]\!]$

$\mathtt{ELSE}\ [\![n]\!].\mathsf{extUse}.\mathsf{decl}$

$\longrightarrow^* \mathtt{IF}\ [\![strEq(\text{“Object”})]\!]([\![s]\!])$

$\mathtt{THEN}\ [\![n_{null}]\!]$

$\mathtt{ELSE}\ [\![n]\!].\mathsf{extUse}.\mathsf{decl}$

$\{\text{as } s \neq \text{“Object”}\}$

$\longrightarrow [\![n]\!].\mathsf{extUse}.\mathsf{decl}$

$\{\text{as } n_2 \text{ must be a } \mathit{TypeUse} \text{ node}\}$

$\longrightarrow [\![n_2]\!].\mathsf{decl}$

$\longrightarrow [\![n_2]\!].\mathsf{lookupType}([\![n_2]\!].\mathsf{name})$

$\longrightarrow [\![n_2]\!].\mathsf{lookupType}([\![s_2]\!])$

At this point both implementation have stepped to expressions that allow the theorem to be proven by Theorem 42.

Therefore Theorem 44 is proven. □

Theorem 45 (fields is in $A_{k\equiv j}$)**.**

$$\begin{aligned} &\forall(n \in \boldsymbol{N}), (t \in \boldsymbol{T}), (v \in t), \\ &\quad \sigma_k \vdash [\![n]\!].\mathsf{fields} \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff &\quad \sigma_j \vdash [\![n]\!].\mathsf{fields} \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Proof. This theorem is given immediately by Lemma 39. Both context functions return the same expression for the fields attribute, and that expression only contains references to itself and attributes in $A_{k\equiv j}$ (including superClass, which is in $A_{k\equiv j}$ according to Theorem 44). □

Theorem 46 (subTypeOf is in $A_{k\equiv j}$)**.**

$$\begin{aligned} &\forall(n \in \boldsymbol{N}), (s \in \mathit{string}), (t \in \boldsymbol{T}), (v \in t), \\ &\quad \sigma_k \vdash [\![n]\!].\mathsf{subTypeOf}([\![s]\!]) \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff &\quad \sigma_j \vdash [\![n]\!].\mathsf{subTypeOf}([\![s]\!]) \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Proof. σ_k and σ_j share the same definition of subTypeOf. The expression returned by each context function will always be the same, independent of properties of s or n. This expression references the attributes name, superClass, and subTypeOf. name is known to be in $A_{k\equiv j}$, Theorem 44 proves that superClass is in $A_{k\equiv j}$, and subTypeOf is a recursive call to the same attribute. Lemma 39 therefore provides $\mathsf{subTypeOf} \in A_{k\equiv j}$, and Theorem 46 is proven. □

Lemma 47 (findFP with *cons* and *concat*)**.** Assuming $n_h \neq n_{null}$, we have the following. Note that we are not comparing the JastAdd and Kiama implementations here; both relations use the Kiama implementation.

$$\begin{aligned} &\forall(n, n_h \in \boldsymbol{N}), (l_r \in \mathit{listNode}), (e_2, e_s \in \boldsymbol{E}), (t \in \boldsymbol{T}), (v \in t), \\ &\quad \sigma_k \vdash [\![n]\!].\mathsf{findFP}((([\![\mathit{concat}]\!]([\![n_h :: l_r]\!])(e_2), e_s)) \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff &\quad \sigma_k \vdash [\![n]\!].\mathsf{findFP}(([\![[n_h]]\!], e_s)) \\ &\qquad \mathtt{IFNULL}\ [\![n]\!].\mathsf{findFP}((([\![\mathit{concat}]\!]([\![l_r]\!])(e_2), e_s)) \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Proof. We proceed by evaluating both expressions in parallel. As both evaluations use the same context function σ_k, we can assume that evaluating identical expressions will yield the same result. We say that e_2 evaluates to some value $[\![l_2]\!]$, that $[\![n_h]\!].\mathsf{getName}$ evaluates to some value $[\![s_h]\!]$, and that e_s evaluates to some value $[\![s]\!]$.

$$\begin{aligned} &\sigma_k \vdash [\![n]\!].\mathsf{findFP}((([\![\mathit{concat}]\!]([\![n_h :: l_r]\!])(e_2), e_s)) \\ &\longrightarrow^* [\![n]\!].\mathsf{findFP}((([\![\mathit{concat}]\!]([\![n_h :: l_r]\!])([\![l_2]\!]), e_s)) \\ &\longrightarrow^* [\![n]\!].\mathsf{findFP}(([\![\mathit{concat}(n_h :: l_r)(l_2)]\!], [\![s]\!])) \end{aligned}$$

$$
\begin{aligned}
&\qquad\qquad\qquad\qquad\{\text{via manipulation of underlying terms}\}\\
&= [\![n]\!].\mathsf{findFP}(([\![n_h :: concat(l_r)(l_2)]\!], e_s))\\
&\longrightarrow^* [\![n]\!].\mathsf{findFP}(([\![n_h :: concat(l_r)(l_2)]\!], [\![s]\!]))\\
&\longrightarrow^* [\![n_{null}]\!].\_\mathsf{matchName}([\![(s, n_h :: concat(l_r)(l_2))]\!])\\
&\longrightarrow \mathtt{IF}\ [\![strEq]\!]([\![n_h]\!].\mathsf{getName})([\![s]\!])\ \mathtt{THEN}\ [\![n_h]\!]\\
&\qquad \mathtt{ELSE}\ [\![n_{null}]\!].\_\mathsf{matchName}([\![(s, concat(l_r)(l_2))]\!])\\
&\longrightarrow^* \mathtt{IF}\ [\![strEq]\!]([\![s_h]\!])([\![s]\!])\ \mathtt{THEN}\ [\![n_h]\!]\\
&\qquad \mathtt{ELSE}\ [\![n_{null}]\!].\_\mathsf{matchName}([\![(s, concat(l_r)(l_2))]\!])
\end{aligned}
$$

$$
\begin{aligned}
&\sigma_k \vdash [\![n]\!].\mathsf{findFP}(([\![[n_h]]\!], e_s))\ \mathtt{IFNULL}\ [\![n]\!].\mathsf{findFP}(([\![concat]\!]([\![l_r]\!])(e_2), e_s))\\
&\longrightarrow^* \vdash [\![n]\!].\mathsf{findFP}(([\![[n_h]]\!], [\![s]\!]))\ \mathtt{IFNULL}\ [\![n]\!].\mathsf{findFP}(([\![concat]\!]([\![l_r]\!])(e_2), e_s))\\
&\longrightarrow^* [\![n_{null}]\!].\_\mathsf{matchName}([\![(s, [n_h])]\!])\ \mathtt{IFNULL}\ [\![n]\!].\mathsf{findFP}(([\![concat]\!]([\![l_r]\!])(e_2), e_s))\\
&\longrightarrow^* (\mathtt{IF}\ [\![strEq]\!]([\![n_h]\!].\mathsf{getName})([\![s]\!])\ \mathtt{THEN}\ [\![n_h]\!]\ \mathtt{ELSE}\ [\![n_{null}]\!].\_\mathsf{matchName}([\![(s, [])]\!]))\\
&\qquad \mathtt{IFNULL}\ [\![n]\!].\mathsf{findFP}(([\![concat]\!]([\![l_r]\!])(e_2), e_s))\\
&\longrightarrow^* (\mathtt{IF}\ [\![strEq]\!]([\![s_h]\!])([\![s]\!])\ \mathtt{THEN}\ [\![n_h]\!]\ \mathtt{ELSE}\ [\![n_{null}]\!].\_\mathsf{matchName}([\![(s, [])]\!]))\\
&\qquad \mathtt{IFNULL}\ [\![n]\!].\mathsf{findFP}(([\![concat]\!]([\![l_r]\!])(e_2), e_s))
\end{aligned}
$$

At this point we consider the case where $s = s_h$, and the case where $s \neq s_h$.

Case 47.1 ($s = s_h$)**.**

$$
\begin{aligned}
&\sigma_k \vdash \mathtt{IF}\ [\![strEq]\!]([\![s]\!])([\![s]\!])\ \mathtt{THEN}\ [\![n_h]\!]\ \mathtt{ELSE}\ [\![n_{null}]\!].\_\mathsf{matchName}([\![(s, concat(l_r)(l_2))]\!])\\
&\longrightarrow^* [\![n_h]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\sigma_k \vdash (\mathtt{IF}\ [\![strEq]\!]([\![s]\!])([\![s]\!])\ \mathtt{THEN}\ [\![n_h]\!]\ \mathtt{ELSE}\ [\![n_{null}]\!].\_\mathsf{matchName}([\![(s, [])]\!]))\\
&\qquad \mathtt{IFNULL}\ [\![n]\!].\mathsf{findFP}(([\![concat]\!]([\![l_r]\!])(e_2), e_s))\\
&\longrightarrow^* [\![n_h]\!]\ \mathtt{IFNULL}\ [\![n]\!].\mathsf{findFP}(([\![concat]\!]([\![l_r]\!])(e_2), e_s))\\
&\qquad\qquad\qquad\qquad\{\text{as } n_h \neq n_{null}\}\\
&\longrightarrow^* [\![n_h]\!]
\end{aligned}
$$

When $s = s_h$, both expressions will evaluate to $[\![n_h]\!]$, so this case is satisfied.

Case 47.2 ($s \neq s_h$)**.**

$$
\sigma_k \vdash \mathtt{IF}\ [\![strEq]\!]([\![s_h]\!])([\![s]\!])\ \mathtt{THEN}\ [\![n_h]\!]\ \mathtt{ELSE}\ [\![n_{null}]\!].\_\mathsf{matchName}([\![(s, concat(l_r)(l_2))]\!])
$$

$$\{\text{as } s \neq s_h\}$$
$$\longrightarrow^* [\![n_{null}]\!].\_\texttt{matchName}([\![(s, concat(l_r)(l_2))]\!])$$

$$\sigma_k \vdash (\texttt{IF } [\![strEq]\!]([\![s_h]\!])([\![s]\!]) \texttt{ THEN } [\![n_h]\!] \texttt{ ELSE } [\![n_{null}]\!].\_\texttt{matchName}([\![(s,[])]\!]))$$
$$\texttt{IFNULL } [\![n]\!].\texttt{findFP}(([\![concat]\!]([\![l_r]\!])(e_2), e_s))$$
$$\{\text{as } s \neq s_h\}$$
$$\longrightarrow^* [\![n_{null}]\!].\_\texttt{matchName}([\![(s,[])]\!])$$
$$\texttt{IFNULL } [\![n]\!].\texttt{findFP}(([\![concat]\!]([\![l_r]\!])(e_2), e_s))$$
$$\{\text{as } \_\texttt{matchName} \text{ returns } n_{null} \text{ for the empty list}\}$$
$$\longrightarrow^* [\![n_{null}]\!] \texttt{ IFNULL } [\![n]\!].\texttt{findFP}(([\![concat]\!]([\![l_r]\!])(e_2), e_s))$$
$$\longrightarrow^* [\![n]\!].\texttt{findFP}(([\![concat]\!]([\![l_r]\!])(e_2), e_s))$$
$$\longrightarrow^* [\![n]\!].\texttt{findFP}(([\![concat]\!]([\![l_r]\!])([\![l_2]\!]), e_s))$$
$$\longrightarrow^* [\![n]\!].\texttt{findFP}(([\![concat]\!]([\![l_r]\!])([\![l_2]\!]), [\![s]\!]))$$
$$\longrightarrow^* [\![n_{null}]\!].\_\texttt{matchName}([\![(s, concat(l_r)(l_2))]\!])$$

At this point, both instances have evaluated to the same expression. As we are evaluating under the same context, we can use multistep determinism to satisfy this case.

Both cases have been satisfied, so Lemma 47 is proven. □

Lemma 48. For any list of nodes l_1 that does not contain n_{null}, we have the following. Once again we are working with the same context function σ_k in both cases.

$$\forall (n \in \boldsymbol{N}), (l_1 \in listNode), (e_2, e_s \in \boldsymbol{E}), (t \in \boldsymbol{T}), (v \in t),$$
$$\sigma_k \vdash [\![n]\!].\texttt{findFP}(([\![concat]\!]([\![l_1]\!])(e_2), e_s)) \longrightarrow^* \_ \vdash [\![v]\!]$$
$$\iff \sigma_k \vdash [\![n]\!].\texttt{findFP}(([\![l_1]\!], e_s)) \texttt{ IFNULL } [\![n]\!].\texttt{findFP}((e_2, e_s)) \longrightarrow^* \_ \vdash [\![v]\!]$$

Proof. We proceed by induction on the list of nodes l_1. As both evaluations use the same context function σ_k, we can assume that evaluating identical expressions with yield the same result. We say that e_2 evaluates to some value l_2 and that e_s evaluates to value $[\![s]\!]$.

Case 48.1 (l_1 is the empty list)**.**

$$\sigma_k \vdash [\![n]\!].\texttt{findFP}(([\![concat]\!]([\![[]]\!])(e_2), e_s))$$
$$\longrightarrow^* [\![n]\!].\texttt{findFP}(([\![concat]\!]([\![[]]\!])([\![l_2]\!]), [\![s]\!]))$$
$$\longrightarrow^* [\![n]\!].\texttt{findFP}([\![(l_2, s)]\!])$$

$$\sigma_k \vdash [\![n]\!].\texttt{findFP}(([\![[]]\!], e_s)) \texttt{ IFNULL } [\![n]\!].\texttt{findFP}((e_2, e_s))$$
$$\longrightarrow^* [\![n]\!].\texttt{findFP}(([\![[]]\!], [\![s]\!])) \texttt{ IFNULL } [\![n]\!].\texttt{findFP}((e_2, e_s))$$

$$\{\text{as findFP returns } [\![n_{null}]\!] \text{ for the empty list}\}$$
$$\longrightarrow^* \overline{[\![n_{null}]\!]}\ \texttt{IFNULL}\ [\![n]\!].\texttt{findFP}((e_2, e_s))$$
$$\longrightarrow^* [\![n]\!].\texttt{findFP}((\underline{e_2}, \underline{e_s}))$$
$$\longrightarrow^* [\![n]\!].\texttt{findFP}((\underline{\overline{[\![l_2]\!]}, \overline{[\![s]\!]}}))$$
$$\longrightarrow^* [\![n]\!].\texttt{findFP}(\overline{[\![(l_2, s)]\!]})$$

At this point both evaluations have reached an identical expression. As the same context function is being used in both cases, the result is now given from multistep determinism.

Case 48.2 (l_1 has the form $n_h :: l_r$)**.** Our goal is now the following.

$$\sigma_k \vdash [\![n]\!].\texttt{findFP}(([\![concat]\!]([\![n_h :: l_r]\!])(e_2), e_s)) \longrightarrow^* \_ \vdash [\![v]\!]$$
$$\iff \sigma_k \vdash [\![n]\!].\texttt{findFP}(([\![n_h :: l_r]\!], e_s))\ \texttt{IFNULL}\ [\![n]\!].\texttt{findFP}((e_2, e_s)) \longrightarrow^* \_ \vdash [\![v]\!]$$

We can use Lemma 47 to replace the first term of this goal, obtaining the following new goal.

$$\sigma_k \vdash [\![n]\!].\texttt{findFP}(([\![[n_h]]\!], e_s))\ \texttt{IFNULL}\ [\![n]\!].\texttt{findFP}(([\![concat]\!]([\![l_r]\!])(e_2), e_s))$$
$$\iff \sigma_k \vdash [\![n]\!].\texttt{findFP}(([\![n_h :: l_r]\!], e_s))\ \texttt{IFNULL}\ [\![n]\!].\texttt{findFP}((e_2, e_s)) \longrightarrow^* \_ \vdash [\![v]\!]$$

By induction, we have the following.

$$\sigma_k \vdash [\![n]\!].\texttt{findFP}(([\![concat]\!]([\![l_r]\!])(e_2), e_s)) \longrightarrow^* \_ \vdash [\![v]\!]$$
$$\iff \sigma_k \vdash [\![n]\!].\texttt{findFP}(([\![l_r]\!], e_s))\ \texttt{IFNULL}\ [\![n]\!].\texttt{findFP}((e_2, e_s)) \longrightarrow^* \_ \vdash [\![v]\!]$$

The node n_h is the first item in l_1. As l_1 does not contain n_{null}, we know $n_h \neq n_{null}$. We say that $[\![n_h]\!].\texttt{getName}$ evaluates to some value $[\![s_h]\!]$. Now we consider separately the cases where s_h is and is not equal to s.

Case 48.2.1 ($s_h = s$)**.**

$$\sigma_k \vdash [\![n]\!].\texttt{findFP}(([\![[n_h]]\!], \underline{e_s}))\ \texttt{IFNULL}\ [\![n]\!].\texttt{findFP}(([\![concat]\!]([\![l_r]\!])(e_2), e_s))$$
$$\longrightarrow^* [\![n]\!].\texttt{findFP}(([\![[n_h]]\!], \overline{[\![s]\!]}))\ \texttt{IFNULL}\ [\![n]\!].\texttt{findFP}(([\![concat]\!]([\![l_r]\!])(e_2), e_s))$$
$$\{\texttt{findFP will evaluate } [\![n_h]\!].\texttt{getName to } [\![s_h]\!], \text{ returning } [\![n_h]\!] \text{ as it matches } [\![s]\!]\}$$
$$\longrightarrow^* [\![n_h]\!]\ \texttt{IFNULL}\ [\![n]\!].\texttt{findFP}(([\![concat]\!]([\![l_r]\!])(e_2), e_s))$$
$$\{\text{as } n_h \text{ is not } n_{null}\}$$
$$\longrightarrow^* [\![n_h]\!]$$

$$\sigma_k \vdash [\![n]\!].\texttt{findFP}(([\![n_h :: l_r]\!], \underline{e_s}))\ \texttt{IFNULL}\ [\![n]\!].\texttt{findFP}((e_2, e_s))$$
$$\longrightarrow^* [\![n]\!].\texttt{findFP}(([\![n_h :: l_r]\!], \overline{[\![s]\!]}))\ \texttt{IFNULL}\ [\![n]\!].\texttt{findFP}((e_2, e_s))$$
$$\{\texttt{findFP will evaluate } [\![n_h]\!].\texttt{getName to } [\![s_h]\!], \text{ returning } [\![n_h]\!] \text{ as it matches } [\![s]\!]\}$$
$$\longrightarrow^* [\![n_h]\!]\ \texttt{IFNULL}\ [\![n]\!].\texttt{findFP}((e_2, e_s))$$

{as n_h is not n_{null}}

$\longrightarrow^* [\![n_h]\!]$

When s_h matches the s, both expressions evaluate to $[\![n_h]\!]$, so this case is satisfied.

Case 48.2.2 ($s_h \neq s$)**.**

$\sigma_k \vdash [\![n]\!.\texttt{findFP}((([\![[n_h]]\!], e_s))\ \texttt{IFNULL}\ [\![n]\!.\texttt{findFP}((([\![concat]\!]([\![l_r]\!])(e_2), e_s))$

$\longrightarrow^* [\![n]\!.\texttt{findFP}((([\![[n_h]]\!], [\![s]\!]))\ \texttt{IFNULL}\ [\![n]\!.\texttt{findFP}((([\![concat]\!]([\![l_r]\!])(e_2), e_s))$

{findFP will evaluate $[\![n_h]\!]$.getName to $[\![s_h]\!]$, moving on as it does not match $[\![s]\!]$}

$\longrightarrow^* [\![n_{null}]\!].\_\texttt{matchName}([\![(s,[])]\!])\ \texttt{IFNULL}\ [\![n]\!.\texttt{findFP}((([\![concat]\!]([\![l_r]\!])(e_2), e_s))$

{_matchName returns n_{null} for the empty list}

$\longrightarrow^* [\![n_{null}]\!]\ \texttt{IFNULL}\ [\![n]\!.\texttt{findFP}((([\![concat]\!]([\![l_r]\!])(e_2), e_s))$

$\longrightarrow^* [\![n]\!.\texttt{findFP}((([\![concat]\!]([\![l_r]\!])(e_2), e_s))$

{using the inductive hypothesis}

$\cong [\![n]\!.\texttt{findFP}((([\![l_r]\!], e_s))\ \texttt{IFNULL}\ [\![n]\!.\texttt{findFP}((e_2, e_s))$

$\longrightarrow^* [\![n]\!.\texttt{findFP}((([\![l_r]\!], [\![s]\!]))\ \texttt{IFNULL}\ [\![n]\!.\texttt{findFP}((e_2, e_s))$

$\longrightarrow^* [\![n_{null}]\!].\_\texttt{matchName}([\![(s,l_r)]\!])\ \texttt{IFNULL}\ [\![n]\!.\texttt{findFP}((e_2, e_s))$

$\sigma_k \vdash [\![n]\!.\texttt{findFP}((([\![n_h :: l_r]\!], e_s))\ \texttt{IFNULL}\ [\![n]\!.\texttt{findFP}((e_2, e_s))$

$\longrightarrow^* [\![n]\!.\texttt{findFP}((([\![n_h :: l_r]\!], [\![s]\!]))\ \texttt{IFNULL}\ [\![n]\!.\texttt{findFP}((e_2, e_s))$

{findFP will evaluate $[\![n_h]\!]$.getName to $[\![s_h]\!]$, moving on as it does not match $[\![s]\!]$}

$\longrightarrow^* [\![n_{null}]\!].\_\texttt{matchName}([\![(s,l_r)]\!])\ \texttt{IFNULL}\ [\![n]\!.\texttt{findFP}((e_2, e_s))$

At this point both evaluations have reached an identical expression. As the same context function is being used in both cases, the result is now given from multistep determinism.

As all cases have been satisfied, Lemma 48 is proven. □

Definition 7.4.1 (The "superclass chain" and "superclass induction")**.** For a number of theorems in this chapter we use the concept of a superclass chain to structure our proofs. In Featherweight Java, every class must specify a superclass, even if that superclass is the Object class. If class A has superclass B, and class B has superclass C, we call the sequence $[A, B, C, ...]$ the "superclass chain", sometimes abbreviated as "super chain". The final element in any valid superclass chain must be the Object class, as this is the only class that does not have a superclass.

Performing induction on this finite list of classes, a process we refer to as "superclass induction" or "induction on the superclass chain", can be helpful for a process that recurses to its superclass. If we want to prove some property P of a class C, it suffices to show the following two subgoals.

- P holds for the Object class.
- We can derive that P holds for C, given that P holds for its superclass.

Lemma 49. For any nodes n_x and n and any string s that adhere to the following restrictions, and assuming that under σ_k, e_s evaluates to $[\![s]\!]$, we have the following.

$$\begin{aligned}
&\forall(n, n_x \in \boldsymbol{N}), (e_s \in \boldsymbol{E}), (s \in \mathit{string}), (t \in \boldsymbol{T}), (v \in t),\\
&\quad \sigma_k \vdash [\![n_x]\!].\mathsf{findFP}((\![n]\!].\mathsf{fields}, e_s)) \longrightarrow^* _ \vdash [\![v]\!]\\
\iff\ &\sigma_j \vdash [\![n]\!].\mathsf{remoteLookup}([\![s]\!]) \longrightarrow^* _ \vdash [\![v]\!]
\end{aligned}$$

The restrictions are:

- n is of type *ClassDecl*
- $[\![n]\!].\mathsf{name}$ is not "Object"
- n is part of a valid super chain (to be explained). This is true of any *ClassDecl* node in any valid Featherweight Java tree.

Further, we assert that v is either n_{null} or some node of type FPDecl.

Proof. We proceed by induction on n's super chain. This means we need to solve the case where n is n_{null}, the case where n is the "Object" class, and where n is neither, but given that the proposition is true for its superclass.

Case 49.1 ($n = n_{null}$)**.**

$$\begin{aligned}
&\sigma_k \vdash [\![n_x]\!].\mathsf{findFP}((\underline{[\![n_{null}]\!].\mathsf{fields}}, [\![s]\!]))\\
&\longrightarrow^* [\![n_x]\!].\mathsf{findFP}((\overline{.[\![]\!]}, \underline{e_s}))\\
&\longrightarrow^* [\![n_x]\!].\mathsf{findFP}((\overline{.[\![]\!]}, \overline{[\![s]\!]}))\\
&\{\text{as } \mathsf{findFP} \text{ returns } [\![n_{null}]\!] \text{ for the empty list}\}\\
&\longrightarrow^* [\![n_{null}]\!]
\end{aligned}$$

$$\begin{aligned}
&\sigma_j \vdash [\![n_{null}]\!].\mathsf{remoteLookup}([\![s]\!])\\
&\quad \{\text{as } [\![n_{null}]\!].\mathsf{nodeType} \text{ is not } \mathit{ClassDecl}\}\\
&\longrightarrow^* [\![n_{null}]\!]
\end{aligned}$$

For the n_{null} case, n_{null} is reached by both expressions, so this case is satisfied.

Case 49.2 (n is the Object Class)**.**

$$\begin{aligned}
&\sigma_k \vdash [\![n_x]\!].\mathsf{findFP}((\underline{[\![n]\!].\mathsf{fields}}, e_s))\\
&\qquad\qquad \{\text{as } [\![n]\!].\mathsf{name} \text{ is "Object"}\}\\
&\longrightarrow^* [\![n_x]\!].\mathsf{findFP}((\overline{.[\![]\!]}, \underline{e_s}))\\
&\longrightarrow^* [\![n_x]\!].\mathsf{findFP}((\overline{.[\![]\!]}, \overline{[\![s]\!]}))
\end{aligned}$$

$$
\begin{array}{l}
\qquad\qquad \{\text{as findFP returns } [\![n_{null}]\!] \text{ for the empty list}\} \\
\longrightarrow^* [\![n_{null}]\!]
\end{array}
$$

$$
\begin{array}{l}
\sigma_j \vdash [\![n]\!].\mathsf{remoteLookup}([\![s]\!]) \\
\qquad\qquad \{\text{as } n \text{ is a } \mathit{ClassDecl} \text{ node}\} \\
\longrightarrow^* [\![n_{null}]\!].\_\mathsf{matchName}((.[\![s]\!], [\![n]\!].\mathsf{args})) \\
\qquad \mathtt{IFNULL}\ [\![n]\!].\mathsf{superClass}.\mathsf{remoteLookup}([\![s]\!]) \\
\qquad\qquad \{\text{the object class's args is the empty list}\} \\
\longrightarrow^* [\![n_{null}]\!].\_\mathsf{matchName}((.[\![s]\!], [\![[]]\!])) \\
\qquad \mathtt{IFNULL}\ [\![n]\!].\mathsf{superClass}.\mathsf{remoteLookup}([\![s]\!]) \\
\longrightarrow^* [\![n_{null}]\!]\ \mathtt{IFNULL}\ [\![n]\!].\mathsf{superClass}.\mathsf{remoteLookup}([\![s]\!]) \\
\longrightarrow^* [\![n]\!].\mathsf{superClass}.\mathsf{remoteLookup}([\![s]\!]) \\
\qquad\qquad \{\text{as } n \neq n_{null}\} \\
\longrightarrow (\mathtt{IF}\ [\![\mathit{strEq}(\text{“Object”})]\!]([\![n]\!].\mathsf{name})\ \mathtt{THEN}\ [\![n_{null}]\!] \\
\qquad \mathtt{ELSE}\ [\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{extUse}.\mathsf{name})).\mathsf{remoteLookup}([\![s]\!]) \\
\longrightarrow^* (\mathtt{IF}\ [\![\mathit{strEq}(\text{“Object”})]\!]([\![\text{“Object”}]\!])\ \mathtt{THEN}\ [\![n_{null}]\!] \\
\qquad \mathtt{ELSE}\ [\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{extUse}.\mathsf{name})).\mathsf{remoteLookup}([\![s]\!]) \\
\longrightarrow^* [\![n_{null}]\!].\mathsf{remoteLookup}([\![s]\!]) \\
\longrightarrow [\![n_{null}]\!]
\end{array}
$$

For the Object case, n_{null} is reached by both expressions, so this case is satisfied.

Case 49.3 (The inductive case)**.** By induction we are given the following.

$$
\begin{array}{ll}
& \sigma_k \vdash [\![n_x]\!].\mathsf{findFP}((\![n]\!].\mathsf{superClass}.\mathsf{fields}, e_s)) \longrightarrow^* \_ \vdash [\![v]\!] \\
\iff & \sigma_j \vdash [\![n]\!].\mathsf{superClass}.\mathsf{remoteLookup}([\![s]\!]) \longrightarrow^* \_ \vdash [\![v]\!]
\end{array}
$$

We proceed by evaluating both implementations in parallel. We say that $[\![n]\!].\mathsf{args}$ evaluates to the value $[\![l_n]\!]$. We also say that $[\![n_{null}]\!].\_\mathsf{matchName}([\![(s,l_n)]\!])$ evaluates to the value $[\![v_m]\!]$.

$$
\begin{array}{l}
\sigma_k \vdash [\![n_x]\!].\mathsf{findFP}((\![n]\!].\mathsf{fields}, e_s)) \\
\longrightarrow [\![n_x]\!].\mathsf{findFP}((\![concat]\!]([\![n]\!].\mathsf{args})([\![n]\!].\mathsf{superClass}.\mathsf{fields}), e_s)) \\
\longrightarrow^* [\![n_x]\!].\mathsf{findFP}((\![concat]\!]([\![l_n]\!])([\![n]\!].\mathsf{superClass}.\mathsf{fields}), e_s)) \\
\cong [\![n_x]\!].\mathsf{findFP}((\![l_n]\!], e_s)) \\
\qquad \mathtt{IFNULL}\ [\![n]\!].\mathsf{findFP}((\![n]\!].\mathsf{superClass}.\mathsf{fields}, e_s))
\end{array}
$$

{using Lemma 48}

$\longrightarrow^* [\![n_x]\!].\mathsf{findFP}((.[\![l_n]\!], [\![s]\!]))$
$\quad\quad \texttt{IFNULL}\ [\![n]\!].\mathsf{findFP}(([\![n]\!].\mathsf{superClass.fields}, e_s))$
$\longrightarrow^* [\![n_{null}]\!].\_\mathsf{matchName}([\![(s,l_n)]\!])$
$\quad\quad \texttt{IFNULL}\ [\![n]\!].\mathsf{findFP}(([\![n]\!].\mathsf{superClass.fields}, e_s))$
$\longrightarrow^* [\![v_m]\!]\ \texttt{IFNULL}\ [\![n]\!].\mathsf{findFP}(([\![n]\!].\mathsf{superClass.fields}, e_s))$

$\sigma_j \vdash [\![n]\!].\mathsf{remoteLookup}([\![s]\!])$

{as $n \neq n_{null}$}

$\longrightarrow [\![n_{null}]\!].\_\mathsf{matchName}(([\![s]\!], [\![n]\!].\mathsf{args}))$
$\quad\quad \texttt{IFNULL}\ [\![n]\!].\mathsf{superClass.remoteLookup}([\![s]\!])$
$\longrightarrow [\![n_{null}]\!].\_\mathsf{matchName}(([\![s]\!], [\![l_n]\!]))$
$\quad\quad \texttt{IFNULL}\ [\![n]\!].\mathsf{superClass.remoteLookup}([\![s]\!])$
$\longrightarrow [\![n_{null}]\!].\_\mathsf{matchName}([\![(s,l_n)]\!])\ \texttt{IFNULL}\ [\![n]\!].\mathsf{superClass.remoteLookup}([\![s]\!])$
$\longrightarrow [\![v_m]\!]\ \texttt{IFNULL}\ [\![n]\!].\mathsf{superClass.remoteLookup}([\![s]\!])$

We know that v_m is either n_{null} or a member of l_n, as per the semantics of $\_\mathsf{matchName}$. All members of l_n must be of type *FPDecl*, as l_n is retrieved from $n_4.\mathsf{args}$, according to the abstract grammar in Figure 7.3. If $v_m = n_{null}$, the theorem is proven using the inductive hypothesis. If $v_m \neq n_{null}$, evaluation of both expressions terminates on $[\![v_m]\!]$, so the theorem is satisfied, as v_m is a *FPDecl* node.

All cases have been satisfied, so Lemma 49 is proven. □

Lemma 50. For any n that is n_{null} or is of type *FPDecl*, we have the following.

$$\begin{aligned} &\forall (n \in \boldsymbol{N}), (t \in \boldsymbol{T})(v \in t), \\ &\sigma_k \vdash [\![n]\!].\mathsf{fpType} \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff\ &\sigma_j \vdash [\![n]\!].\mathsf{type} \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Proof. We separately consider the cases where n is an *FPDecl* node and where $n = n_{null}$.

Case 50.1 (n is an *FPDecl* node)**.** We say that $[\![n]\!].\mathsf{typeUse}$ evaluates to some value $[\![n_2]\!]$, and that $[\![n_2]\!].\mathsf{name}$ evaluates to some value $[\![s]\!]$.

$$\sigma_k \vdash [\![n]\!].\mathsf{fpType}$$

$$
\begin{array}{l}
\qquad\qquad \{\text{as } n \neq n_{null}\} \\
\longrightarrow [\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{typeUse}.\mathsf{name}) \\
\longrightarrow [\![n]\!].\mathsf{decl}([\![n_2]\!].\mathsf{name}) \\
\longrightarrow [\![n]\!].\mathsf{decl}([\![s_2]\!])
\end{array}
$$

$$
\begin{array}{l}
\sigma_j \vdash [\![n]\!].\mathsf{type} \\
\qquad\qquad \{\text{as } n \text{ is of type } \mathit{FPDecl}\} \\
\longrightarrow [\![n]\!].\mathsf{typeUse}.\mathsf{decl} \\
\longrightarrow [\![n_2]\!].\mathsf{decl} \\
\longrightarrow [\![n_2]\!].\mathsf{lookupType}([\![n_2]\!].\mathsf{name}) \\
\longrightarrow [\![n_2]\!].\mathsf{lookupType}([\![s_2]\!])
\end{array}
$$

At this point both implementations have stepped to expressions that allow the goal to be proved by Theorem 42, so this case is satisfied.

Case 50.2 ($n = n_{null}$)**.**

$$
\begin{array}{l}
\sigma_k \vdash [\![n_{null}]\!].\mathsf{fpType} \\
\longrightarrow [\![n_{null}]\!]
\end{array}
$$

$$
\begin{array}{l}
\sigma_j \vdash [\![n_{null}]\!].\mathsf{type} \\
\longrightarrow [\![n_{null}]\!]
\end{array}
$$

This case is shown trivially, as both attributes return immediately the null node.

Both cases are shown, so Lemma 50 is proven. □

7.4.2 The `type` Attribute

In this section we prove that `type` is in $A_{k\equiv j}$, which is our primary goal for proving that name and type analysis is equivalent between σ_k and σ_j. The `type` attribute is define differently in σ_k and σ_j, but separates its returned equations using the same seven selectors. For each of these cases, we will prove that `type` evaluates to the same value for both implementations, assuming that all subsequent uses of `type` evaluate identically.

The EFld Case

Theorem 51. Assuming that all subsequent calls to `type` evaluate identically in both contexts, we assert that any n of type *EFld* will evaluate $[\![n]\!].\mathsf{type}$ to the same value in both σ_k and σ_j.

$$\begin{aligned} &\forall (n \in \boldsymbol{N}), (t \in \boldsymbol{T}), (v \in t), \\ &\quad \sigma_k \vdash [\![n]\!].\mathsf{type} \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff &\ \sigma_j \vdash [\![n]\!].\mathsf{type} \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Proof. We proceed by evaluating both expressions in parallel. We assert the following equivalent evaluations, using $A_{k \equiv j}$.

- $[\![n]\!].\mathsf{expr}$ yields $[\![n_3]\!]$
- $[\![n]\!].\mathsf{nameUse}$ yields $[\![n_2]\!]$
- $[\![n_3]\!].\mathsf{type}$ yields $[\![n_4]\!]$ (the inductive case)

$\sigma_k \vdash [\![n]\!].\mathsf{type}$

{as n is an *EFld* node}

$\longrightarrow^* [\![n]\!].\mathsf{findFP}((\underline{[\![n]\!].\mathsf{expr}}.\mathsf{type}.\mathsf{fields}, [\![n]\!].\mathsf{nameUse}.\mathsf{name})).\mathsf{fpType}$

$\longrightarrow^* [\![n]\!].\mathsf{findFP}((\underline{[\![n_3]\!].\mathsf{type}}.\mathsf{fields}, [\![n]\!].\mathsf{nameUse}.\mathsf{name})).\mathsf{fpType}$

$\longrightarrow^* [\![n]\!].\mathsf{findFP}((\underline{[\![n_4]\!]}.\mathsf{fields}, [\![n]\!].\mathsf{nameUse}.\mathsf{name})).\mathsf{fpType}$

$\sigma_j \vdash [\![n]\!].\mathsf{type}$

{as n is an *EFld* node}

$\longrightarrow^* \underline{[\![n]\!].\mathsf{nameUse}}.\mathsf{decl}.\mathsf{type}$

$\longrightarrow^* \underline{[\![n_2]\!]}.\mathsf{decl}.\mathsf{type}$

{as n_2 must be of type *VarUse*, according to the abstract grammar}

$\longrightarrow^* \underline{[\![n_2]\!].\mathsf{lookup}([\![n_2]\!].\mathsf{name})}.\mathsf{type}$

$\longrightarrow^* \underline{[\![n_2]\!].\mathsf{lookup}([\![s_2]\!])}.\mathsf{type}$

{as n_2 is the nameUse of n, an *EFld* node}

$\longrightarrow^* \underline{[\![n_2]\!].\mathsf{parent}.\mathsf{expr}}.\mathsf{type}.\mathsf{remoteLookup}([\![s_2]\!]).\mathsf{type}$

$\longrightarrow^* \underline{[\![n]\!].\mathsf{expr}}.\mathsf{type}.\mathsf{remoteLookup}([\![s_2]\!]).\mathsf{type}$

$\longrightarrow^* \underline{[\![n_3]\!].\mathsf{type}}.\mathsf{remoteLookup}([\![s_2]\!]).\mathsf{type}$

$\longrightarrow^* \underline{[\![n_4]\!]}.\mathsf{remoteLookup}([\![s_2]\!]).\mathsf{type}$

At this point, we substitute the subexpressions $[\![n]\!].\mathsf{findFP}((\underline{[\![n_4]\!]}.\mathsf{fields}, [\![n]\!].\mathsf{nameUse}.\mathsf{name}))$ and $[\![n_4]\!].\mathsf{remoteLookup}([\![s_2]\!])$ with the common value n_6, using Lemma 49. This lemma also tells us that n_6 is either n_{null} or a node of type *FPDecl*.

$$\sigma_k \vdash [\![n_6]\!].\mathsf{fpType}$$

$$\sigma_j \vdash [\![n_6]\!].\mathsf{type}$$

Without performing any evaluation, we know that these two expressions will evaluate to the same result, using Lemma 50. The use of this lemma requires the knowledge that n_6 is either n_{null} or a node of type *FPDecl*, which we know from our use of Lemma 49. □

The EIdn Case

Lemma 52.

$$\begin{aligned} &\forall (n, n_1 \in \boldsymbol{N}), (e_2, e_s \in \boldsymbol{E}), (t \in \boldsymbol{T}), (v \in t), \\ &\sigma_k \vdash [\![n]\!].\mathsf{findFP}((([\![\mathit{prepend}]\!]([\![n_1]\!]))(e_2), e_s)) \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff\ &\sigma_k \vdash [\![n]\!].\mathsf{findFP}((([\![\mathit{concat}]\!]([\![[n_1]]\!]))(e_2), e_s)) \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Proof. We proceed by evaluating both expressions in parallel.

$$\begin{aligned} &\sigma_k \vdash [\![n]\!].\mathsf{findFP}((([\![\mathit{prepend}]\!]([\![n_1]\!]))(e_2), e_s)) \longrightarrow^* \_ \vdash [\![v]\!] \\ &\longrightarrow [\![n]\!].\mathsf{findFP}(([\![\mathit{prepend}(n_1)]\!](e_2), e_s)) \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

$$\begin{aligned} &\sigma_k \vdash [\![n]\!].\mathsf{findFP}((([\![\mathit{concat}]\!]([\![[n_1]]\!]))(e_2), e_s)) \longrightarrow^* \_ \vdash [\![v]\!] \\ &\longrightarrow [\![n]\!].\mathsf{findFP}(([\![\mathit{concat}([n_1])]\!](e_2), e_s)) \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Both expressions have stepped to the same expression in only one step. From the semantics of *concat* and *prepend*, and via functional extensionality, we trivially have $\mathit{prepend}(n_1) = \mathit{concat}([n_1])$. By multistep determinism, Lemma 52 is therefore proven. □

Theorem 53. Assuming that all subsequent calls to `type` evaluate identically in both contexts, we assert that any n of type EIdn will evaluate $[\![n]\!].\mathsf{type}$ to the same value in both σ_k and σ_j.

$$\begin{aligned} &\forall (n \in \boldsymbol{N}), (t \in \boldsymbol{T}), (v \in t), \\ &\sigma_k \vdash [\![n]\!].\mathsf{type} \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff\ &\sigma_j \vdash [\![n]\!].\mathsf{type} \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Proof. We proceed by evaluating both expressions in parallel. We say that $[\![n]\!].\mathsf{varUse}$ yields $[\![n_2]\!]$ and that $[\![n_2]\!].\mathsf{name}$ yields $[\![s_2]\!]$.

$\sigma_k \vdash [\![n]\!].\mathsf{type}$

{as n is an *EIdn* node}

$\longrightarrow^* [\![n]\!].\mathsf{varUse}.\mathsf{varUseType}$

$\longrightarrow^* [\![n_2]\!].\mathsf{varUseType}$

$\longrightarrow^* [\![n_2]\!].\mathsf{findFP}(([\![n_2]\!].\mathsf{env}, [\![n_2]\!].\mathsf{name})).\mathsf{fpType}$

$\sigma_j \vdash [\![n]\!].\mathsf{type}$

{as n is an *EIdn* node}

$\longrightarrow^* [\![n]\!].\mathsf{varUse}.\mathsf{decl}.\mathsf{type}$

$\longrightarrow^* [\![n_2]\!].\mathsf{decl}.\mathsf{type}$

{as n_2 must be of type *VarUse*, according to the abstract grammar}

$\longrightarrow^* [\![n_2]\!].\mathsf{lookup}([\![n_2]\!].\mathsf{name}).\mathsf{type}$

$\longrightarrow^* [\![n_2]\!].\mathsf{lookup}([\![s_2]\!]).\mathsf{type}$

At this point, the Kiama implementation will evaluate $[\![n_2]\!].\mathsf{env}$, while the JastAdd implementation will evaluate $[\![n_2]\!].\mathsf{decl}$. Evaluation begins by considering n_2, and then n_2's parent, and so on until one of the selectors match. Each implementation has four selectors, summarised in the following table.

Kiama's env	JastAdd's lookup
(ClassDecl)	(ClassDecl).methods*
(CtorDecl)	(EFld).nameUse
(MethodDecl)	(MethodDecl).expr
(Program)	(Program).*

We know that n_2 does not match any of these nodes, as its type is *VarUse* and its parent is n, whose type is *EIdn*. Remember that the selectors on the left of the above table are matching against the node in question (in this case n_2), while the selectors on the right are matching against the parent of the node in question (in this case n).

We will show that inheritance for both implementations will traverse the tree and stop at nodes n_3 and n_4, such that n_4 is n_3's parent. It is possible that $n_3 = n$.

There is no way that an *EIdn* node can be contained in the subtree of a *VarUse* node, so the (EFld).nameUse case will never match. Similarly, there is no way that any type of expression node can be contained in a *CtorDecl* subtree, so the (CtorDecl) case will also never match.

Similarly, the only way that any kind of expression node can be contained in a *ClassDecl* subtree is if it is nested inside a *MethodDecl* subtree. If this is the case, traversal up the tree will terminate once the *MethodDecl* node is reached, and will not reach the *ClassDecl* node. This eliminates the possibility of matching Kiama's (ClassDecl) selector or JastAdd's (ClassDecl).method* selector.

This leaves only two ways that the traversal up the tree will terminate: At a *MethodDecl* node, or at the *Program* node. We consider these two cases separately.

Case 53.1 (n_4 is a Program node)**.**

$$\begin{aligned}
&\sigma_k \vdash [\![n_2]\!].\mathsf{findFP}((\underline{[\![n_2]\!].\mathsf{env}}, [\![n_2]\!].\mathsf{name})).\mathsf{fpType} \\
&\qquad \{\mathsf{env}\ \text{traverses to the root } \textit{Program}\ \text{node}\} \\
&\longrightarrow^* [\![n_2]\!].\mathsf{findFP}((.\overline{[\![[]]\!]}, \underline{[\![n_2]\!].\mathsf{name}})).\mathsf{fpType} \\
&\longrightarrow^* [\![n_2]\!].\mathsf{findFP}((.[\![[]]\!], \underline{\overline{[\![s_2]\!]}})).\mathsf{fpType} \\
&\longrightarrow^* \overline{[\![n_{null}]\!]}.\mathsf{fpType} \\
&\longrightarrow [\![n_{null}]\!]
\end{aligned}$$

$$\begin{aligned}
&\sigma_j \vdash \underline{[\![n_2]\!].\mathsf{lookup}([\![s_2]\!])}.\mathsf{type} \\
&\qquad \{\mathsf{lookup}\ \text{traverses to the root } \textit{Program}\ \text{node's child}\} \\
&\longrightarrow \overline{[\![n_{null}]\!]}.\mathsf{type} \\
&\longrightarrow [\![n_{null}]\!]
\end{aligned}$$

Both implementations have evaluated to the same value, so this case is satisfied.

In terms of the FeatherWeight Java semantics, this is the case where a program's mandatory expression contains an identifier expression. This is not legal, as there are only types in scope for this expression; no variables. So it makes sense that n_{null} would be returned, as no type can be given to a semantically invalid expression.

Case 53.2 (n_4 is a MethodDecl node)**.** As a MethodDecl node can only legally exist as a child of a ClassDecl node, we will say that n_4's parent node is some ClassDecl n_5. Further, we know that a ClassDecl's parent must be the root Program node n_0. For this case, we split the case further into the cases where s_2 is and is not equal to "this".

We assert the following equivalent evaluations, based on the contents of $A_{k\equiv j}$.

- $[\![n_4]\!].\mathsf{args}$ yields $[\![l_4]\!]$
- $[\![n_5]\!].\mathsf{fields}$ yields $[\![l_{5f}]\!]$
- $[\![n_5]\!].\mathsf{thisFPDecl}$ yields $[\![n_{5t}]\!]$

Further, we know from the definition of $\mathsf{thisFPDecl}$ that n_{5t} is a *FPDecl* node, and $[\![n_{5t}]\!].\mathsf{name}$ yields $[\![\text{"this"}]\!]$.

Case 53.2.1 (s_2 = "this")**.**

$$\begin{aligned}
&\sigma_k \vdash [\![n_2]\!].\mathsf{findFP}((\underline{[\![n_2]\!].\mathsf{env}}, [\![n_2]\!].\mathsf{name})).\mathsf{fpType} \\
&\qquad \{\mathsf{env}\ \text{traverses to a } \textit{MethodDecl}\ \text{node}\} \\
&\longrightarrow \underline{[\![n_2]\!].\mathsf{findFP}(([\![concat]\!](\underline{[\![n_4]\!].\mathsf{args}})([\![n_4]\!].\mathsf{parent.env}), [\![n_2]\!].\mathsf{name})).\mathsf{fpType}} \\
&\longrightarrow \underline{[\![n_2]\!].\mathsf{findFP}(([\![concat]\!](\overline{[\![l_4]\!]})([\![n_4]\!].\mathsf{parent.env}), [\![n_2]\!].\mathsf{name})).\mathsf{fpType}} \\
&\qquad \{\text{using Lemma 48}\} \\
&\cong \underline{([\![n_2]\!].\mathsf{findFP}(([\![l_4]\!], [\![n_2]\!].\mathsf{name}))} \\
&\qquad \underline{\texttt{IFNULL}\ [\![n_2]\!].\mathsf{findFP}(([\![n_4]\!].\mathsf{parent.env}, [\![n_2]\!].\mathsf{name})))}.\mathsf{fpType}
\end{aligned}$$

$\longrightarrow$ ($[\![n_2]\!]$.findFP(($[\![l_4]\!]$, $[\![$"this"$]\!]$))
IFNULL $[\![n_2]\!]$.findFP(($[\![n_4]\!]$.parent.env, $[\![n_2]\!]$.name))).fpType
{as l_4 cannot contain any nodes whose name is "this"}
$\longrightarrow$ ($[\![n_{null}]\!]$ IFNULL $[\![n_2]\!]$.findFP(($[\![n_4]\!]$.parent.env, $[\![n_2]\!]$.name))).fpType
$\longrightarrow$ $[\![n_2]\!]$.findFP(($[\![n_4]\!]$.parent.env, $[\![n_2]\!]$.name)).fpType
$\longrightarrow$ $[\![n_2]\!]$.findFP(($[\![n_5]\!]$.env, $[\![n_2]\!]$.name)).fpType
{as n_5 is a *ClassDecl* node}
$\longrightarrow$ $[\![n_2]\!]$.findFP(($[\![prepend]\!]$($[\![n_5]\!]$.thisFPDecl)($[\![n_5]\!]$.fields), $[\![n_2]\!]$.name)).fpType
$\longrightarrow^*$ $[\![n_2]\!]$.findFP(($[\![prepend]\!]$($[\![n_{5t}]\!]$)($[\![n_5]\!]$.fields), $[\![n_2]\!]$.name)).fpType
$\longrightarrow^*$ $[\![n_2]\!]$.findFP(($[\![prepend]\!]$($[\![n_{5t}]\!]$)($[\![l_{5f}]\!]$), $[\![n_2]\!]$.name)).fpType
$\longrightarrow^*$ $[\![n_2]\!]$.findFP($[\![(n_{5t} :: l_{5f}$,"this")$]\!]$).fpType
{as n_{5t}.name yields $[\![$"this"$]\!]$, so the first element will match}
$\longrightarrow^*$ $[\![n_{5t}]\!]$.fpType

$\sigma_j \vdash$ $[\![n_2]\!]$.lookup($[\![$"this"$]\!]$).type
{lookup traverses to a *MethodDecl* node's child}
$\longrightarrow$ (IF $[\![strEq($"this"$)($"this"$)]\!]$ THEN $[\![n_3]\!]$.parent.enclosingClassDecl.thisDecl
ELSE ($[\![n_3]\!]$._matchName($[\![$"this"$]\!]$, $[\![n_3]\!]$.parent.args))
IFNULL $[\![n_3]\!]$.parent.lookup($[\![$"this"$]\!]$))).type
$\longrightarrow$ $[\![n_3]\!]$.parent.enclosingClassDecl.thisDecl.type
$\longrightarrow$ $[\![n_4]\!]$.enclosingClassDecl.thisDecl.type
{as n_4 is one of the *MethodDecl* nodes contained in n_5's methods attribute}
$\longrightarrow$ $[\![n_4]\!]$.parent.thisDecl.type
$\longrightarrow$ $[\![n_5]\!]$.thisFPDecl.type
$\longrightarrow$ $[\![n_{5t}]\!]$.type

At this point each implementation has evaluated to a state that can be proven equivalent using Lemma 50, given that we know that n_{5t} is a *FPDecl* node. Therefore this case is satisfied.

Case 53.2.2 ($s_2 \neq$ "this"). We proceed by evaluating each expressions in parallel. We use Lemma 41 to say that $[\![n_?]\!]$._matchName($[\![(s_2, l_4)]\!]$) yields n_m.

$\sigma_k \vdash$ $[\![n_2]\!]$.findFP(($[\![n_2]\!]$.env, $[\![n_2]\!]$.name)).fpType
$\longrightarrow$ $[\![n_2]\!]$.findFP(($[\![concat]\!]$($[\![n_4]\!]$.args)($[\![n_4]\!]$.parent.env), $[\![n_2]\!]$.name)).fpType
$\longrightarrow$ $[\![n_2]\!]$.findFP(($[\![concat]\!]$($[\![l_4]\!]$)($[\![n_4]\!]$.parent.env), $[\![n_2]\!]$.name)).fpType

$\cong$ ($[\![n_2]\!]$.findFP(($[\![l_4]\!]$, $[\![n_2]\!]$.name))
IFNULL $[\![n_2]\!]$.findFP(($[\![n_4]\!]$.parent.env, $[\![n_2]\!]$.name))).fpType
{using Lemma 48}
$\longrightarrow$ ($[\![n_2]\!]$.findFP(($[\![l_4]\!]$, $[\![s_2]\!]$))
IFNULL $[\![n_2]\!]$.findFP(($[\![n_4]\!]$.parent.env, $[\![n_2]\!]$.name))).fpType
$\longrightarrow^*$ ($[\![n_{null}]\!]$._matchName($[\![(s_2, l_4)]\!]$)
IFNULL $[\![n_2]\!]$.findFP(($[\![n_4]\!]$.parent.env, $[\![n_2]\!]$.name))).fpType
$\longrightarrow^*$ ($[\![n_m]\!]$ IFNULL $[\![n_2]\!]$.findFP(($[\![n_4]\!]$.parent.env, $[\![n_2]\!]$.name))).fpType

---

$\sigma_j \vdash [\![n_2]\!]$.lookup($[\![s_2]\!]$).type
$\longrightarrow$ (IF $[\![strEq(s_2)(\text{“this”})]\!]$ THEN $[\![n_3]\!]$.parent.enclosingClassDecl.thisDecl
ELSE ($[\![n_3]\!]$._matchName(($[\![s_2]\!]$, $[\![n_3]\!]$.parent.args))
IFNULL $[\![n_3]\!]$.parent.lookup($[\![s_2]\!]$))).type
{as $s_2 \neq$ “this”}
$\longrightarrow$ ($[\![n_3]\!]$._matchName(($[\![s_2]\!]$, $[\![n_3]\!]$.parent.args))
IFNULL $[\![n_3]\!]$.parent.lookup($[\![s_2]\!]$)).type
$\longrightarrow$ ($[\![n_3]\!]$._matchName(($[\![s_2]\!]$, $[\![n_4]\!]$.args))
IFNULL $[\![n_3]\!]$.parent.lookup($[\![s_2]\!]$)).type
$\longrightarrow$ ($[\![n_3]\!]$._matchName(($[\![s_2]\!]$, $[\![l_4]\!]$))
IFNULL $[\![n_3]\!]$.parent.lookup($[\![s_2]\!]$)).type
$\longrightarrow^*$ ($[\![n_3]\!]$._matchName($[\![(s_2, l_4)]\!]$)
IFNULL $[\![n_3]\!]$.parent.lookup($[\![s_2]\!]$)).type
$\longrightarrow^*$ $[\![n_m]\!]$ IFNULL $[\![n_3]\!]$.parent.lookup($[\![s_2]\!]$).type

We know that n_m is either n_{null} or a member of l_4, and all members of l_4 must be of type FPDecl, as l_4 comes from n_4.args. We now consider the cases where $n_m = n_{null}$ and where $n_m \neq n_{null}$.

Case 53.2.2.1 ($n_m = n_{null}$)**.**

$\sigma_k \vdash$ ($[\![n_{null}]\!]$ IFNULL $[\![n_2]\!]$.findFP(($[\![n_4]\!]$.parent.env, $[\![n_2]\!]$.name))).fpType
$\longrightarrow^*$ ($[\![n_2]\!]$.findFP(($[\![n_4]\!]$.parent.env, $[\![n_2]\!]$.name))).fpType
$\longrightarrow^*$ ($[\![n_2]\!]$.findFP(($[\![n_5]\!]$.env, $[\![n_2]\!]$.name))).fpType
$\longrightarrow^*$ ($[\![n_2]\!]$.findFP(($[\![prepend]\!]$($[\![n_5]\!]$.thisFPDecl)($[\![n_5]\!]$.fields), $[\![n_2]\!]$.name))).fpType
$\longrightarrow^*$ ($[\![n_2]\!]$.findFP(($[\![prepend]\!]$($[\![n_{5t}]\!]$)($[\![n_5]\!]$.fields), $[\![n_2]\!]$.name))).fpType
{using Lemma 52}
$\cong$ ($[\![n_2]\!]$.findFP(($[\![concat]\!]$($[\![[n_{5t}]]\!]$)($[\![n_5]\!]$.fields), $[\![n_2]\!]$.name))).fpType

{using Lemma 48}

$\cong$ ($[\![n_2]\!]$.findFP(($[\![[n_{5t}]]\!]$, $[\![n_2]\!]$.name))
IFNULL $[\![n_2]\!]$.findFP(($[\![n_5]\!]$.fields, $[\![n_2]\!]$.name))).fpType

$\longrightarrow^*$ ($[\![n_2]\!]$.findFP((.$[\![[n_{5t}]]\!]$, $[\![s_2]\!]$))
IFNULL $[\![n_2]\!]$.findFP(($[\![n_5]\!]$.fields, $[\![n_2]\!]$.name))).fpType

{as $s_2 \neq$ "this", and $[\![n_{5t}]\!]$.name yields $[\![$"this"$]\!]$}

$\longrightarrow^*$ ($[\![n_{null}]\!]$ IFNULL $[\![n_2]\!]$.findFP(($[\![n_5]\!]$.fields, $[\![n_2]\!]$.name))).fpType

$\longrightarrow^*$ $[\![n_2]\!]$.findFP(($[\![n_5]\!]$.fields, $[\![n_2]\!]$.name)).fpType

---

$\sigma_j \vdash [\![n_{null}]\!]$ IFNULL $[\![n_3]\!]$.parent.lookup($[\![s_2]\!]$).type

$\longrightarrow^*$ $[\![n_3]\!]$.parent.lookup($[\![s_2]\!]$).type

$\longrightarrow^*$ $[\![n_4]\!]$.lookup($[\![s_2]\!]$).type

{as n_4 is one of the *MethodDecl* nodes contained in n_5's methods attribute}

$\longrightarrow^*$ $[\![n_4]\!]$.parent.remoteLookup($[\![s_2]\!]$).type

$\longrightarrow^*$ $[\![n_5]\!]$.remoteLookup($[\![s_2]\!]$).type

Both implementations have evaluated to two expressions which can be proved equivalent using first Lemma 49 and then Lemma 50. Therefore this case is satisfied.

Case 53.2.2.2 ($n_m \neq n_{null}$)**.**

$\sigma_k \vdash$ ($[\![n_m]\!]$ IFNULL $[\![n_2]\!]$.findFP(($[\![n_4]\!]$.parent.env, $[\![n_2]\!]$.name))).fpType

$\longrightarrow^*$ $[\![n_m]\!]$.fpType

$\sigma_j \vdash [\![n_m]\!]$ IFNULL $[\![n_3]\!]$.parent.lookup($[\![s_2]\!]$).type

$\longrightarrow^*$ $[\![n_m]\!]$.type

Both implementations have stepped to expressions that are proven equivalent using Lemma 50, with the knowledge that n_m is a node of type *FPDecl*.

All cases have been satisfied, so Theorem 53 is proven. □

The ECall Case

Lemma 54. For any node n and string s, the method attribute in the Kiama implementation and the lookupMethod attribute in the JastAdd implementation will evaluate to the same value.

$$\begin{aligned} &\forall (n \in \boldsymbol{N}), (s \in \mathit{string}), (t \in \boldsymbol{T}), (v \in t), \\ &\sigma_k \vdash [\![n]\!].\mathsf{method}([\![s]\!]) \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff &\sigma_j \vdash [\![n]\!].\mathsf{lookupMethod}([\![s]\!]) \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Proof. We will proceed using superclass chain induction. If s = "Object", then both attributes will return n_{null}. If n is not a ClassDecl, then both attributes will return n_{null}. Therefore we need only prove the case where n is a *ClassDecl* node, and $s \neq$ "Object". As we are performing induction on the superclass chain, our inductive hypothesis is that Lemma 54 holds for $[\![n]\!].\mathsf{superClass}$.

Case 54.1 (The inductive case)**.** We are given the following inductive hypothesis.

$$\begin{aligned} &\sigma_k \vdash [\![n]\!].\mathsf{superClass}.\mathsf{method}([\![s]\!]) \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff &\sigma_j \vdash [\![n]\!].\mathsf{superClass}.\mathsf{lookupMethod}([\![s]\!]) \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

We proceed by evaluating the two expressions in parallel. We say that $[\![n]\!].\mathsf{methods}$ yields $[\![l_m]\!]$. As $\_\mathsf{matchName}$ is in $A_{k \equiv j}$, we say that $[\![n]\!].\_\mathsf{matchName}([\![s]\!], [\![l_m]\!])$ yields n_m.

$$\begin{aligned} &\sigma_k \vdash [\![n]\!].\mathsf{method}([\![s]\!]) \\ &\longrightarrow [\![n]\!].\_\mathsf{matchName}([\![s]\!], [\![n]\!].\mathsf{methods})\ \mathtt{IFNULL}\ [\![n]\!].\mathsf{superClass}.\mathsf{method}([\![s]\!]) \\ &\longrightarrow [\![n]\!].\_\mathsf{matchName}([\![s]\!], \overline{[\![l_m]\!]})\ \mathtt{IFNULL}\ [\![n]\!].\mathsf{superClass}.\mathsf{method}([\![s]\!]) \\ &\longrightarrow^* \overline{[\![n_m]\!]}\ \mathtt{IFNULL}\ [\![n]\!].\mathsf{superClass}.\mathsf{method}([\![s]\!]) \end{aligned}$$

$$\begin{aligned} &\sigma_j \vdash [\![n]\!].\mathsf{lookupMethod}([\![s]\!]) \\ &\longrightarrow [\![n]\!].\_\mathsf{matchName}([\![s]\!], [\![n]\!].\mathsf{methods}) \\ &\qquad \mathtt{IFNULL}\ [\![n]\!].\mathsf{superClass}.\mathsf{lookupMethod}([\![s]\!]) \\ &\longrightarrow [\![n]\!].\_\mathsf{matchName}([\![s]\!], \overline{[\![l_m]\!]})\ \mathtt{IFNULL}\ [\![n]\!].\mathsf{superClass}.\mathsf{lookupMethod}([\![s]\!]) \\ &\longrightarrow^* \overline{[\![n_m]\!]}\ \mathtt{IFNULL}\ [\![n]\!].\mathsf{superClass}.\mathsf{lookupMethod}([\![s]\!]) \end{aligned}$$

From this point, proof is trivial. If $n_m = n_{null}$, then the inductive hypothesis will solve the case. If not, then both expressions will evaluate to $[\![n_m]\!]$, and the case is also solved. Therefore Lemma 54 is proven.

□

Lemma 55. Assuming all subsequent calls to type will evaluate to the same value for both context functions, and assuming n is an *ECall* node, we have the following.

$$\begin{aligned} &\forall(n \in \boldsymbol{N}), (t \in \boldsymbol{T}), (v \in t), \\ &\sigma_k \vdash [\![n]\!].\mathsf{methodDecl} \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff\ &\sigma_j \vdash [\![n]\!].\mathsf{decl} \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Proof. We proceed by evaluating both expressions. We use the following assertions, derived from the contents of $A_{k \equiv j}$.

- $[\![n]\!].\mathsf{expr}$ yields $[\![n_2]\!]$
- $[\![n_2]\!].\mathsf{type}$ yields $[\![n_t]\!]$ (given by the first assumption for this theorem)
- $[\![n]\!].\mathsf{nameUse}$ yields $[\![n_3]\!]$
- $[\![n_3]\!].\mathsf{name}$ yields $[\![s_3]\!]$

$$\begin{aligned} &\sigma_k \vdash [\![n]\!].\mathsf{methodDecl} \\ &\longrightarrow [\![n]\!].\mathsf{expr}.\mathsf{type}.\mathsf{method}([\![n]\!].\mathsf{nameUse}.\mathsf{name}) \\ &\longrightarrow \overline{[\![n_2]\!]}.\mathsf{type}.\mathsf{method}([\![n]\!].\mathsf{nameUse}.\mathsf{name}) \\ &\longrightarrow \overline{[\![n_t]\!]}.\mathsf{method}([\![n]\!].\mathsf{nameUse}.\mathsf{name}) \\ &\longrightarrow [\![n_t]\!].\mathsf{method}(\overline{[\![n_3]\!]}.\mathsf{name}) \\ &\longrightarrow [\![n_t]\!].\mathsf{method}(\overline{[\![s_3]\!]}) \end{aligned}$$

$$\begin{aligned} &\sigma_j \vdash [\![n]\!].\mathsf{decl} \\ &\longrightarrow [\![n]\!].\mathsf{expr}.\mathsf{type}.\mathsf{lookupMethod}([\![n]\!].\mathsf{nameUse}.\mathsf{name}) \\ &\longrightarrow \overline{[\![n_2]\!]}.\mathsf{type}.\mathsf{lookupMethod}([\![n]\!].\mathsf{nameUse}.\mathsf{name}) \\ &\longrightarrow \overline{[\![n_t]\!]}.\mathsf{lookupMethod}([\![n]\!].\mathsf{nameUse}.\mathsf{name}) \\ &\longrightarrow [\![n_t]\!].\mathsf{lookupMethod}(\overline{[\![n_3]\!]}.\mathsf{name}) \\ &\longrightarrow [\![n_t]\!].\mathsf{lookupMethod}(\overline{[\![s_3]\!]}) \end{aligned}$$

At this point both implementations have stepped to expressions that are proven equivalent using Lemma 54. Therefore Lemma 55 is proven. □

Lemma 56. Assuming all subsequent calls to type will evaluate to the same value for both context functions, we have the following.

$$\begin{aligned} &\forall(n \in \boldsymbol{N}), (es, fps \in \mathit{listNode}), (t \in \boldsymbol{T}), (v \in t), \\ &\sigma_k \vdash [\![n]\!].\_\mathsf{exprsMatchFPs}([\![(es, fps)]\!]) \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff\ &\sigma_j \vdash [\![n]\!].\_\mathsf{exprsMatchFPs}([\![(es, fps)]\!]) \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Further, we assert that v is a boolean value.

Proof. This theorem assumes two underlying values, each lists of nodes. The value *es* is a list of expression nodes. The value *fps* is a list of *FPDecl* nodes.

If the lists *es* and *fps* have different lengths, then _exprsMatchFPs will return the value $[\![false]\!]$. For the cases where *es* and *fps* are the same length, we will proceed by pair induction on the lists *es* and *fps*. If the lists *es* and *fps* are both the empty list, then _exprsMatchFPs will return the value $[\![true]\!]$. Therefore we need only consider the inductive case.

Case 56.1 (The inductive case)**.** We have the following by induction.

$$\begin{aligned} & \sigma_k \vdash [\![n]\!].\_exprsMatchFPs([\![(es_r, fps_r)]\!]) \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff & \sigma_j \vdash [\![n]\!].\_exprsMatchFPs([\![(es_r, fps_r)]\!]) \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

We proceed by evaluating both expressions in parallel. We use the following assertions.

- $[\![e_h]\!]$.type yields $[\![n_t]\!]$ (given by the first assumption for this theorem)
- $[\![fp_h]\!]$.typeUse.name evaluates to $[\![n_{ht}]\!]$ (as typeUse and name are in $A_{k \equiv j}$)
- $[\![n_t]\!]$.subTypeOf($[\![n_{ht}]\!]$) evaluates to $[\![b]\!]$ (using the inductive hypothesis)

$$\begin{aligned} & \sigma_k \vdash [\![n]\!].\_exprsMatchFPs([\![(e_h :: es_r, fp_r :: fps_r)]\!]) \\ & \longrightarrow \texttt{IF } [\![e_h]\!].type.subTypeOf([\![fp_h]\!].typeUse.name) \\ & \qquad \texttt{THEN } [\![n_{null}]\!].\_exprsMatchFPs([\![(es_r, fps_r)]\!]) \texttt{ ELSE } [\![false]\!] \\ & \longrightarrow^* \texttt{IF } [\![n_t]\!].subTypeOf([\![fp_h]\!].typeUse.name) \\ & \qquad \texttt{THEN } [\![n_{null}]\!].\_exprsMatchFPs([\![(es_r, fps_r)]\!]) \texttt{ ELSE } [\![false]\!] \\ & \longrightarrow^* \texttt{IF } [\![n_t]\!].subTypeOf([\![n_{ht}]\!]) \\ & \qquad \texttt{THEN } [\![n_{null}]\!].\_exprsMatchFPs([\![(es_r, fps_r)]\!]) \texttt{ ELSE } [\![false]\!] \\ & \longrightarrow^* \texttt{IF } [\![b]\!] \texttt{ THEN } [\![n_{null}]\!].\_exprsMatchFPs([\![(es_r, fps_r)]\!]) \texttt{ ELSE } [\![false]\!] \end{aligned}$$

$$\begin{aligned} & \sigma_j \vdash [\![n]\!].\_exprsMatchFPs([\![(e_h :: es_r, fp_r :: fps_r)]\!]) \\ & \longrightarrow \texttt{IF } [\![e_h]\!].type.subTypeOf([\![fp_h]\!].typeUse.name) \\ & \qquad \texttt{THEN } [\![n_{null}]\!].\_exprsMatchFPs([\![(es_r, fps_r)]\!]) \texttt{ ELSE } [\![false]\!] \\ & \longrightarrow^* \texttt{IF } [\![n_t]\!].subTypeOf([\![fp_h]\!].typeUse.name) \\ & \qquad \texttt{THEN } [\![n_{null}]\!].\_exprsMatchFPs([\![(es_r, fps_r)]\!]) \texttt{ ELSE } [\![false]\!] \\ & \longrightarrow^* \texttt{IF } [\![n_t]\!].subTypeOf([\![n_{ht}]\!]) \\ & \qquad \texttt{THEN } [\![n_{null}]\!].\_exprsMatchFPs([\![(es_r, fps_r)]\!]) \texttt{ ELSE } [\![false]\!] \\ & \longrightarrow^* \texttt{IF } [\![b]\!] \texttt{ THEN } [\![n_{null}]\!].\_exprsMatchFPs([\![(es_r, fps_r)]\!]) \texttt{ ELSE } [\![false]\!] \end{aligned}$$

Continuing evaluation from this point hinges on the value of the boolean value *b*. If *b* is *false*, then both expressions step immediately to the value *false*. If *b* is true, then we step to a state that is solved immediately using the inductive hypothesis. We can use the inductive hypothesis in this case, as _exprsMatchFPs is an attribute that ignores its input node.

□

Theorem 57. Assuming that all subsequent calls to type evaluate identically in both contexts, we assert that any n of type *ECall* will evaluate $[\![n]\!].\mathsf{type}$ to the same value in both σ_k and σ_j.

$$\begin{aligned} &\forall (n \in \boldsymbol{N}), (t \in \boldsymbol{T}), (v \in t), \\ &\quad \sigma_k \vdash [\![n]\!].\mathsf{type} \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff &\quad \sigma_j \vdash [\![n]\!].\mathsf{type} \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Proof. We proceed by evaluating both expressions in parallel. We use the following assertions.

- $[\![n]\!].\mathsf{methodDecl}$ in σ_k and $[\![n]\!].\mathsf{decl}$ in σ_j both evaluate to $[\![n_d]\!]$ (using Lemma 55)
- $[\![n]\!].\mathsf{exprs}$ yields $[\![l_e]\!]$
- $[\![n_d]\!].\mathsf{args}$ yields $[\![l_d]\!]$
- $[\![n]\!].\_\mathsf{exprsMatchFPs}([\![l_e]\!], [\![l_d]\!])$ evaluates to $[\![b]\!]$ (using Lemma 56)
- $[\![n_d]\!].\mathsf{rtnUse}$ yields $[\![n_{du}]\!]$
- $[\![n_{du}]\!].\mathsf{name}$ yields $[\![s_{du}]\!]$

$\sigma_k \vdash [\![n]\!].\mathsf{type}$

{as n is an *ECall* node}

$\longrightarrow$ IF $[\![isNull]\!]([\![n]\!].\mathsf{methodDecl})$ THEN $[\![n_{null}]\!]$
ELSE IF $[\![n]\!].\_\mathsf{exprsMatchFPs}([\![n]\!].\mathsf{exprs}, [\![n]\!].\mathsf{methodDecl.args}))$
THEN $[\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{methodDecl.rtnUse.name})$ ELSE $[\![n_{null}]\!]$

$\longrightarrow$ IF $[\![isNull]\!]([\![n_d]\!])$ THEN $[\![n_{null}]\!]$
ELSE IF $[\![n]\!].\_\mathsf{exprsMatchFPs}([\![n]\!].\mathsf{exprs}, [\![n]\!].\mathsf{methodDecl.args}))$
THEN $[\![n]\!].\mathsf{decl}([\![n]\!].\mathsf{methodDecl.rtnUse.name})$ ELSE $[\![n_{null}]\!]$

$\sigma_j \vdash [\![n]\!].\mathsf{type}$

{as n is an *ECall* node}

$\longrightarrow$ IF $[\![isNull]\!]([\![n]\!].\mathsf{decl})$ THEN $[\![n_{null}]\!]$
ELSE IF $[\![n]\!].\_\mathsf{exprsMatchFPs}([\![n]\!].\mathsf{exprs}, [\![n]\!].\mathsf{decl.args}))$
THEN $[\![n]\!].\mathsf{decl.rtnUse}$ ELSE $[\![n_{null}]\!]$

$\longrightarrow^*$ IF $[\![isNull]\!]([\![n_d]\!])$ THEN $[\![n_{null}]\!]$
ELSE IF $[\![n]\!].\_\mathsf{exprsMatchFPs}([\![n]\!].\mathsf{exprs}, [\![n]\!].\mathsf{decl.args}))$
THEN $[\![n]\!].\mathsf{decl.rtnUse.decl}$ ELSE $[\![n_{null}]\!]$

We now separately consider the cases where n_d is and is not equal to n_{null}.

Case 57.1 ($n_d = n_{null}$)**.**

$\sigma_k \vdash$ IF $[\![isNull]\!]([\![n_{null}]\!])$ THEN $[\![n_{null}]\!]$
ELSE IF $[\![n]\!]$._exprsMatchFPs(($[\![n]\!]$.exprs, $[\![n]\!]$.methodDecl.args))
THEN $[\![n]\!]$.decl($[\![n]\!]$.methodDecl.rtnUse.name) ELSE $[\![n_{null}]\!]$
$\longrightarrow [\![n_{null}]\!]$

$\sigma_j \vdash$ IF $[\![isNull]\!]([\![n_{null}]\!])$ THEN $[\![n_{null}]\!]$
ELSE IF $[\![n]\!]$._exprsMatchFPs(($[\![n]\!]$.exprs, $[\![n]\!]$.decl.args))
THEN $[\![n]\!]$.decl.rtnUse.decl ELSE $[\![n_{null}]\!]$
$\longrightarrow [\![n_{null}]\!]$

This case has been shown trivially.

Case 57.2 ($n_d \neq n_{null}$)**.**

$\sigma_k \vdash$ IF $[\![isNull]\!]([\![n_d]\!])$ THEN $[\![n_{null}]\!]$
ELSE IF $[\![n]\!]$._exprsMatchFPs(($[\![n]\!]$.exprs, $[\![n]\!]$.methodDecl.args))
THEN $[\![n]\!]$.decl($[\![n]\!]$.methodDecl.rtnUse.name) ELSE $[\![n_{null}]\!]$
$\longrightarrow^*$ IF $[\![n]\!]$._exprsMatchFPs(($[\![n]\!]$.exprs, $[\![n]\!]$.methodDecl.args))
THEN $[\![n]\!]$.decl($[\![n]\!]$.methodDecl.rtnUse.name) ELSE $[\![n_{null}]\!]$
$\longrightarrow$ IF $[\![n]\!]$._exprsMatchFPs((.$\overline{[\![l_e]\!]}$, $[\![n]\!]$.methodDecl.args))
THEN $[\![n]\!]$.decl($[\![n]\!]$.methodDecl.rtnUse.name) ELSE $[\![n_{null}]\!]$
$\longrightarrow^*$ IF $[\![n]\!]$._exprsMatchFPs((.$\overline{[\![l_e]\!]}$, $\overline{[\![n_d]\!]}$.args))
THEN $[\![n]\!]$.decl($[\![n]\!]$.methodDecl.rtnUse.name) ELSE $[\![n_{null}]\!]$
$\longrightarrow$ IF $[\![n]\!]$._exprsMatchFPs((.$\overline{[\![l_e]\!]}$, $\overline{[\![l_d]\!]}$))
THEN $[\![n]\!]$.decl($[\![n]\!]$.methodDecl.rtnUse.name) ELSE $[\![n_{null}]\!]$

{using Lemma 56}

$\longrightarrow^*$ IF $\overline{[\![b]\!]}$ THEN $[\![n]\!]$.decl($[\![n]\!]$.methodDecl.rtnUse.name) ELSE $[\![n_{null}]\!]$

$\sigma_j \vdash$ IF $[\![isNull]\!]([\![n_d]\!])$ THEN $[\![n_{null}]\!]$
ELSE IF $[\![n]\!]$._exprsMatchFPs(($[\![n]\!]$.exprs, $[\![n]\!]$.decl.args))
THEN $[\![n]\!]$.decl.rtnUse.decl ELSE $[\![n_{null}]\!]$
$\longrightarrow^*$ IF $[\![n]\!]$._exprsMatchFPs(($[\![n]\!]$.exprs, $[\![n]\!]$.decl.args))
THEN $[\![n]\!]$.decl.rtnUse.decl ELSE $[\![n_{null}]\!]$
$\longrightarrow$ IF $[\![n]\!]$._exprsMatchFPs(($\overline{[\![l_e]\!]}$, $[\![n]\!]$.decl.args))
THEN $[\![n]\!]$.decl.rtnUse.decl ELSE $[\![n_{null}]\!]$
$\longrightarrow^*$ IF $[\![n]\!]$._exprsMatchFPs(($[\![l_e]\!]$, $\overline{[\![n_d]\!]}$.args))

$$
\begin{aligned}
&\texttt{THEN } [\![n]\!]\texttt{.decl.rtnUse.decl ELSE } [\![n_{null}]\!] \\
\longrightarrow^* &\texttt{IF } [\![n]\!]\texttt{._exprsMatchFPs(([}\![l_e]\!]\texttt{, } \overline{[\![l_d]\!]}\texttt{))} \\
&\texttt{THEN } [\![n]\!]\texttt{.decl.rtnUse.decl ELSE } [\![n_{null}]\!] \\
&\qquad\qquad \{\text{using Lemma 56}\} \\
\longrightarrow^* &\texttt{IF } \overline{[\![b]\!]} \texttt{ THEN } [\![n]\!]\texttt{.decl.rtnUse.decl ELSE } [\![n_{null}]\!]
\end{aligned}
$$

If b is *false*, then both expressions step immediately to the value n_{null}, and this case is satisfied. We will now continue evaluation assuming b is *true*.

$$
\begin{aligned}
\sigma_k \vdash &\texttt{IF } [\![\mathit{true}]\!] \texttt{ THEN } [\![n]\!]\texttt{.decl(}[\![n]\!]\texttt{.methodDecl.rtnUse.name) ELSE } [\![n_{null}]\!] \\
\longrightarrow\ &[\![n]\!]\texttt{.decl(}[\![n]\!]\texttt{.methodDecl.rtnUse.name)} \\
\longrightarrow^*\ &[\![n]\!]\texttt{.decl(}\overline{[\![n_d]\!]}\texttt{.rtnUse.name)} \\
\longrightarrow\ &[\![n]\!]\texttt{.decl(}\overline{[\![n_{du}]\!]}\texttt{.name)} \\
\longrightarrow\ &[\![n]\!]\texttt{.decl(}\overline{[\![s_{du}]\!]}\texttt{)}
\end{aligned}
$$

$$
\begin{aligned}
\sigma_j \vdash &\texttt{IF } [\![\mathit{true}]\!] \texttt{ THEN } [\![n]\!]\texttt{.decl.rtnUse.decl ELSE } [\![n_{null}]\!] \\
\longrightarrow\ &[\![n]\!]\texttt{.decl.rtnUse.decl} \\
\longrightarrow^*\ &\overline{[\![n_d]\!]}\texttt{.rtnUse.decl} \\
\longrightarrow\ &\overline{[\![n_{du}]\!]}\texttt{.decl} \\
\longrightarrow\ &[\![n_{du}]\!]\texttt{.lookupType(}[\![n_{du}]\!]\texttt{.name)} \\
\longrightarrow\ &[\![n_{du}]\!]\texttt{.lookupType(}\overline{[\![s_{du}]\!]}\texttt{)}
\end{aligned}
$$

At this point, both evaluations have reached expressions which can be proven equivalent using Theorem 42, so this case is satisfied.

All cases have been considered, so Theorem 53 is proven. □

The ENew Case

Theorem 58. Assuming that all subsequent calls to `type` evaluate identically in both contexts, we assert that any n of type *ENew* will evaluate $[\![n]\!]\texttt{.type}$ to the same value in both σ_k and σ_j.

$$
\begin{aligned}
&\forall (n \in \boldsymbol{N}), (t \in \boldsymbol{T}), (v \in t), \\
&\quad \sigma_k \vdash [\![n]\!]\texttt{.type} \longrightarrow^* \_ \vdash [\![v]\!] \\
\iff\ &\sigma_j \vdash [\![n]\!]\texttt{.type} \longrightarrow^* \_ \vdash [\![v]\!]
\end{aligned}
$$

Proof. We proceed by evaluating both expressions in parallel. We use the following assertions.

- $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name) under σ_k and $[\![n]\!]$.typeUse.decl under σ_j both evaluate to $[\![n_d]\!]$ (using Lemma 43)
- $[\![n]\!]$.exprs yields $[\![l_e]\!]$
- $[\![n_d]\!]$.fields yields $[\![l_d]\!]$
- $[\![n]\!]$._exprsMatchFPs(($[\![l_e]\!]$, $\overline{[\![l_d]\!]}$)) evaluates to $[\![b]\!]$, and b is a boolean value (from Lemma 56)

$\sigma_k \vdash [\![n]\!]$.type

{as n is a *ENew* node}

$\longrightarrow$ IF $[\![isNull]\!]$($[\![n]\!]$.decl($[\![n]\!]$.typeUse.name)) THEN $[\![n_{null}]\!]$
ELSE IF $[\![n]\!]$._exprsMatchFPs(
($[\![n]\!]$.exprs, $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name).fields))
THEN $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name) ELSE $[\![n_{null}]\!]$

$\longrightarrow^*$ IF $[\![isNull]\!]$($\overline{[\![n_d]\!]}$) THEN $[\![n_{null}]\!]$
ELSE IF $[\![n]\!]$._exprsMatchFPs(
($[\![n]\!]$.exprs, $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name).fields))
THEN $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name) ELSE $[\![n_{null}]\!]$

$\sigma_j \vdash [\![n]\!]$.type

{as n is a *ENew* node}

$\longrightarrow$ IF $[\![isNull]\!]$($[\![n]\!]$.typeUse.decl) THEN $[\![n_{null}]\!]$
ELSE IF $[\![n]\!]$._exprsMatchFPs(($[\![n]\!]$.exprs, $[\![n]\!]$.typeUse.decl.fields))
THEN $[\![n]\!]$.typeUse.decl ELSE $[\![n_{null}]\!]$

$\longrightarrow^*$ IF $[\![isNull]\!]$($\overline{[\![n_d]\!]}$) THEN $[\![n_{null}]\!]$
ELSE IF $[\![n]\!]$._exprsMatchFPs(($[\![n]\!]$.exprs, $[\![n]\!]$.typeUse.decl.fields))
THEN $[\![n]\!]$.typeUse.decl ELSE $[\![n_{null}]\!]$

If n_d is equal to n_{null}, then both implementations will step immediately to $[\![n_{null}]\!]$, and the goal is satisfied. We will continue evaluation assuming n_d is not equal to n_{null}.

$\sigma_k \vdash$ IF $[\![isNull]\!]$($[\![n_d]\!]$) THEN $[\![n_{null}]\!]$
ELSE IF $[\![n]\!]$._exprsMatchFPs(
($[\![n]\!]$.exprs, $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name).fields))
THEN $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name) ELSE $[\![n_{null}]\!]$

{as $n_d \neq n_{null}$}

$\longrightarrow^*$ IF $[\![n]\!]$._exprsMatchFPs(($[\![n]\!]$.exprs, $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name).fields))
THEN $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name) ELSE $[\![n_{null}]\!]$

$\longrightarrow$ IF $[\![n]\!]$._exprsMatchFPs(($[\![l_e]\!]$, $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name).fields))
THEN $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name) ELSE $[\![n_{null}]\!]$
$\longrightarrow^*$ IF $[\![n]\!]$._exprsMatchFPs(($[\![l_e]\!]$, $[\![n_d]\!]$.fields))
THEN $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name) ELSE $[\![n_{null}]\!]$
$\longrightarrow^*$ IF $[\![n]\!]$._exprsMatchFPs(($[\![l_e]\!]$, $[\![l_d]\!]$))
THEN $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name) ELSE $[\![n_{null}]\!]$
$\longrightarrow^*$ IF $[\![b]\!]$ THEN $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name) ELSE $[\![n_{null}]\!]$

$\sigma_j \vdash$ IF $[\![isNull]\!]$($[\![n_d]\!]$) THEN $[\![n_{null}]\!]$
ELSE IF $[\![n]\!]$._exprsMatchFPs(($[\![n]\!]$.exprs, $[\![n]\!]$.typeUse.decl.fields))
THEN $[\![n]\!]$.typeUse.decl ELSE $[\![n_{null}]\!]$
{as $n_d \neq n_{null}$}
$\longrightarrow^*$ IF $[\![n]\!]$._exprsMatchFPs(($[\![n]\!]$.exprs, $[\![n]\!]$.typeUse.decl.fields))
THEN $[\![n]\!]$.typeUse.decl ELSE $[\![n_{null}]\!]$
$\longrightarrow$ IF $[\![n]\!]$._exprsMatchFPs(($[\![l_e]\!]$, $[\![n]\!]$.typeUse.decl.fields))
THEN $[\![n]\!]$.typeUse.decl ELSE $[\![n_{null}]\!]$
$\longrightarrow^*$ IF $[\![n]\!]$._exprsMatchFPs(($[\![l_e]\!]$, $[\![n_d]\!]$.fields))
THEN $[\![n]\!]$.typeUse.decl ELSE $[\![n_{null}]\!]$
$\longrightarrow^*$ IF $[\![n]\!]$._exprsMatchFPs(($[\![l_e]\!]$, $[\![l_d]\!]$))
THEN $[\![n]\!]$.typeUse.decl ELSE $[\![n_{null}]\!]$
$\longrightarrow^*$ IF $[\![b]\!]$ THEN $[\![n]\!]$.typeUse.decl ELSE $[\![n_{null}]\!]$

If b is false, then both expressions will step immediately to $[\![n_{null}]\!]$, and the goal is satisfied. We will continue evaluation assuming b is *true*.

$\sigma_k \vdash$ IF $[\![true]\!]$ THEN $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name) ELSE $[\![n_{null}]\!]$
$\longrightarrow$ $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name)

$\sigma_j \vdash$ IF $[\![true]\!]$ THEN $[\![n]\!]$.typeUse.decl ELSE $[\![n_{null}]\!]$
$\longrightarrow$ $[\![n]\!]$.typeUse.decl

At this point both implementations have evaluated to expressions that can be proved equivalent using Lemma 43. Therefore Theorem 58 is proven. □

The ECast Case

Theorem 59. Assuming that all subsequent calls to type evaluate identically in both contexts, we assert that any n of type *ECast* will evaluate $[\![n]\!]$.type to the same value in both σ_k and σ_j.

$$\begin{aligned} &\forall (n \in \boldsymbol{N}), (t \in \boldsymbol{T}), (v \in t), \\ &\quad \sigma_k \vdash [\![n]\!].\texttt{type} \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff &\quad \sigma_j \vdash [\![n]\!].\texttt{type} \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Proof. We proceed by evaluating both expressions in parallel. We use the following assertions.

- $[\![n]\!]$.expr yields $[\![n_2]\!]$ (expr $\in A_{k \equiv j}$)
- $[\![n_2]\!]$.type evaluates to $[\![n_t]\!]$ (the inductive hypothesis)

$\sigma_k \vdash [\![n]\!]$.type

{as n is an *ECast* node}

$\longrightarrow$ IF $[\![isNull]\!]([\![n]\!]$.expr.type) THEN $[\![n_{null}]\!]$ ELSE $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name)

$\longrightarrow$ IF $[\![isNull]\!](\overline{[\![n_2]\!]}$.type) THEN $[\![n_{null}]\!]$ ELSE $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name)

$\longrightarrow^*$ IF $[\![isNull]\!](\overline{[\![n_t]\!]})$ THEN $[\![n_{null}]\!]$ ELSE $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name)

$\sigma_j \vdash [\![n]\!]$.type

{as n is an *ECast* node}

$\longrightarrow$ IF $[\![isNull]\!]([\![n]\!]$.expr.type) THEN $[\![n_{null}]\!]$ ELSE $[\![n]\!]$.typeUse.decl

$\longrightarrow$ IF $[\![isNull]\!](\overline{[\![n_2]\!]}$.type) THEN $[\![n_{null}]\!]$ ELSE $[\![n]\!]$.typeUse.decl

$\longrightarrow^*$ IF $[\![isNull]\!](\overline{[\![n_t]\!]})$ THEN $[\![n_{null}]\!]$ ELSE $[\![n]\!]$.typeUse.decl

Now we must separately consider the cases where n_t is and is not equal to n_{null}. If n_t is equal to n_{null}, then both expressions will step immediately to the value $[\![n_{null}]\!]$, and the theorem will hold. We will continue evaluation assuming n_t is not equal to n_{null}.

$\sigma_k \vdash$ IF $[\![isNull]\!]([\![n_t]\!])$ THEN $[\![n_{null}]\!]$ ELSE $[\![n]\!]$.decl($[\![n]\!]$.typeUse.name)
$\longrightarrow^* [\![n]\!]$.decl($[\![n]\!]$.typeUse.name)

$\sigma_j \vdash$ IF $[\![isNull]\!]([\![n_t]\!])$ THEN $[\![n_{null}]\!]$ ELSE $[\![n]\!]$.typeUse.decl
$\longrightarrow^* [\![n]\!]$.typeUse.decl

At this point both implementations have evaluated to expressions that can be proven equivalent using Lemma 43, so Theorem 59 is proven. □

Theorem 60. We assert that any node n of type *FPDecl* will evaluate $[\![n]\!].\texttt{type}$ to the same value in both σ_k and σ_j.

$$\begin{aligned} &\forall(n \in \boldsymbol{N}), (t \in \boldsymbol{T}), (v \in t), \\ &\quad \sigma_k \vdash [\![n]\!].\texttt{type} \longrightarrow^* \_ \vdash [\![v]\!] \\ \iff &\ \sigma_j \vdash [\![n]\!].\texttt{type} \longrightarrow^* \_ \vdash [\![v]\!] \end{aligned}$$

Proof. We proceed by evaluating both expressions in parallel.

$$\frac{\begin{aligned}&\sigma_k \vdash [\![n]\!].\texttt{type} \\ &\longrightarrow [\![n]\!].\texttt{fpType}\end{aligned}}{\sigma_j \vdash [\![n]\!].\texttt{type}}$$

Already both implementations have evaluated to expressions that can be proven equivalent using Lemma 50, so Theorem 42 is proven. □

The `type` Attribute Theorem

Theorem 61 ($\texttt{type} \in A_{k\equiv j}$)**.** We assert that $\texttt{type} \in A_{k\equiv j}$.

Proof. To prove this, we break down the possible output expressions for `type` in both σ_k and σ_j, and we perform induction on the stack of recursive calls to `type` that could occur during evaluation. We are evaluating `type` on some node n. We will consider the following values that $[\![n]\!].\texttt{nodeType}$ could hold.

- If n is an *EFld* node, we use Theorem 51.
- If n is an *EIdn* node, we use Theorem 53.
- If n is an *ECall* node, we use Theorem 57.
- If n is an *ENew* node, we use Theorem 58.
- If n is an *ECast* node, we use Theorem 59.
- If n is an *FPDecl* node, we use Theorem 60.
- If n is any other kind of node, both implementations return $[\![n_{null}]\!]$, so the theorem is proven trivially.

We have proven Theorem 61. As `type` will always evaluate to the same value in both σ_k and σ_j, and `type` is the attribute that performs name analysis and type checking in both implementations of Featherweight Java, we have proven that the Kiama and JastAdd implementations of name analysis are equivalent. □

7.5 Quantitative Analysis

Now that we have proven that type analysis in both implementations is *qualitatively* equivalent, we will perform some *quantitative* analysis, to try to determine under what conditions each implementation will evaluate faster. To do this, we will attempt to produce a formula for the number of evaluation steps needed for each implementation to evaluate a particular branch of type analysis.

Theorem 62. For any node n of type *ClassDecl*, we have the following, where Q is a constant that is independent of the program under evaluation, but specific to the Kiama specification.

$$\begin{aligned} & [\![n]\!].\mathsf{superClass} \text{ evaluates in } Q \text{ steps for any } n && (62.1)\\ & z \text{ is the length of the superclass chain} && (62.2)\\ \implies & \sigma_k \vdash [\![n]\!].\mathsf{fields} \longrightarrow^{5+(8+Q)z} \_ \vdash [\![\_]\!] && (62.3) \end{aligned}$$

Proof. We proceed by induction on the superclass chain (which is the same as induction on z, as z is the length of the superclass chain). This gives us two cases: where n is the Object class, and where n is not.

Case 62.1 (n_p is the Object class)**.** We proceed by evaluating the expression.

$$\begin{aligned} & \sigma_k \vdash [\![n]\!].\mathsf{fields}\\ & \qquad \{\text{as } n \text{ is the Object class}\}\\ \longrightarrow^4 & \texttt{IF } [\![\mathit{strEq}(\text{“Object”})(\text{“Object”})]\!]\\ & \quad \texttt{THEN } [\![[]]\!]\\ & \quad \texttt{ELSE } [\![\mathit{concat}]\!]\\ & \qquad ([\![n]\!].\mathsf{args})\\ & \qquad ([\![n]\!].\mathsf{superClass}.\mathsf{fields})\\ \longrightarrow & [\![[]]\!] \end{aligned}$$

In the case where n is the Object class, we have $z = 0$.

$$\begin{aligned} \text{steps taken} &= 5\\ &= 5 + (8 + Q)z \end{aligned}$$

Case 62.2 (n is not the Object class)**.** We have the inductive hypothesis as follows.

$$\begin{aligned} & (z-1)\text{is the length of the superclass chain of } n\text{'s } \mathsf{superClass}\ n_s && (62.4)\\ & \sigma_k \vdash [\![n_s]\!].\mathsf{fields} \longrightarrow^{5+(8+Q)(z-1)} \_ \vdash [\![\_]\!] && (62.5) \end{aligned}$$

We proceed by evaluating the expression.

$$
\begin{aligned}
&\sigma_k \vdash [\![n]\!].\mathsf{fields} \\
&\qquad \{s \text{ is the } \mathsf{name} \text{ of } n\} \\
&\longrightarrow^{4} \texttt{IF } [\![strEq(s)(\text{“Object”})]\!] \\
&\qquad\quad \texttt{THEN } [\![[]]\!] \\
&\qquad\quad \texttt{ELSE } [\![concat]\!]([\![n]\!].\mathsf{args}) \\
&\qquad\qquad ([\![n]\!].\mathsf{superClass}.\mathsf{fields}) \\
&\{s \neq \text{“Object”, as } n \text{ is not the Object class}\} \\
&\longrightarrow [\![concat]\!]([\![n]\!].\mathsf{args}) \\
&\qquad ([\![n]\!].\mathsf{superClass}.\mathsf{fields}) \\
&\qquad\quad \{l \text{ is the } \mathsf{args} \text{ of } n\} \\
&\longrightarrow^{2} [\![concat(l)]\!]([\![n]\!].\mathsf{superClass}.\mathsf{fields}) \\
&\{Q \text{ is the steps needed for } \mathsf{superClass}\} \\
&\longrightarrow^{Q} [\![concat(l)]\!]([\![n_s]\!].\mathsf{fields}) \\
&\longrightarrow^{5+(8+Q)(z-1)} [\![concat(l)]\!]([\![\_]\!]) \\
&\longrightarrow [\![\_]\!]
\end{aligned}
$$

$$
\begin{aligned}
\text{steps taken} &= 8 + Q + (5 + (8 + Q)(z - 1)) \\
&= 5 + (8 + Q)z
\end{aligned}
$$

□

Theorem 63. For any node n of type *ClassDecl*, we have the following.

$$
\begin{aligned}
&[\![n]\!].\mathsf{superClass} \text{ evaluates in } Q \text{ steps for any } n && (63.1)\\
&z \text{ is the length of the superclass chain} && (63.2)\\
\implies &\sigma_k \vdash [\![n]\!].\mathsf{env} \longrightarrow^{10+(8+Q)z} \_ \vdash [\![\_]\!] && (63.3)
\end{aligned}
$$

Proof. We proceed by evaluating the expression.

$$
\begin{aligned}
&\sigma_k \vdash [\![n]\!].\mathsf{env} \\
&\longrightarrow [\![prepend]\!]([\![n]\!].\mathsf{thisFPDecl})([\![n]\!].\mathsf{fields}) \\
&\longrightarrow^{2} [\![prepend(n_{this})]\!]([\![n]\!].\mathsf{fields}) \\
&\qquad \{\text{from Theorem 62}\} \\
&\longrightarrow^{5+(8+Q)z} [\![prepend(n_{this})]\!]([\![l]\!]) \\
&\longrightarrow [\![n_{this} :: l]\!]
\end{aligned}
$$

$$\begin{aligned}\text{steps taken} &= 5 + 5 + (8 + Q)z \\ &= 10 + (8 + Q)z\end{aligned}$$

□

Theorem 64. For any node n of type *EIdn*, whose parent is a *MethodDecl* node, we have the following.

$$[\![n]\!].\mathsf{superClass} \text{ evaluates in } Q \text{ steps for any } n \tag{64.1}$$
$$z \text{ is the length of the superclass chain} \tag{64.2}$$
$$\implies \sigma_k \vdash [\![n]\!].\mathsf{env} \longrightarrow^{16+(8+Q)z} \_ \vdash [\![l_m \mathbin{++} \_]\!] \tag{64.3}$$

Proof. We proceed by evaluating the expression.

$$\begin{aligned}
&\sigma_k \vdash [\![n]\!].\mathsf{env} \\
&\longrightarrow [\![n_m]\!].\mathsf{env} \\
&\longrightarrow [\![concat]\!]([\![n_m]\!].\mathsf{args})([\![n_m]\!].\mathsf{parent.env}) \\
&\quad \{\text{where } l_m \text{ is the list of nodes returned by } [\![n_m]\!].\mathsf{args}\} \\
&\longrightarrow^{2} [\![concat(l_m)]\!]([\![n_m]\!].\mathsf{parent.env}) \\
&\longrightarrow [\![concat(l_m)]\!]([\![n_c]\!].\mathsf{env}) \\
&\quad \{\text{using Theorem 63}\} \\
&\longrightarrow^{10+(8+Q)z} [\![concat(l_m)]\!]([\![_]\!]) \\
&\longrightarrow [\![l_m \mathbin{++} _]\!]
\end{aligned}$$

$$\begin{aligned}\text{steps taken} &= 6 + (10 + (8 + Q)z) \\ &= 16 + (8 + Q)z\end{aligned}$$

□

Lemma 65. For any list of *FPDecl* nodes l, where none of the nodes in l are n_{null} or have a $\mathsf{getName}$ value of s.

$$\text{The length of } l \text{ is } z \tag{65.1}$$
$$\implies \sigma_j \vdash [\![n_{null}]\!].\_\mathsf{matchName}([\![(s,l)]\!]) \longrightarrow^{7z+1} \_ \vdash [\![n_{null}]\!] \tag{65.2}$$

Proof. We proceed by induction on l. This gives us a case where l is the empty list, and a case where l is at least one element, and the theorem holds for the rest of the list.

Case 65.1 (l is the empty list)**.**

$$\begin{aligned}&\sigma_j \vdash [\![n_{null}]\!].\_\mathsf{matchName}([\![(s,[])]\!])\\ &\longrightarrow \_ \vdash [\![n_{null}]\!]\end{aligned}$$

$$\begin{aligned}\text{steps taken} &= 1\\ &= 7z + 1\end{aligned}$$

Case 65.2 (l is not the empty list)**.** We have the following by induction.

$$\text{The length of } l_2 \text{ is } (z-1) \tag{65.3}$$

$$\sigma_j \vdash [\![n_{null}]\!].\_\mathsf{matchName}([\![(s,l_2)]\!]) \longrightarrow^{7(z-1)+1} \_ \vdash [\![n_{null}]\!] \tag{65.4}$$

We proceed by evaluating the expression.

$$\begin{aligned}&\sigma_j \vdash [\![n_{null}]\!].\_\mathsf{matchName}([\![(s,n_1 :: l_2)]\!])\\ &\longrightarrow \mathtt{IF}\ [\![strEq]\!]([\![n_1]\!].\mathsf{getName})([\![s]\!])\\ &\qquad \mathtt{THEN}\ [\![n_1]\!]\\ &\qquad \mathtt{ELSE}\ [\![n_{null}]\!].\_\mathsf{matchName}([\![(s,l_2)]\!])\\ &\longrightarrow^{5} \mathtt{IF}\ [\![strEq(s_2)(s)]\!]\\ &\qquad \mathtt{THEN}\ [\![n_1]\!]\\ &\qquad \mathtt{ELSE}\ [\![n_{null}]\!].\_\mathsf{matchName}([\![(s,l_2)]\!])\\ &\qquad\quad \{\text{as } s_2 \text{ is not the same value as } s\}\\ &\longrightarrow [\![n_{null}]\!].\_\mathsf{matchName}([\![(s,l_2)]\!])\\ &\qquad\quad \{\text{the inductive hypothesis}\}\\ &\longrightarrow^{7(z-1)+1} [\![n_{null}]\!]\end{aligned}$$

$$\begin{aligned}\text{steps taken} &= 7 + 7(z-1) + 1\\ &= 7z + 1\end{aligned}$$

□

Lemma 66. For any list of *FPDecl* nodes l, where none of the nodes in l are n_{null} or have a $\mathsf{getName}$ value of s.

$$\text{The length of } l \text{ is } z \tag{66.1}$$

$$\implies \sigma_k \vdash [\![n_{null}]\!].\_\mathsf{matchName}([\![(s,l)]\!]) \longrightarrow^{7z+1} \_ \vdash [\![n_{null}]\!] \tag{66.2}$$

Proof. This lemma is the same as Lemma 65, except that we are now evaluating using the Kiama specification σ_k, rather than the JastAdd specification σ_j. As `_matchName` and its related attributes are all specified identically in the Kiama and JastAdd specifications, the proof for this lemma is identical to the proof for Lemma 65. □

Theorem 67. For any node n of type *ClassDecl*, we have the following, where R is the JastAdd equivalent of Kiama's Q.

$$
\begin{aligned}
& [\![n]\!].\mathsf{superClass} \text{ evaluates in } R \text{ steps for any } n && (67.1)\\
& z \text{ is the length of } n\text{'s superclass chain} && (67.2)\\
& f_z \text{ is the length of } n\text{'s } \mathsf{args} && (67.3)\\
& f_{(z-1)} \text{ is the length of } n\text{'s } \mathsf{superClass}\text{'s } \mathsf{args}, \text{ and so on} && (67.4)\\
\implies & \sigma_j \vdash [\![n]\!].\mathsf{remoteLookup}([\![s]\!]) \longrightarrow^k \_ \vdash [\![\_]\!] && (67.5)\\
\wedge\ & 14 \leq k \leq 11 + (6+R)z + 7\sum_{k=1}^{z} f_k && (67.6)
\end{aligned}
$$

Proof. We consider the best-case and worst-case scenarios to find the upper and lower bounds for this evaluation to complete. We will not formally prove why we choose particular best-case and worst-case scenarios, but there is clear intuition that they are correct.

The best-case scenario is when $[\![n]\!].\mathsf{args}$ is some list whose first element's `getName` matches s. The worst-case scenario is when the name being searched is not defined – this means that no nodes with a `name` of s are found in any of the `args` lists on the superclass chain.

We proceed by evaluating the best-case scenario.

$\sigma_j \vdash [\![n]\!].\mathsf{remoteLookup}([\![s]\!])$
$\longrightarrow [\![n_{null}]\!].\_\mathsf{matchName}((.[\![s]\!], [\![n]\!].\mathsf{args}))$
`IFNULL` $[\![n]\!].\mathsf{superClass}.\mathsf{remoteLookup}([\![s]\!])$
{the `args` of n is a list that begins with n_s}
$\longrightarrow^2 [\![n_{null}]\!].\_\mathsf{matchName}([\![(s, n_s :: l)]\!])$
`IFNULL` $[\![n]\!].\mathsf{superClass}.\mathsf{remoteLookup}([\![s]\!])$
$\longrightarrow$ `(IF` $[\![strEq]\!]([\![n_s]\!].\mathsf{getName})([\![s]\!])$ `THEN` $[\![n_s]\!]$ `ELSE` $[\![n]\!].\_\mathsf{matchName}([\![(s,l)]\!])$)
`IFNULL` $[\![n]\!].\mathsf{superClass}.\mathsf{remoteLookup}([\![s]\!])$
{as n_s's `getName` is s}
$\longrightarrow^5$ `(IF` $[\![strEq(s)(s)]\!]$ `THEN` $[\![n_s]\!]$ `ELSE` $[\![n]\!].\_\mathsf{matchName}([\![(s,l)]\!])$)
`IFNULL` $[\![n]\!].\mathsf{superClass}.\mathsf{remoteLookup}([\![s]\!])$
$\longrightarrow [\![n_s]\!]$ `IFNULL` $[\![n]\!].\mathsf{superClass}.\mathsf{remoteLookup}([\![s]\!])$
{as n_s must not be n_{null}, as it is in the list returned by $[\![n]\!].\mathsf{args}$}
$\longrightarrow^5 [\![n_s]\!]$

steps taken = 14

We have proven that the best-case evaluation takes 14 steps.

We proceed in our proof for the worst-case scenario by superclass induction. This gives us a case where n is the Object class, and where n is not.

Case 67.1 (n is the Object class)**.**

$$
\begin{aligned}
&\sigma_j \vdash [\![n]\!].\mathsf{remoteLookup}([\![s]\!]) \\
&\longrightarrow [\![n_{null}]\!].\_\mathsf{matchName}((.[\![s]\!], [\![n]\!].\mathsf{args})) \\
&\quad \texttt{IFNULL}\ [\![n]\!].\mathsf{superClass}.\mathsf{remoteLookup}([\![s]\!]) \\
&\qquad \{\text{the Object class returns the empty list for } \mathsf{args}\} \\
&\longrightarrow [\![n_{null}]\!].\_\mathsf{matchName}((.[\![s]\!], [\![[]]\!])) \\
&\quad \texttt{IFNULL}\ [\![n]\!].\mathsf{superClass}.\mathsf{remoteLookup}([\![s]\!]) \\
&\longrightarrow^2 [\![n_{null}]\!]\ \texttt{IFNULL}\ [\![n]\!].\mathsf{superClass}.\mathsf{remoteLookup}([\![s]\!]) \\
&\longrightarrow^2 [\![n]\!].\mathsf{superClass}.\mathsf{remoteLookup}([\![s]\!]) \\
&\quad \{\text{as } n \text{ is the Object class, its } \mathsf{superClass} \text{ will be } n_{null}\} \\
&\longrightarrow^4 [\![n_{null}]\!].\mathsf{remoteLookup}([\![s]\!]) \\
&\longrightarrow [\![n_{null}]\!]
\end{aligned}
$$

$$
\begin{aligned}
\text{steps taken} &= 11 \\
&= 11 + (6+R)z + 7\sum_{k=1}^{z} f_k
\end{aligned}
$$

Case 67.2 (n is not the Object class)**.**

$$(z-1) \text{ is the length of the superclass chain of } n\text{'s } \mathsf{superClass}\ n_s. \tag{67.7}$$

$$\sigma_j \vdash [\![n_s]\!].\mathsf{remoteLookup}([\![s]\!]) \longrightarrow^{(6+R)(z-1)+7\sum_{k=1}^{z-1} f_k} \_ \vdash [\![n_{null}]\!] \tag{67.8}$$

We proceed by evaluating the expression.

$$
\begin{aligned}
&\sigma_j \vdash [\![n]\!].\mathsf{remoteLookup}([\![s]\!]) \\
&\longrightarrow [\![n_{null}]\!].\_\mathsf{matchName}((.[\![s]\!], [\![n]\!].\mathsf{args})) \\
&\quad \texttt{IFNULL}\ [\![n]\!].\mathsf{superClass}.\mathsf{remoteLookup}([\![s]\!]) \\
&\qquad \{l \text{ is the value of } n\text{'s } \mathsf{args}\} \\
&\longrightarrow^2 [\![n_{null}]\!].\_\mathsf{matchName}([\![(s,l)]\!]) \\
&\quad \texttt{IFNULL}\ [\![n]\!].\mathsf{superClass}.\mathsf{remoteLookup}([\![s]\!])
\end{aligned}
$$

$$
\begin{aligned}
&\{\text{from Lemma 65, as the length of } l \text{ is } f_z\}\\
&\longrightarrow^{7f_z+1} [\![n_{null}]\!]\ \texttt{IFNULL}\ [\![n]\!].\texttt{superClass.remoteLookup}([\![s]\!])\\
&\longrightarrow^{2} [\![n]\!].\texttt{superClass.remoteLookup}([\![s]\!])\\
&\longrightarrow^{R} [\![n_s]\!].\texttt{remoteLookup}([\![s]\!])\\
&\longrightarrow^{11+(6+R)(z-1)+7\sum_{k=1}^{z-1} f_k} [\![n_{null}]\!]
\end{aligned}
$$

$$
\begin{aligned}
\text{steps taken} &= 5 + (7f_z + 1) + R + 11 + (6 + R)(z - 1) + 7\sum_{k=1}^{z-1} f_k\\
&= 11 + 6 + R + (6 + R)(z - 1) + 7f_z + 7\sum_{k=1}^{z-1} f_k\\
&= 11 + (6 + R)z + 7(f_z + \sum_{k=1}^{z-1} f_k)\\
&= 11 + (6 + R)z + 7\sum_{k=1}^{z} f_k
\end{aligned}
$$

We have proved that the step count is 14 in the best-case scenario, and $11 + (6 + R)z + 7\sum_{k=1}^{z} f_k$ in the worst-case scenario. □

Theorem 68. Here we want to prove that the `type` attribute, when evaluated in σ_k on a node of type *EIdn*, will evaluate in a particular number of steps. We are not interested in the evaluation of `fpType`, which takes the declaration of a variable and finds a reference to the declaration of its type, so we assume this will happen in the constant number of steps G. As in Theorem 63, we assume that the `superClass` attribute will evaluate in the fixed number of steps Q.

$$
\begin{aligned}
&[\![n]\!].\texttt{fpType} \text{ evaluates in } G \text{ steps for any } n && (68.1)\\
&[\![n]\!].\texttt{superClass} \text{ evaluates in } Q \text{ steps for any } n && (68.2)\\
&z \text{ is the length of } n\text{'s superclass chain} && (68.3)\\
&x \text{ is the number of names in scope for } n && (68.4)\\
&m \text{ is length of } n\text{'s parent's argument list} && (68.5)\\
\Longrightarrow\ &\sigma_k \vdash [\![n]\!].\texttt{type} \longrightarrow^{k} \_ \vdash [\![\_]\!] && (68.6)\\
\wedge\ &30 + (8 + Q)z + G \leq k \leq 24 + (8 + Q)z + 7m + 7x + G && (68.7)
\end{aligned}
$$

Proof. We proceed by evaluating the expression.

$$
\begin{aligned}
&\sigma_k \vdash [\![n]\!].\texttt{type}\\
&\longrightarrow [\![n]\!].\texttt{varUse.varUseType}\\
&\longrightarrow [\![n_2]\!].\texttt{varUseType}\\
&\longrightarrow [\![n_2]\!].\texttt{findFP}([\![n]\!].\texttt{env}, [\![n_2]\!].\texttt{name}).\texttt{fpType}
\end{aligned}
$$

{from Theorem 64, with z as the length of the superclass chain, and l_m the list of args from the parent *MethodDecl*}

$$\begin{aligned}
&\longrightarrow^{16+(8+Q)z} [\![n_2]\!].\mathsf{findFP}((([\![l_m \mathbin{++} l]\!], [\![n_2]\!].\mathsf{name})).\mathsf{fpType}\\
&\longrightarrow^{3} [\![n_2]\!].\mathsf{findFP}([\![(l_m \mathbin{++} l, s_2)]\!]).\mathsf{fpType}\\
&\longrightarrow [\![n_{null}]\!]._\mathsf{matchName}([\![(s_2, l_m \mathbin{++} l)]\!]).\mathsf{fpType}
\end{aligned}$$

So far, $23 + (8 + Q)z$ steps have been taken. We now consider the best-case and worst-case scenarios for the evaluation of the _matchName attribute. First, we consider the worst-case scenario, where no nodes matching s_2 are found in $l_m \mathbin{++} l$.

$$\begin{aligned}
&\sigma_k \vdash [\![n_{null}]\!]._\mathsf{matchName}([\![(s_2, l_m \mathbin{++} l)]\!]).\mathsf{fpType}\\
&\{\text{using Lemma 66, with } x \text{ as the length of } l \text{ and } m \text{ as the length of } l_m\}\\
&\longrightarrow^{7(m+x)+1} [\![n_{null}]\!].\mathsf{fpType}\\
&\qquad \{\text{with the assumed step count for } \mathsf{fpType}\}\\
&\longrightarrow^{G} [\![_]\!]
\end{aligned}$$

$$\begin{aligned}
\text{steps taken} &= (23 + (8 + Q)z) + (7(m + x) + 1) + G\\
&= 24 + (8 + Q)z + 7m + 7x + G
\end{aligned}$$

We now consider the best-case scenario, where the first item in $l_m \mathbin{++} l$ matches s_2.

$$\begin{aligned}
&\sigma_k \vdash [\![n_{null}]\!]._\mathsf{matchName}([\![(s_2, n_s :: l_2)]\!]).\mathsf{fpType}\\
&\longrightarrow^{6} (\mathtt{IF}\ [\![strEq(s_2)(s_2)]\!]\\
&\qquad \mathtt{THEN}\ [\![n_s]\!]\\
&\qquad \mathtt{ELSE}\ [\![n]\!]._\mathsf{matchName}([\![(s, l_2)]\!])).\mathsf{fpType}\\
&\longrightarrow [\![n_s]\!].\mathsf{fpType}\\
&\qquad \{\text{with the assumed step count for } \mathsf{fpType}\}\\
&\longrightarrow^{G} [\![_]\!]
\end{aligned}$$

$$\begin{aligned}
\text{steps taken} &= (23 + (8 + Q)z) + 7 + G\\
&= 30 + (8 + Q)z + G
\end{aligned}$$

Therefore the expression $[\![n]\!].\mathsf{type}$ evaluates in a number of steps ranging from $30 + (8 + Q)z + G$ to $24 + (8 + Q)z + 7m + 7x + G$. □

Theorem 69. Here we want to prove that the type attribute, when evaluated in σ_j on a node of type *EIdn*, which is contained in a *MethodDecl* subtree, will evaluate in a particular number of steps. We are not interested in the evaluation of type on nodes of type *FPDecl*, which takes the declaration of a variable and finds a reference to the declaration of its type, so we assume this will happen in the constant number of steps F. As in Theorem 67, we

assume that the superClass attribute will evaluate in the fixed number of steps R. We also assume that the node in question does not have a name of "this".

$$
\begin{aligned}
&[\![n]\!].\mathsf{type} \text{ evaluates in } H \text{ steps for any } n \text{ of type } \mathit{FPDecl} && (69.1)\\
&[\![n]\!].\mathsf{superClass} \text{ evaluates in } R \text{ steps for any } n && (69.2)\\
&z \text{ is the length of } n\text{'s superclass chain} && (69.3)\\
&x \text{ is the number of names in scope for } n && (69.4)\\
\Longrightarrow\ &\sigma_j \vdash [\![n]\!].\mathsf{type} \longrightarrow^{k} \_ \vdash [\![\_]\!] && (69.5)\\
\wedge\ &23 \leq k \leq 29 + (6+R)z + 7m + 7\sum_{k=1}^{z} f_k + H && (69.6)
\end{aligned}
$$

Proof. We proceed by evaluating the expression.

$$
\begin{aligned}
&\sigma_j \vdash [\![n]\!].\mathsf{type}\\
&\longrightarrow [\![n]\!].\mathsf{varUse.decl.type}\\
&\longrightarrow [\![n_2]\!].\mathsf{decl.type}\\
&\longrightarrow [\![n_2]\!].\mathsf{lookup}([\![n_2]\!].\mathsf{name}).\mathsf{type}\\
&\longrightarrow [\![n_2]\!].\mathsf{lookup}([\![s_2]\!]).\mathsf{type}\\
&\longrightarrow (\mathtt{IF}\ [\![\mathit{strEq}(s_2)(\text{“this”})]\!]\\
&\qquad \mathtt{THEN}\ [\![n_2]\!].\mathsf{parent.enclosingClassDecl.thisFPDecl}\\
&\qquad \mathtt{ELSE}\ ([\![n_2]\!].\mathsf{\_matchName}(([\![s_2]\!], [\![n_2]\!].\mathsf{parent.args}))\\
&\qquad\ \mathtt{IFNULL}\ [\![n_2]\!].\mathsf{parent.lookup}([\![s_2]\!]))).\mathsf{type}\\
&\qquad\qquad \{\text{we assume that } s_2 \neq \text{“this”}\}\\
&\longrightarrow ([\![n_2]\!].\mathsf{\_matchName}(([\![s_2]\!], [\![n_2]\!].\mathsf{parent.args}))\\
&\qquad\ \mathtt{IFNULL}\ [\![n_2]\!].\mathsf{parent.lookup}([\![s_2]\!])).\mathsf{type}\\
&\qquad\qquad \{l_m \text{ is the list of parameters for the parent method}\}\\
&\longrightarrow^{4} ([\![n_2]\!].\mathsf{\_matchName}([\![(s_2, l_m)]\!])\\
&\qquad\ \mathtt{IFNULL}\ [\![n_2]\!].\mathsf{parent.lookup}([\![s_2]\!])).\mathsf{type}
\end{aligned}
$$

So far, 10 steps have been taken. We now consider the best-case and worst-case scenarios for the rest of evaluation. First, we consider the worst-case scenario, where no nodes matching s_2 are found in l_m.

$$
\begin{aligned}
&\sigma_j \vdash ([\![n_2]\!].\mathsf{\_matchName}([\![(s_2, l_m)]\!])\\
&\qquad\ \mathtt{IFNULL}\ [\![n_2]\!].\mathsf{parent.lookup}([\![s_2]\!])).\mathsf{type}
\end{aligned}
$$

$$
\begin{aligned}
&\{\text{using Lemma 65, where } m \text{ is the length of } l_m\}\\
&\longrightarrow^{7m+1} (\llbracket n_{null} \rrbracket\\
&\qquad \texttt{IFNULL } \llbracket n_2 \rrbracket\texttt{.parent.lookup(}\llbracket s_2 \rrbracket\texttt{)).type}\\
&\quad \longrightarrow^{2} (\llbracket n_2 \rrbracket\texttt{.parent.lookup(}\llbracket s_2 \rrbracket\texttt{)).type}\\
&\{n_m \text{ is the parent of } n_2, \text{ which is a } \textit{MethodDecl} \text{ node}\}\\
&\quad \longrightarrow (\llbracket n_m \rrbracket\texttt{.lookup(}\llbracket s_2 \rrbracket\texttt{)).type}\\
&\quad \longrightarrow (\llbracket n_m \rrbracket\texttt{.parent.remoteLookup(}\llbracket s_2 \rrbracket\texttt{)).type}\\
&\{n_c \text{ is the parent of } n_m, \text{ which is a } \textit{ClassDecl} \text{ node}\}\\
&\quad \longrightarrow (\llbracket n_c \rrbracket\texttt{.remoteLookup(}\llbracket s_2 \rrbracket\texttt{)).type}\\
&\{\text{using Theorem 67, where } 14 \le d \le 11 + (6+R)z + 7\textstyle\sum_{k=1}^{z} f_k\}\\
&\quad \longrightarrow^{d} (\llbracket \_ \rrbracket\texttt{).type}\\
&\qquad \{\text{we aren't interested in this evaluation of } \texttt{type}\}\\
&\quad \longrightarrow^{H} \llbracket \_ \rrbracket
\end{aligned}
$$

$$
\begin{aligned}
\text{steps taken} &= 10 + 7m + 1 + 7 + d + H\\
&= 18 + 7m + d + H\\
&\qquad \{\text{we assume the worst-case value of d}\}\\
&= 18 + 7m + 11 + (6+R)z + 7\sum_{k=1}^{z} f_k + H\\
&= 29 + (6+R)z + 7m + 7\sum_{k=1}^{z} f_k + H
\end{aligned}
$$

We now consider the best-case scenario, where the first item of l_m matches s_2.

$$
\begin{aligned}
&\sigma_j \vdash (\llbracket n_2 \rrbracket\texttt{.\_matchName(}\llbracket (s_2, n_s :: l_r) \rrbracket\texttt{)}\\
&\qquad \texttt{IFNULL } \llbracket n_2 \rrbracket\texttt{.parent.lookup(}\llbracket s_2 \rrbracket\texttt{)).type}\\
&\longrightarrow^{6} (\texttt{(IF } \llbracket strEq(s_2)(s_2) \rrbracket\\
&\qquad \texttt{THEN } \llbracket n_s \rrbracket\\
&\qquad \texttt{ELSE } \llbracket n \rrbracket\texttt{.\_matchName(}\llbracket (s, l_2) \rrbracket\texttt{))}\\
&\qquad \texttt{IFNULL } \llbracket n_2 \rrbracket\texttt{.parent.lookup(}\llbracket s_2 \rrbracket\texttt{)).type}\\
&\longrightarrow \llbracket n_s \rrbracket \texttt{ IFNULL } \llbracket n_2 \rrbracket\texttt{.parent.lookup(}\llbracket s_2 \rrbracket\texttt{)).type}\\
&\longrightarrow^{6} \llbracket n_s \rrbracket
\end{aligned}
$$

$$
\begin{aligned}
\text{steps taken} &= 10 + 13\\
&= 23
\end{aligned}
$$

Therefore the expression $\llbracket n \rrbracket\texttt{.type}$ evaluates in a number of steps ranging from 23 to $29 + (6+R)z + 7m + 7\sum_{k=1}^{z} f_k + H$. □

7.5.1 Comparing Performance

While Saiga was not built with performance in mind, our calculus does allow for formal analyses of attribute grammar evaluation that includes step counts. In Theorems 68 and 69 we produced some upper and lower bounds for step counts of name analysis for an identifier expression (*EIdn*) in the Kiama and JastAdd implementations of Featherweight Java.

In both instances, we generalised away some similar operations, such as superclass lookup and name analysis for classes. While these operations are not always computed in a fixed number of steps, they are similar enough in each specification that we can simplify our analysis by assuming they are a constant, which can be factored away. If we evaluate the `type` attribute under each specification, but on the same tree (as we have for all proofs in this chapter), we know that a number of measurements will be the same between our two step counts.

The Kiama specification's best-case evaluation takes $30 + (8 + Q)z + G$ steps (where Q and G are constants, and z is the length of the superclass chain), and the JastAdd specification's best-case evaluation takes 23 steps. It is clear that, even when z is very small, the Kiama specification will always take longer. This is because the Kiama specification uses the environment approach to name analysis, which constructs a full environment at each point in the tree, which takes some overhead, but can be very useful for caching.

The Kiama specification's worst-case evaluation takes $24 + (8 + Q)z + 7m + 7x + G$ steps, and the JastAdd specification's worst-case evaluation takes $29 + (6 + R)z + 7m + 7\sum_{k=1}^{z} f_k + H$ steps. We can assume that Q and R are similar values, as well as G and H. x is the number of names in scope, m is the number of arguments to the containing method, and f_k is the number of names declared at each point in the scopes around n. Therefore we can say that $\sum_{k=1}^{z} f_k = x$.

We now have a worst-case comparison of $24 + (8 + Q)z + 7m + 7x + G$ and $29 + (6 + Q)z + 7m + 7x + G$, if we combine similar constants G/H and Q/R.

	Best	**Worst**
Kiama	$30 + (8 + Q)z + G$	$24 + (8 + Q)z + 7m + 7x + G$
JastAdd	23	$29 + (6 + Q)z + 7m + 7x + G$

We can see from this analysis that the JastAdd specification outperforms the Kiama specification considerably in the best-case scenario, and the two specifications take a remarkably similar number of steps in the worst-case scenario. The difference in worst-case scenario step counts is only $|5 - 2z|$, where z is the number of superclasses that the class containing n has. It is interesting to observe this similarity, as the two specifications take very different approaches to solve the problem.

Much more analysis could be performed to compare these two specifications, particularly on the effects of caching on repeated name lookups[3], which is left as an exercise for the reader. The analyses we have presented in this chapter demonstrate Saiga's utility as a framework for the formal analysis of attribute grammar evaluation.

[3]Note that the JastAdd 'lookup' style algorithm is, according to the creators, quadratic in a multiple evaluation setting. However, real-world JastAdd implementations use a variant of the 'lookup' approach which caches local scopes as maps, providing much faster evaluation under caching. It would still be interesting to perform a comparison of the simple 'lookup' and 'environment' approaches, and we suspect that the 'environment' approach would significantly outperform the 'naive lookup' approach when many lookups are performed. Of course JastAdd developers are aware of this, which is why they employ the mapping variant.

Those beer nerds are fucking merciless.

Anthony Bourdain

8 Mechanisation

From the beginning of Saiga's development, the on-paper calculus has developed alongside a mechanised counterpart. In the early stages we mechanised our calculus in Coq [53], as mentioned in our first published work on Saiga [51]. In 2018 we moved from Coq to Lean [8], as we had run into some difficulties with the efficiency of proof automation in Coq.

Saiga's mechanisation is not one of our major contributions; we mechanise to sanity-check our on-paper semantics and proofs. However, our approach and our results using Lean may be interesting to the reader, so they are presented here.

We will not present the full code of our mechanisation in this thesis, as we have produced many thousands of lines of definitions and proof scripts. The full code of our mechanisation can be found at `https://bitbucket.org/scottbuckley/saigalean`. The code referenced in Sections 8.1 to 8.5 is defined in the file `saiga.lean`.

8.1 Configuration

The first thing we define in our mechanised semantics is a *configuration*: this is a set of variables that make up an evaluation context. We call this configuration `saigaconfig`, which is defined in the Lean snippet shown below.

```
1 structure saigaconfig :=
2   (A:  Type)
3   (τ ρ:  A -> Type)
4   [Adec: decidable_eq A]
5   [τinh: ∀ (a:A), inhabited τ a)]
6   [ρdec: ∀ (a:A), decidable_eq (ρ a)]
7
8 variable cfg: saigaconfig
```

We also declare the variable `cfg`, which is a global variable of type `saigaconfig`. What this means is that we assume some `cfg` of type `saigaconfig` exists. This way we can write proofs that assume a `saigaconfig` exists, and Lean will automatically generalise these

proofs over `saigaconfig`. The contents of a `saigaconfig` are simple: line 2 specifies that there is some type representing the set of attributes $\boldsymbol{A}$. Line 3 specifies that there are two functions τ and ρ which return a type for every member of $\boldsymbol{A}$. Line 4 specifies that there is decidable equality between members of $\boldsymbol{A}$. Lines 5 and 6 specify that the return types of τ and ρ are inhabited and have decidable equality, respectively.

8.2 Expression Language

We define the expression language of Saiga as an inductive type.

```
1 structure ap :=  mk :: (attr:cfg.A) (param:cfg.ρ attr)
2
3 inductive exp
4 | eVal    {T:Type} (v:T):                              exp
5 | eCond   (eC eT eF:exp):                              exp
6 | eApp    (eF eP: exp):                                exp
7 | eAttr   (eN: exp) (a:cfg.A) (eP:exp):                exp
8 | eCache  (n:node) (a:cfg.A) (p:cfg.ρ a) (e:exp):      exp
9 | eMk     (fl:node -> list (exp × ap cfg)):            exp
```

Lines 3-9 give the definition of the `exp` type, which matches the expression grammar given in Section 5.1.1. This type describes the *abstract* syntax of a Saiga expression, not the *concrete* syntax, so parsing strings such as "IF" and "THEN" are not included here. Each of the constructors of `exp` on lines 4-9 clearly implement one of the productions of the Saiga expression grammar. The type `ap` is a dependent tuple of an attribute and an associated parameter value, used in the definition of `eMk`. In our mechanisation we do not force the attribute written to a higher order node to be parameterless, as expressing $\boldsymbol{A}_u$ would be more complicated in Lean than simply allowing parameterised attributes to be written. Therefore our definition of a `MK` expression differs slightly in our mechanisation to the version presented in Chapter 5.

We define notations for each of the constructors of `exp` to allow expressions to be written as follows.

- `{{`v`}}` in Lean represents $[\![v]\!]$.
- `IFF` e_1 `THEN` e_2 `ELSE` e_3 in Lean represents `IF` e_1 `THEN` e_2 `ELSE` e_3. We use `IFF` instead of `IF` to avoid conflict with Lean keywords.
- e_1 `OF` e_2 in Lean represents e_1`(`e_2`)`. The parameter syntax is more difficult to represent in Lean than an infix.
- e_1 `DOT` a `WITH` e_2 in Lean represents e_1`.`a`(`e_2`)`. Again we prefer an infix notation in Lean. We also define e_1 `DOT` a in Lean to represent e_1`.`a, allowing the unit value to be inserted transparently.
- n `/` a `/` p `:=` e in Lean represents n`.`a`(`p`):=` e. Overloading the `DOT`, `WITH` syntax would be too difficult to achieve in Lean.
- `MK` f_l in Lean represents `MK`f_l. We usually use Lean's inbuilt lambda syntax to define the function f_l.

8.3 Type Rules

We define type inference rules in Lean as an inductive proposition. We show a snippet of this definition below.

```
1 inductive expType {cfg} : exp cfg -> Type -> Prop
2 | eVal {T:Type} {v:T}:
3       expType {{v}} T
4 | eAttr {eN eP:exp cfg} {a:cfg.A}
5     (nt: expType eN (node))
6     (pt: expType eP (cfg.ρ a)):
7       expType (eN DOT a WITH eP) (cfg.τ a)
```

The inductive proposition given above represents a relation between values in `exp` and values in `Type` (types in Lean are how we implement $\boldsymbol{T}$). We can see from the constructor on lines 2-3 that the type of a value expression is taken directly from its contained value, as in TypeVal in Section 5.1.2. Similarly, lines 4-7 define types for attribution expressions using τ, enforcing subexpression types to $\boldsymbol{N}$ and $\rho(a)$. Type inference rules are given for all other expression types, matching the rules presented in Section 5.1.2.

In earlier versions of our mechanisation, we built expression types into the constructors of `exp`, giving `exp` a type parameter. For example, the type `exp bool` would represent an expression of type `bool`. This was easy to express, and allowed Lean's type system to perform all type checking on our expression language. Further, we were able to create a context function type that always returns an expression of a type that matches the input attribute. However, Lean's tactics would often fail when trying to perform induction on these dependent inductive types. This issue was significant enough to cause us to abandon the approach entirely and resort to specifying expression types using a relation as described above.

8.4 The Context Function

We define a context function as a simple (dependent) function in Lean, which takes as parameters a node, an attribute, and a parameter value, returning an expression.

```
def context := ∀ (n:node) (a:cfg.A) (p:cfg.ρ a), exp cfg
```

We then implement the $\oplus$ operator in the function `emptyC` shown below.

```
1 def extendC (n:node) (a:cfg.A) (p:cfg.ρ a) (v:cfg.τ a)
2             (c:context cfg) : context cfg :=
3   λ n' a' p',
4     bite (napmatch cfg n n' a a' p p')
5     ({{v}})
6     (c n' a' p')
```

The inputs to `extendC` are a node, an attribute, and a parameter and value that are typed according to τ and ρ, as well as a context function. The output is a function that matches against the first three inputs, returning either the value `v` given, or deferring to the original context function `c`.

We similarly implement `extendE`, which modifies a context function's output with an expression (rather than specifically a value expression). We then define `writeAL`, which repeatedly applies `extendE`, implementing the $\otimes$ operator.

8.4.1 Node Existence in Lean

Instead of implementing node existence in our Lean mechanisation using the methods discussed in Chapter 5, we instead define a type `ctxN`, which contains both a context function and a list of nodes, such that node existence now depends on a `ctxN`, rather than just a context function. This makes reasoning about node existence simpler during proofs, but means that proofs around higher order attribution differ slightly between our mechanised and on-paper proofs.

Higher order semantics were the hardest part of Saiga to mechanise, and not all of our mechanised proofs around higher order semantics are complete. Specifically, our mechanised proof for cache irrelevance has some holes around our higher order semantics.

8.5 The Step Relation

As with type inference rules, we define the step relation in Lean as an inductive proposition, as shown below. The `step` proposition takes a context function (specifically a `ctxN`, which represents a context function and a list of nodes), an initial expression, a second context function and expression. The proposition `step c1 e2 c2 e2` is equivalent to `c1` $\vdash$ `e1` $\longrightarrow$ `c2` $\vdash$ `e2` in the notation we have introduced in this thesis. In Lean, we implement the notation `c1` $\vDash$ `e1` $\longrightarrow$ `c2` $\vDash$ `e2`, allowing us to express the same proposition in a similar fashion.

```
1  inductive step {cfg}:ctxN cfg -> exp cfg -> ctxN cfg -> exp cfg -> Prop
2  | condTrue {ctx:ctxN _} {eT eF:exp cfg}:
3    step ctx (IFF {{tt}} THEN eT ELSE eF) ctx eT
4
5  | appApp {T1 T2:Type} {ctx:ctxN cfg} {f:(T1 -> T2)} {p:T1}:
6    step ctx ({{f}} OF {{p}}) ctx {{f p}}
7
8  | attrFetchCached {ctx:ctxN _} {a:cfg.A} {n:node} {p:cfg.ρ a}
9    (nv: notvalue (ctx.c n a p)):
10   step ctx ({{n}} DOT a WITH {{p}}) ctx (n/a/p ;= (ctx.c n a p))
11
12 | cacheWrite {ctx:ctxN _} {a:cfg.A} {v:cfg.τ a} {n:node} {p:cfg.ρ a}:
13   step ctx (n / a / p ;= {{v}}) (@extendCN cfg n a p v ctx) {{v}}
```

The `step` relation in Lean has 13 constructors, matching the 13 rules shown in Section 5.1.3. Above, we have shown only four of these constructors. `condTrue` implements the simple CondTrue step. `appApp` implements the FunApp step, where the syntax `f p` indicates function application using Lean's function language (the underlying system in our Lean implementation is, of course, Lean). `attrFetchCached` implements the AttrFetchCached rule, requiring the `notvalue` predicate defined elsewhere. Finally, `cacheWrite` implements the CacheWrite rule, using `extendCN` to update a context function.

Similarly, we define the multistep and big step relations as the inductive propositions `multistep` and `bigstep`.

8.6 Metatheoretic Properties

All of the code referenced in the preceding sections of this chapter is contained in `saiga.lean` in our repository. In `saiga_lemmas.lean` we define many lemmas that are used to aid proofs in this and later sections, none of which are interesting enough to share here. Built upon these lemmas are the metatheoretic theorems contained in `saiga_metatheoretic.lean`.

To give the reader some understanding the process we undertake to write proofs about Saiga, we will walk through the first (and simplest) methatheoretic theorem `value.nostep`, which states that a value expression can not be on the left of a step relation. We first present the entirety of this theorem below, including its proof.

```
1 theorem value.nostep {cfg:saigaconfig}:
2   ∀ {c1 c2:ctxN cfg} {e:exp cfg} {T:Type} {v:T},
3     (c1 ⊨ {{v}} ⟶ c2 ⊨ e) -> false
4 := begin
5   intros c1 c2 e T v hs,
6   cases hs,
7 end
```

The crux of the definition of this theorem is given on line 3, which uses our Lean notation for the single step relation and value expressions to say that a relation with a value expression on its left implies `false`. `false` is the proposition in Lean that can never occur, so saying that something implies `false` is saying that something can not occur. Before this, line 2 says that this theorem holds for any values of `c1` and `c2` (which are context functions), `e` (which is an expression), and `v` (which is of some type `T`).

The proof for `value.nostep` is given between the keywords `begin` and `end`, on lines 5 and 6. Line 5 introduces the variables defined on line 2 and the step relation on line 3 as named hypotheses. Lean's proof state after line 5 is given below.

```
1 1 goal
2 cfg : saigaconfig,
3 c1 c2 : ctxN cfg,
4 e : exp cfg,
5 T : Type,
6 v : T,
7 hs : c1 ⊨ {{v}} ⟶ c2 ⊨ e
8 ⊢ false
```

In the above proof state, line 1 indicates that there is only one goal to solve. Lines 2 to 6 describe the six givens, with their names, and line 7 gives the name `hs` to the hypothesis describing the step relation. Line 8 is the single goal of `false`.

This proof is completed using the `cases` tactic on the hypothesis `hs`, as shown on line 6 of the first code listing of this section. The `cases` tactic replaces the existing goal with one goal for each of the rules that could have created the proposition in question. As there are exactly zero ways that `hs` could have been created, this tactic immediately satisfies the theorem by removing all goals. After applying this tactic, Lean reports `goals accomplished`, indicating that the theorem is proven.

We define and prove a number of related theorems. For each of the following theorems, we will show only the definition of the theorem, not the body of the proof.

8.6.1 Step Determinism

Determinism of the step relation, as discussed in Section 6.3, is specified as follows, and its full proof is shown in `saiga_metatheoretic.lean`.

```
theorem step.determinism {cfg:saigaconfig}:
  ∀ {c1 c2 c3:ctxN cfg} {e1 e2 e3:exp cfg},
    (c1 ⊨ e1 ⟶ c2 ⊨ e2) ->
    (c1 ⊨ e1 ⟶ c3 ⊨ e3) ->
    e3 = e2 ∧ c3 = c2
```

8.6.2 Type Determinism

Determinism of the type relation, as discussed in Section 6.4, is specified as follows, and its full proof is shown in `saiga_metatheoretic.lean`.

```
theorem type.determinism {cfg:saigaconfig}:
  ∀ {e:exp cfg} {T1 T2:Type}
    (t1: expType e T1)
    (t2: expType e T2),
    T2 = T1
```

8.6.3 Type Preservation

Preservation of the type relation over the step relation, as discussed in Section 6.5, is specified as follows, and its full proof is shown in `saiga_metatheoretic.lean`.

```
theorem step.type_preservation {cfg:saigaconfig}:
  ∀ {c1 c2:ctxN cfg} {e1 e2:exp cfg} {T:Type}
    (hts: type_safe c1)
    (ht: expType e1 T)
    (hs: c1 ⊨ e1 ⟶ c2 ⊨ e2),
    expType e2 T
```

8.6.4 Progress

Progress of the step relation, as discussed in Section 6.6, is specified as follows, and its full proof is shown in `saiga_metatheoretic.lean`.

```
theorem step.progress {cfg:saigaconfig}:
  ∀ {c1:ctxN cfg} {e1:exp cfg} {T:Type}
    (hts: type_safe c1)
    (ht: expType e1 T),
    (∃ c2 e2, c1 ⊨ e1 ⟶ c2 ⊨ e2) ∨ value e1 ∨ cont_mk e1
```

8.6.5 Big Step and Multistep

We represent the big step relation (`bigstep` in Lean) using the notation `c1 ⊨ e »» c2 ⊨ v`, and the multistep relation (`multistep` in Lean) using the notation `c1 ⊨ e ⟹ c2 ⊨ v`. Equivalence of the big step and multistep relations, as discussed in Section 6.7, is specified as shown in the following two snippets, and their full proofs are shown in `saiga_metatheoretic.lean`.

```
theorem multistep.bigstep {cfg:saigaconfig}:
  ∀ {c1 c2:ctxN cfg} {e:exp cfg} {T:Type} {v:T}
    (hs: c1 ⊨ e ⟹ c2 ⊨ v),
    (c1 ⊨ e »» c2 ⊨ v)
```

```
theorem bigstep.multistep {cfg:saigaconfig}:
  ∀ {c1 c2:ctxN cfg} {e:exp cfg} {T:Type} {v:T}
    (hs: c1 ⊨ e »» c2 ⊨ v),
    (c1 ⊨ e ⟹ c2 ⊨ v)
```

The approach to proving these theorems in Lean is the same as our proofs of Theorems 17 and 24 in Chapter 6: we prove a lemma about a single step followed by a big step (Lemma 16) to prove `multistep.bigstep`, and we prove a number of similar lemmas (Lemmas 18 to 23) to prove `bigstep.multistep`.

8.7 Cache Irrelevance

As in Section 6.8, our mechanisation approaches cache irrelevance by defining an equivalence relation between context functions and proving that big step evaluation will always produce an output context function that is equivalent to the input context function. Our proofs are found in the Lean files `saiga_irrelprep.lean` and `saiga_irrel.lean`. We state and prove the same series of lemmas we present in Section 6.8, which build upon one another to eventually provide a proof for "big step equivalence".

We mechanise Axiom 1, as well as a number of axioms to implement "node existence" behaviour. For the higher order section of our proof, we also rely on some properties that we take as axioms. Our mechanisation for cache irrelevance is not complete, as we discuss in Section 9.2.

The other day upon the stair;
I met a man who wasn't there.
He wasn't there again today.
I wish that man would go away.

Paraphrased excerpt from Antigonish by
William Hughes Mearns

9 Conclusion

In this thesis we have presented a foundational semantics of dynamically scheduled attribute grammar evaluation, in the calculus we call Saiga. We began in Chapter 2 with a history of attribute grammars, and discussed the state of the art in dynamic evaluation, including discussion of a number of common extensions to attribute grammar platforms. We presented the core version of Saiga in Chapter 3, extended this with parameterised attributes and caching in Chapter 4, and further extended our semantics to include higher order evaluation in Chapter 5.

In Chapter 6 we presented and proved a number of metatheoretic properties about Saiga, in its core, extended, and higher order forms. The most complex property to prove was cache irrelevance, which states that including caching operations (and higher order operations) does not affect the semantics of evaluation for any valid attribute grammar program.

We demonstrated the utility of Saiga in Chapter 7, where we used our calculus to formally compare the type analysis of two compilers for the same language, which were written using two different attribute grammar platforms and using two different approaches to name analysis. We were able to show that a key attribute would always evaluate to the same value for each approach, and we also compared the number of evaluation steps taken by each approach, finding their worst-case step count to be almost identical.

9.1 Evaluation of Contributions

In Section 1.1 we outlined the eight specific contributions of this thesis. Here we will repeat those contributions (in italics), and briefly discuss how and where each contribution was demonstrated.

1. *A **calculus** which captures the fundamental semantics of dynamically-scheduled attribute grammar evaluation, without being obscured by a general purpose language or by a particular attribute grammar platform's implementations.*

   We presented Saiga in Chapters 3 to 5. Saiga models attribute grammar evaluation, focusing on the semantics of things like fetching attributes, writing to cache, and accessing basic function calls, without focusing on the details of those function calls

(by deferring to the underlying system) or the details of the equation selection process (by deferring to the context function). The example in Chapter 7 demonstrated Saiga's ability to specify attributes in notations faithful to different real-world attribute grammar platforms, while being unconcerned with the particulars of either platform's equation selection process.

2. *This calculus is defined in the form of an **expression language**, **type rules**, and a **small-step operational semantics**.*

   Saiga's expression language is presented in its core, extended, and higher order forms in Sections 3.1.3, 4.2.1 and 5.1.1, its type rules in Sections 3.1.4, 4.2.2 and 5.1.2, and its operational semantics in Sections 3.1.5, 4.2.3 and 5.1.3. Each of these sets of specifications is simple in its design, and each variation is discussed in detail, explaining the design decisions that went into their specification, as well as presenting motivating examples to demonstrate their use.

3. *A key feature of the semantics presented is **the context function**, which is a very flexible framework for defining attribute grammar equations that is notation agnostic.*

   The context function is defined for the core calculus in Chapter 3, modified slightly in Chapter 4 to allow for parameterisation, and left unchanged in Chapter 5. We define two simple methods of specifying a modified context function with the $\oplus$ and $\otimes$ operators, without specifying or relying on the inner workings of the function. We present a set of "selector" notations in Section 3.2.2, which are a useful example of how a context function might be defined, but we do not rely on these notations at any point in our calculus – the context function remains open to be defined arbitrarily, as long as it remains a mathematical function.

4. *We begin with a core calculus which we extend by implementing some common and representative attribute grammar **extensions**, including parameterised attributes, attribute caching, and higher order attributes.*

   Parameterised attributes and attribute caching are presented in Chapter 4. Parameterisation is achieved by modifying the context function to take a third parameter, and giving the expression language the ability to specify such a parameter. Attribute caching is achieved by changing our semantics to "write" the computed value of an attribute to the context function during evaluation. This required the addition of the cache expression, a production of the expression language intended only for intermediary use during evaluation. If we had specified big step semantics only, such an expression would not have been necessary, but we were motivated to specify small-step semantics to allow for better quantitative analysis.

   Higher order attributes are presented in Chapter 5. We implement higher order semantics by providing an expression form that specifies a set of attribute equation expressions to be "written" to the context function. This generalised approach allows higher order attributes to be specified in a way that does or does not "tie in" to the existing tree, a generalisation that is important as different attribute grammar platforms differ widely in this regard. The open and "tree-structure-agnostic" nature of the context function allowed higher order attribution to be added to our semantics with relative ease; as the tree structure is not being tracked in any way, modifying the structure can happen without any hassle.

5. *Our calculus represents a framework for reasoning about and comparing the behaviour of attribute grammar programs, aided by a set of **metatheoretic properties**, for which we provide comprehensive proofs.*

   Chapter 6 outlines and proves a number of metatheoretic properties about Saiga, starting with simple properties such as determinism and progress, as well as proven equivalence between the small-step, multistep, and big step semantics. The most complex property to prove was cache irrelevance – the property that writing computed attribute values to the cache will not change the semantic outputs of a context function.

6. *The calculus itself, as well as proofs for the majority of these metatheoretic properties, are **mechanised** in Lean.*

   We present our complete Lean code via external repositories, presenting only small examples in Chapter 8. The core and extended versions of Saiga were mechanised with ease, including their metatheoretic proofs. Proof of cache irrelevance for the extended calculus was mechanised, although not without considerable effort.

   The higher order calculus was also mechanised, with proofs of most of its metatheoretic properties similarly mechanised. The nature of a changing context function, especially when it comes to nodes "existing" or "not existing" in a black-box mathematical function, presented considerable difficulties during mechanisation. As a result, our mechanisation does not include a complete proof of cache irrelevance for the higher order semantics. We believe this proof is possible to mechanise, given enough effort (see Section 9.2).

7. *We **demonstrate the utility** of these techniques through analysis of a real-world scale problem: comparing two different approaches to name and type analysis for Featherweight Java, translated directly from implementations in two different real-world attribute grammar platforms.*

   A major contribution of this thesis, Chapter 7 considers the Featherweight Java language, specifies two different approaches to name and type analysis written for two different attribute grammar platforms, and proves them to be equivalent. Thanks to the maintainers of Kiama and JastAdd, who were kind enough to provide implementations of Featherweight Java in their own platforms, we were able to translate the two different approaches into Saiga specifications, and prove that one particular attribute would always evaluate to the same result, for any Featherweight Java program.

8. *This comparison is primarily a proof that two particular attributes always evaluate to the same value, but we also demonstrate Saiga's facility for **quantitative analysis** by comparing the number of evaluation steps taken for a particular computation.*

   We continue our analysis in Chapter 7 by proving that name binding on Featherweight Java identifier expressions complete in a number of evaluation steps that conform to a specific formula. We show that JastAdd's approach is more efficient in the best-case scenario (when a name's binding is in its smallest containing scope), and that the two approaches are very close to identical in efficiency for the worst-case scenario (when the name is not found in scope).

   This is a major contribution, as the Kiama and JastAdd implementations use two different but accepted name analysis strategies – the "environment" and "lookup" strategies.

Before Saiga, there was no framework for formally analysing such strategies, and the result that their worst-case time complexities are close to identical is an interesting one. Much more analysis could be performed to compare the effects of attribute caching on repeated name lookup evaluations – the example presented in Chapter 7 is merely a demonstration of the kinds of analysis that can be performed using the Saiga calculus.

9.2 Future Work

Based on the work presented in this thesis, there are a number of directions for potential future work.

- **Circular attributes could be implemented using iterative fixed-point evaluation.** This would probably involve the context function returning not only an expression to represent the attribute equation, but also some indication of a starting value for fixed-point computation, perhaps as an Option so that not all attributes allow fixed-point circular evaluation. The fact that normal evaluation wraps all non-value attribute computations in cache expressions provides an existing "log" of the nested stack of attributes being evaluated, which would be a useful starting point for fixed-point evaluation. However in the current approach to our semantics, each expression is evaluated without regard for what expression contains it. To implement fixed-point circular evaluation, we would need to include some extra contextual information to our step relation, to allow evaluation to know when a cycle has been reached.

- **Nodes and attributes could be extended with type information, so that not all attributes are applicable to all nodes.** In the version of Saiga presented in this thesis, we abstract all information away from a node, including its structure and its type. This abstraction has proved useful in providing the flexibility to specify attribute grammar programs without being bound to a particular type system, but we have found that we have been implementing a simple type system inside Saiga in every example we have presented, which may be seen as an indicator that the type system should be part of the calculus. It would be possible to tag each instance of $\boldsymbol{N}$ and each instance of $\boldsymbol{A}$ with a simple enumerated type, and redefine a context function in the following way, such that σ is still a mathematical (total) function, but not all attributes need to be defined on all nodes.

  $$\sigma \in \boldsymbol{N}[q] \to (a : \boldsymbol{A}[q]) \to \rho(a) \to \boldsymbol{E}$$

  Further, this would allow the context function to return different expressions for nodes of different types, without reflecting on intrinsic attributes such as `nodeType`. We opted not to take this approach in the work presented in this thesis, instead choosing to present a simpler calculus. Nevertheless, node types are an important part of almost all real-world attribute grammar program specifications, so our calculus might benefit from modelling them as an explicit part of the calculus, instead of modelling them using the calculus.

- **Our mechanisation of higher order attributes could be improved.** Our experiences with mechanising Saiga, both in the original Coq version and now in Lean, has been that grammars and relations are easy to model as inductive datatypes. The context function, on the other hand, is designed specifically to be a black-box function, which

does not play so well with our mechanisation tools. In our approaches to mechanisation we have implemented a concrete base context function, and created specific functions to implement the $\oplus$ and $\otimes$ operations. This introduces a kind of process ordering to our context functions that does not naturally occur in our "on paper" semantics, and makes proofs about changing context functions more difficult to frame. More work could be done in expressing the context function and the way it changes in our mechanisation, in such a way that allows for more convenient reasoning.

A Appendix

A.1 Name Analysis Example

A.1.1 Kiama's Abstract Grammar for Featherweight Java

A sample of the abstract grammar used by the Kiama implementation.

```
1  object FWJavaParserSyntax {
2    sealed abstract class ASTNode extends Product
3
4    case class Program (optClassDecls : Vector[ClassDecl],
5                                expr : Expr) extends ASTNode
6
7    case class ClassDecl (identifier : String,
8                              idnUse : IdnUse,
9                optFieldOrParamDecls : Vector[FieldOrParamDecl],
10                           ctorDecl : CtorDecl,
11                     optMethodDecls : Vector[MethodDecl])
12     extends ASTNode
13
14   case class CtorDecl (idnUse : IdnUse,
15          optFieldOrParamDecls : Vector[FieldOrParamDecl],
16                     optIdnUses : Vector[IdnUse],
17                  optFieldInits : Vector[FieldInit]) extends ASTNode
18
19   case class MethodDecl (idnUse : IdnUse,
20                       identifier : String,
21             optFieldOrParamDecls : Vector[FieldOrParamDecl],
22                           expr : Expr) extends ASTNode
23
24   case class FieldInit (idnUse1 : IdnUse, idnUse2 : IdnUse)
25     extends ASTNode
```

```
26
27    case class FieldOrParamDecl (idnUse : IdnUse, idnDef : IdnDef)
28      extends ASTNode
29
30    sealed abstract class Expr extends ASTNode with
31      org.bitbucket.inkytonik.kiama.output.PrettyExpression
32    case class Inv (expr : Expr, idnUse : IdnUse, optExprs:Vector[Expr])
33      extends Expr
34      with org.bitbucket.inkytonik.kiama.output.PrettyNaryExpression {
35        val priority = 2
36        val fixity = org.bitbucket.inkytonik.kiama.output.Infix
37          (org.bitbucket.inkytonik.kiama.output.LeftAssoc)
38    }
39    case class Fld (expr : Expr, idnUse : IdnUse) extends Expr
40      with org.bitbucket.inkytonik.kiama.output.PrettyNaryExpression {
41        val priority = 2
42        val fixity = org.bitbucket.inkytonik.kiama.output.Infix
43          (org.bitbucket.inkytonik.kiama.output.LeftAssoc)
44    }
45    case class Cst (idnUse : IdnUse, expr : Expr) extends Expr
46      with org.bitbucket.inkytonik.kiama.output.PrettyNaryExpression {
47        val priority = 1
48        val fixity = org.bitbucket.inkytonik.kiama.output.Infix
49          (org.bitbucket.inkytonik.kiama.output.RightAssoc)
50    }
51    case class New (idnUse : IdnUse, optExprs : Vector[Expr])
52      extends Expr
53      with org.bitbucket.inkytonik.kiama.output.PrettyNaryExpression {
54        val priority = 0
55        val fixity = org.bitbucket.inkytonik.kiama.output.Infix
56          (org.bitbucket.inkytonik.kiama.output.NonAssoc)
57    }
58    case class Idn (idnUse : IdnUse) extends Expr
59      with org.bitbucket.inkytonik.kiama.output.PrettyNaryExpression {
60        val priority = 0
61        val fixity = org.bitbucket.inkytonik.kiama.output.Infix
62          (org.bitbucket.inkytonik.kiama.output.NonAssoc)
63    }
64
65    case class IdnUse (identifier : String) extends ASTNode
66    case class IdnDef (identifier : String) extends ASTNode
67  }
```

A.1.2 JastAdd's Abstract Grammar for Featherweight Java

A sample of the abstract grammar used by the JastAdd implementation.

```
1   Program ::= ClassDecl* [Expr];
2
3   abstract TypeDecl;
4   ClassDecl : TypeDecl ::= <Name> Extends:TypeUse
5                            FPDecl* CtorDecl MethodDecl*;
6   FPDecl ::= TypeUse IdnDef;
7   CtorDecl ::= TypeUse FPDecl* Super:VarUse* FieldInit*;
8   FieldInit ::= <FieldName> <ParName>;
9   MethodDecl ::= TypeUse <Name> FPDecl* Expr;
10
11  abstract Expr;
12  EIdn  : Expr ::= VarUse;
13  EFld  : EIdn ::= Expr;
14  ECall : Expr ::= Expr VarUse Argument:Expr*;
15  ENew  : Expr ::= TypeUse Argument:Expr*;
16  ECast : Expr ::= TypeUse Expr;
17
18  VarUse ::= <Name>;
19  TypeUse ::= <Name>;
20  IdnDef ::= <Name>;
21
22  UnknownTypeDecl : TypeDecl;
```

A.2 Featherweight Java Code in Kiama and JastAdd

A.2.1 Kiama

```
1     // Names via environments
2
3     val env : ASTNode => Defs =
4         attr {
5             case l @ ClassDecl(c, _, _, _, _) =>
6             FPDecl(TypeUse(c), IdnDef("this")) +: fieldsRef(l).get
7
8             case k : CtorDecl =>
9                 k.optFPDecls
10
11            case parent.pair(m : MethodDecl, p) =>
12                m.optFPDecls ++ env(p)
13
14            case parent(p) =>
15                env(p)
16
17            case _ =>
```

```
18                 Vector()
19         }
20
21     // Types
22
23     def findType(defs : Defs, x : String) : Option[String] =
24         defs.collectFirst {
25             case FPDecl(TypeUse(c), IdnDef(y)) if x == y =>
26                 c
27         }
28
29     def findFP(defs : Defs, x : String) : Option[FPDecl] =
30         defs.collectFirst {
31             case fp @ FPDecl(_, IdnDef(y)) if x == y =>
32                 fp
33         }
34
35
36     val varUseTypeRef : VarUse => Option[ClassDecl] =
37         attr {
38             case u @ VarUse(x) =>
39                 findFP(env(u), x) match {
40                     case Some(FPDecl(TypeUse(y), _)) =>
41                         decl(y)(u)
42                     case None =>
43                         None
44                 }
45         }
46
47     val tipeRef : Expr => Option[ClassDecl] =
48         attr {
49             // VAR
50             case EIdn(u) =>
51                 varUseTypeRef(u)
52
53             // FLD
54             case n @ EFld(e, VarUse(f)) =>
55                 tipeRef(e) match {
56                     case Some(c) =>
57                         fieldsRef(c) match {
58                             case Some(defs) =>
59                                 findFP(defs, f) match {
60                                     case Some(FPDecl(TypeUse(y), _))=>
61                                         decl(y)(n)
62                                     case None =>
63                                         None
64                                 }
65                             case _ =>
```

```
66                              None
67                          }
68                      case None =>
69                          None
70                  }
71
72              // CALL
73              case n @ ECall(e, VarUse(m), es) =>
74                  tipeRef(e) match {
75                      case Some(c) =>
76                          method(m)(c) match {
77                              case Some(MethodDecl(
78                                        TypeUse(r), _, defs, _)) =>
79                                  if (esubtypes((es, defs))(n))
80                                      decl(r)(n)
81                                  else
82                                      None
83                              case None =>
84                                  None
85                          }
86                      case None =>
87                          None
88                  }
89
90              // NEW
91              case n @ ENew(TypeUse(c), es) =>
92                  decl(c)(n) match {
93                      case Some(c) =>
94                          fieldsRef(c) match {
95                              case Some(flds) =>
96                                  if (esubtypes((es, flds))(n))
97                                      Some(c)
98                                  else
99                                      None
100                             case _ =>
101                                 None
102                         }
103                     case None =>
104                         None
105                 }
106
107             // UCAST, DCAST
108             case n@ECast(TypeUse(c), e) =>
109                 tipeRef(e) match {
110                     case Some(d) =>
111                         decl(c)(n)
112                     case None =>
113                         None
```

```
114                 }
115
116         }
117
118
119     val tipeName : Expr => Option[String] =
120         attr {
121             case e =>
122                 tipeRef(e) match {
123                     case Some(ClassDecl(id, _, _, _, _)) =>
124                         Some(id)
125                     case None =>
126                         None
127                 }
128         }
129
130     // ClassDeclookups
131
132     val decl : String => ASTNode => Option[ClassDecl] =
133         paramAttr {
134             case "Object" => {
135                 case _ =>
136                     Some(ClassDecl("Object",
137                     TypeUse("Object"),
138                     Vector(),
139                     CtorDecl(
140                         TypeUse("Object"),
141                         Vector(),
142                         VarUses(Vector()),
143                         FieldInits(Vector())
144                     ),
145                     Vector()
146                     ))
147             }
148             case c => {
149                 case parent(p) =>
150                     decl(c)(p)
151                 case Program(classDecls, _) => {
152                     classDecls.find
153                        (classDecl => className(classDecl) == c)
154                 }
155             }
156         }
157
158     val method : String => ClassDecl => Option[MethodDecl] =
159         paramAttr {
160             case m => {
161                 case ClassDecl("Object", _, _, _, _) =>
```

```
162                     None
163                 case l1 @ ClassDecl(_, _, _, _, ms) =>
164                     ms.find(_.identifier == m) match {
165                         case Some(meth) =>
166                             Some(meth)
167                         case None =>
168                             superClass(l1) match {
169                                 case Some(l2) =>
170                                     method(m)(l2)
171                                 case None =>
172                                     None
173                             }
174                     }
175             }
176         }
177
178
179     val superClass : ClassDecl => Option[ClassDecl] =
180         attr {
181             case n @ ClassDecl(_, TypeUse(sc), _, _, _) =>
182                 decl(sc)(n)
183         }
184
185     val fieldsRef : ClassDecl => Option[Defs] =
186         attr {
187             case ClassDecl("Object", _, _, _, _) =>
188                 Some(Vector())
189             case n @ ClassDecl(_, i @ TypeUse(sc), defs1, _, _) => {
190                 decl(sc)(n) match {
191                     case Some(sup) =>
192                         fieldsRef(sup) match {
193                             case Some(defs2) =>
194                                 Some(defs1 ++ defs2)
195                             case None =>
196                                 Some(defs1)
197                         }
198                     case None =>
199                         None
200                 }
201             }
202         }
203
204     // Sub-typing
205
206     val esubtypes : CachedParamAttribute
207                     [(Vector[Expr], Defs), ASTNode, Boolean] =
208         paramAttr {
209             case (es, defs) => {
```

```
210                 case n =>
211                     val ts = es.map(tipeRef)
212                     (ts.length == defs.length) &&
213                         (!ts.contains(None)) &&
214                         subtypes((ts.map(_.get), defs))(n)
215             }
216         }
217
218     val subtypes: CachedParamAttribute
219                     [(Vector[ClassDecl], Defs), ASTNode, Boolean] =
220         paramAttr {
221             case (ts, defs) => {
222                 case n =>
223                     ts.zip(defs).forall {
224                         case (c, FPDecl(TypeUse(sn), _)) =>
225                             subtypeOf(sn)(c)
226                     }
227             }
228         }
229
230     val subtypeOf : CachedParamAttribute[String, ClassDecl, Boolean] =
231         paramAttr {
232             case "Object" => {
233                 case _ => true
234             }
235             case pn => {
236                 case ClassDecl("Object", _, _, _, _) => false
237                 case ClassDecl(pn2, _, _, _, _) if pn2 == pn =>
238                     true
239                 case n => {
240                     superClass(n) match {
241                         case Some(pc) =>
242                             subtypeOf(pn)(pc)
243                         case None =>
244                             false
245                     }
246                 }
247             }
248         }
```

A.2.2 JastAdd

```
aspect TypeLookup {
        syn TypeDecl TypeUse.decl() = lookupType(getName());
        inh TypeDecl TypeUse.lookupType(String name);
        eq Program.getChild().lookupType(String name) {
                if ("Object".equals(name)) {
                        return objectClassDecl();
                }
                for (ClassDecl cd: getClassDecls()) {
                        if (cd.getName().equals(name)) {
                                return cd;
                        }
                }
                return unknownTypeDeclNTA();
        }

        syn nta ClassDecl Program.objectClassDecl() {
                return new ClassDecl(
                        "Object",
                        new TypeUse(""),
                        new List(),
                        new CtorDecl(
                                new TypeUse("Object"),
                                new List(),
                                new List(),
                                new List()
                        ),
                        new List()
                );
        }

        syn nta UnknownTypeDecl Program.unknownTypeDeclNTA()
                = new UnknownTypeDecl();
        inh UnknownTypeDecl ASTNode.unknownTypeDecl();
        eq Program.getChild().unknownTypeDecl()
                = unknownTypeDeclNTA();
        syn boolean TypeDecl.isUnknown() = false;
        eq UnknownTypeDecl.isUnknown() = true;
}

aspect VariableLookup {
        syn FPDecl VarUse.decl() = lookup(getName());
        inh FPDecl VarUse.lookup(String name);
        inh FPDecl MethodDecl.lookup(String name);
        eq MethodDecl.getExpr().lookup(String name) {
                if (name.equals("this")) {
                        return enclosingClassDecl().thisDecl();
```

```
47                  }
48                  for (FPDecl p: getFPDecls()) {
49                          if (p.getIdnDef().getName().equals(name)) {
50                                  return p;
51                          }
52                  }
53                  // FJ paper forces fields to be accessed via "this".
54                  // Thus, the following is not needed, which allows
55                  // fields to be accessed without "this". However,
56                  // this is the same semantics as in "Scopes as Types"
57                  // paper. Just return null if this behaviour is not
58                  // wanted.
59                  return lookup(name);
60          }
61          eq ClassDecl.getMethodDecl().lookup(String name)
62                  = remoteLookup(name);
63          eq EFld.getVarUse().lookup(String name)
64                  = getExpr().type().remoteLookup(name);
65          eq Program.getChild().lookup(String name)
66                  = null;
67
68          syn FPDecl TypeDecl.remoteLookup(String name) = null;
69          eq ClassDecl.remoteLookup(String name) {
70                  for (FPDecl fd: getFPDecls()) {
71                          if (fd.getIdnDef().getName().equals(name)) {
72                                  return fd;
73                          }
74                  }
75                  return superClass().remoteLookup(name);
76          }
77
78          syn nta FPDecl ClassDecl.thisDecl()
79                  = new FPDecl(new TypeUse(getName()),
80                               new IdnDef("this"));
81
82          inh ClassDecl MethodDecl.enclosingClassDecl();
83          eq ClassDecl.getMethodDecl().enclosingClassDecl() = this;
84  }
85
86  aspect MethodLookup {
87          syn MethodDecl ECall.decl()
88            = getExpr().type().lookupMethod(getVarUse().getName());
89
90          syn MethodDecl TypeDecl.lookupMethod(String name) = null;
91          eq ClassDecl.lookupMethod(String name) {
92                  for (MethodDecl md: getMethodDecls()) {
93                          if (md.getName().equals(name)) {
94                                  return md;
```

```
95                          }
96                  }
97                  return superClass().lookupMethod(name);
98          }
99  }
100
101 aspect TypeAnalysis {
102         syn String TypeDecl.typeName();
103         eq ClassDecl.typeName()       = getName();
104         eq UnknownTypeDecl.typeName() = "";
105
106         syn TypeDecl Expr.type();
107         eq ENew.type() {
108                 if (getTypeUse().decl().isUnknown()) {
109                         return unknownTypeDecl();
110                 }
111                 ClassDecl cd = (ClassDecl) getTypeUse().decl();
112                 if (cd.fields().size() != getNumArgument()) {
113                         return unknownTypeDecl();
114                 }
115                 for (int i = 0; i < cd.fields().size(); i++) {
116                         TypeDecl argTd = getArgument(i).type();
117                         String fieldName = cd.fields().get(i)
118                                                 .getTypeUse().getName();
119                         if (!argTd.subtypeOf(fieldName)) {
120                                 return unknownTypeDecl();
121                         }
122                 }
123                 return cd;
124         }
125         eq ECall.type() {
126                 MethodDecl md = decl();
127                 if (md == null) {
128                         return unknownTypeDecl();
129                 }
130                 if (getNumArgument() != md.getNumFPDecl()) {
131                         return unknownTypeDecl();
132                 }
133                 for (int i = 0; i < getNumArgument(); i++) {
134                         TypeDecl argTd = getArgument(i).type();
135                         String parName = md.getFPDecl(i).getTypeUse()
136                                                 .getName();
137                         if (!argTd.subtypeOf(parName)) {
138                                 return unknownTypeDecl();
139                         }
140                 }
141                 return md.returnType();
142         }
```

```
143         // The following equation is valid for EFld too
144         eq EIdn.type()
145                 = getVarUse().decl() != null
146                 ? getVarUse().decl().type()
147                 : unknownTypeDecl();
148         eq ECast.type()
149                 = !getExpr().type().isUnknown()
150                 ? getTypeUse().decl()
151                 : unknownTypeDecl();
152
153         syn TypeDecl MethodDecl.returnType() = getTypeUse().decl();
154         syn TypeDecl FPDecl.type() = getTypeUse().decl();
155
156         // Circularity check
157         syn boolean TypeDecl.hasCycleOnSuperclassChain()
158                     circular [true] = false;
159         eq ClassDecl.hasCycleOnSuperclassChain()
160                 = getExtends().decl().hasCycleOnSuperclassChain();
161
162         // Only following super if there is no cycle,
163         // making it easier to write other equations
164         syn TypeDecl ClassDecl.superClass() =
165                 !hasCycleOnSuperclassChain() ? getExtends().decl()
166                                              : unknownTypeDecl();
167
168         // Subtyping
169         // syn boolean TypeDecl.isSubtypeOf(TypeDecl other);
170         syn boolean TypeDecl.subtypeOf(String other);
171         // eq UnknownTypeDecl.isSubtypeOf(TypeDecl other) = false;
172         eq UnknownTypeDecl.subtypeOf(String other) = false;
173         // eq ClassDecl.isSubtypeOf(TypeDecl other)
174                 // = this == other || superClass().isSubtypeOf(other);
175         eq ClassDecl.subtypeOf(String other) {
176                 if (other.equals("Object"))     return true;
177                 // object is not a subclass of anything except object
178                 if (getName().equals("Object")) return false;
179                 if (getName().equals(other))    return true;
180                 return superClass().subtypeOf(other);
181         }
182
183         // Type equality
184         syn boolean TypeDecl.equalsTo(TypeDecl other);
185         // We cannot type these
186         eq UnknownTypeDecl.equalsTo(TypeDecl other) = false;
187         eq ClassDecl.equalsTo(TypeDecl other) = this == other;
188 }
```

References

[1] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning. *The state of the art in language workbenches*. In M. Erwig, R. F. Paige, and E. Van Wyk, eds., *Software Language Engineering*, pp. 197–217 (Springer International Publishing, Cham, 2013).

[2] D. E. Knuth. *Semantics of context-free languages*. Mathematical systems theory **2**(2), 127 (1968). URL https://doi.org/10.1007/BF01692511.

[3] M. Jourdan. *An optimal-time recursive evaluator for attribute grammars*. In *Proceedings of the International Symposium on Programming*, Lecture Notes in Computer Science, pp. 167–178 (Springer Berlin Heidelberg, 1984).

[4] G. Hedin. *Reference attributed grammars*. Informatica (Slovenia) **24**(3) (2000).

[5] J. T. Boyland. *Remote attribute grammars*. Journal of the ACM (JACM) **52**(4), 627 (2005).

[6] A. M. Sloane, L. C. L. Kats, and E. Visser. *A pure embedding of attribute grammars*. Science of Computer Programming **78**(10), 1752 (2013).

[7] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. *Silver: an extensible attribute grammar system*. Electronic Notes in Theoretical Computer Science **203**(2), 103 (2008).

[8] L. de Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer. *The Lean theorem prover (system description)*. In *International Conference on Automated Deduction*, pp. 378–388 (Springer, 2015).

[9] A. Igarashi, B. C. Pierce, and P. Wadler. *Featherweight Java: a minimal core calculus for Java and GJ*. ACM Transactions on Programming Languages and Systems (TOPLAS) **23**(3), 396 (2001).

[10] P. M. Lewis, II and R. E. Stearns. *Syntax-directed transduction*. J. ACM **15**(3), 465 (1968). URL http://doi.acm.org/10.1145/321466.321477.

[11] D. E. Knuth. *The genesis of attribute grammars*. In P. Deransart and M. Jourdan, eds., *Attribute Grammars and their Applications*, pp. 1–12 (Springer, Berlin, Heidelberg, 1990).

[12] P. Deransart, M. Jourdan, and B. Lorho. *Attribute Grammars: Definitions, Systems and Bibliography* (Springer-Verlag, Berlin, Heidelberg, 1988).

[13] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. *Attributed translations (extended abstract)*. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pp. 160–171 (ACM, New York, NY, USA, 1973). URL http://doi.acm.org/10.1145/800125.804047.

[14] D. E. Knuth. *Semantics of context-free languages: Correction*. Theory of Computing Systems **5**(2), 95 (1971).

[15] K. Kennedy and S. K. Warren. *Automatic generation of efficient evaluators for attribute grammars*. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 32–49 (ACM, 1976).

[16] B. Courcelle and P. Franchi-Zannettacci. *Attribute grammars and recursive program schemes I.* Theoretical Computer Science **17**(2), 163 (1982). URL http://www.sciencedirect.com/science/article/pii/0304397582900032.

[17] M. Jourdan. *Strongly non-circular attribute grammars and their recursive evaluation*. In *ACM SIGPLAN Notices*, vol. 19, pp. 81–93 (ACM, 1984).

[18] B. H. Mayoh. *Attribute grammars and mathematical semantics*. SIAM Journal on Computing **10**(3), 503 (1981).

[19] *Attribute grammars and their applications*. In P. Deransart and M. Jourdan, eds., *Proceedings of the International Symposium on Programming*, no. 461 in Lecture Notes in Computer Science (Springer, 1990).

[20] U. Kastens. *Ordered attributed grammars*. Acta Informatica **13**(3), 229 (1980).

[21] H. Alblas and B. Melichar. *Attribute Grammars, Applications and Systems: International Summer School SAGA, Prague, Czechoslovakia, June 4-13, 1991. Proceedings*, vol. 545 (Springer Science & Business Media, 1991).

[22] J. Engelfriet and G. Filé. *Simple multi-visit attribute grammars*. Journal of Computer and System Sciences **24**(3), 283 (1982).

[23] U. Kastens. *Ordered attributed grammars*. Tech. rep., Bericht No. 7/78, Universität Karlsruhe (1978).

[24] G. Filé. *Classical and incremental attribute evaluation by means of recursive procedures*. Theoretical Computer Science **53**(1), 25 (1987).

[25] K. Barbar. *Étude comparative de différentes classes de grammaire d'attributs ordonnées*. Ph.D. thesis, Université de Bordeaux (1982).

[26] J. T. Boyland. *Conditional attribute grammars*. ACM Transactions on Programming Languages and Systems (TOPLAS) **18**(1), 73 (1996).

[27] O. L. Madsen. *On defining semantics by means of extended attribute grammars*. In *International Workshop on Semantics-Directed Compiler Generation*, pp. 259–299 (Springer, 1980).

[28] F. Jalili. *A general linear-time evaluator for attribute grammars*. ACM SIGPLAN Notices **18**(9), 35 (1983).

[29] J. Saraiva and S. Swierstra. *Purely functional implementation of attribute grammars* (Universiteit Utrecht, 1999).

[30] G. Hedin and E. Magnusson. *JastAdd—a Java-based system for implementing front ends*. Electronic Notes in Theoretical Computer Science **44**(2), 59 (2001).

[31] G. Hedin and E. Magnusson. *JastAdd—an aspect-oriented compiler construction system*. Science of Computer Programming **47**(1), 37 (2003).

[32] T. Ekman and G. Hedin. *The JastAdd system — modular extensible compiler construction*. Science of Computer Programming **69**(1-3), 14 (2007).

[33] A. Baars, D. Swierstra, and A. Löh. *UU AG system user manual* (2003).

[34] J. Paakki. *Attribute grammar paradigms—a high-level methodology in language implementation*. ACM Computing Surveys (CSUR) **27**(2), 196 (1995).

[35] JastAdd Team. *JastAdd concept overview*. URL http://jastadd.org/web/documentation/concept-overview.php.

[36] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala* (Artima Inc, 2008).

[37] A. Poetzsch-Heffter. *Programming language specification and prototyping using the MAX system*. In *International Symposium on Programming Language Implementation and Logic Programming*, pp. 137–150 (Springer, 1993).

[38] J. T. Boyland. *Descriptional composition of compiler components*. Ph.D. thesis (1996).

[39] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. *Higher order attribute grammars*. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pp. 131–145 (ACM, New York, NY, USA, 1989). URL http://doi.acm.org/10.1145/73141.74830.

[40] S. D. Swierstra and H. Vogt. *Higher order attribute grammars*. In *Proceedings on Attribute Grammars, Applications and Systems*, pp. 256–296 (Springer-Verlag, London, UK, 1991). URL http://dl.acm.org/citation.cfm?id=645843.668460.

[41] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. *Forwarding in attribute grammars for modular language design*. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pp. 128–142 (Springer-Verlag, Berlin, Heidelberg, 2002). URL http://dl.acm.org/citation.cfm?id=647478.727933.

[42] R. Farrow. *Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars*. In *ACM SIGPLAN Notices*, vol. 21, pp. 85–98 (ACM, 1986).

[43] L. G. Jones. *Efficient evaluation of circular attribute grammars*. ACM Transactions on Programming Languages and Systems (TOPLAS) **12**(3), 429 (1990).

[44] E. Magnusson and G. Hedin. *Circular reference attributed grammars—their evaluation and applications*. Science of Computer Programming **68**(1), 21 (2007).

[45] O. de Moor, S. Peyton-Jones, and E. V. Wyk. *Aspect-oriented compilers*. In *Generative and Component-Based Software Engineering*, pp. 121–133 (Springer, Berlin, Heidelberg, 1999).

[46] O. de Moor, K. Backhouse, and S. D. Swierstra. *First-class attribute grammars*. Informatica **24** (2000).

[47] T. Johnsson. *Attribute grammars as a functional programming paradigm*. In *Proceedings of the Functional Programming Languages and Computer Architecture*, pp. 154–173 (1987).

[48] K. Backhouse. *A functional semantics of attribute grammars*. In *Tools and Algorithms for the Construction and Analysis of Systems*, no. 2280 in Lecture Notes in Computer Science, pp. 142–157 (Springer Berlin Heidelberg, 2002).

[49] M. Schäfer, T. Ekman, and O. de Moor. *Formalising and verifying reference attribute grammars in Coq*. In *European Symposium on Programming*, pp. 143–159 (Springer, 2009).

[50] P. Martins, J. a. P. Fernandes, J. a. Saraiva, E. Van Wyk, and A. Sloane. *Embedding attribute grammars and their extensions using functional zippers*. Science of Computer Programming **132, Part 1**, 2 (2016).

[51] S. J. H. Buckley and A. M. Sloane. *A formalisation of parameterised reference attribute grammars*. In *International Conference on Software Language Engineering (SLE)*, pp. 139–150 (ACM Press, New York, New York, USA, 2017).

[52] A. M. Sloane, F. Cassez, and S. Buckley. *The sbt-rats parser generator plugin for Scala (tool paper)*. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, pp. 110–113 (ACM, 2016).

[53] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, *et al. The Coq proof assistant reference manual*. INRIA, version **6**(11) (1999).