ALANLIGHT

sound, functionally correct, bounded acyclic data flow modeling

Master's Thesis

G.J. Kunst

ALANLIGHT
sound, functionally correct, bounded acyclic data flow modeling

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

G.J. Kunst
born in Gouda, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

M-industries
Rotterdamseweg 183c
Delft, the Netherlands
`www.m-industries.com`

ALANLIGHT

sound, functionally correct, bounded acyclic data flow modeling

Author:	G.J. Kunst
Student id:	1552147
Email:	`gjkunst@gmail.com`

Abstract

For programs controlling industrial processes, it is of vital importance that they produce results conform their functional specification. Furthermore, it is important that their running times are bounded, and that we can predict corresponding worst-case scenarios. Programs written in general purpose programming languages can crash or produce erroneous output. Data modeling and query languages are typically more restrictive. However, they either do not guarantee soundness (where values are of a predefined indivisible type) or functional correctness (including deterministic output). Alternatively, they have unbounded or unpredictable worst-case running times, or have limited expressiveness.

We present ALANLIGHT, a data modeling language for expressing complex recursive calculations, while guaranteeing soundness, functional correctness, and polynomial time complexity in the size of user data. To achieve this, we use complex referential integrity constraints and an elegant, formally defined analysis over constraint and calculation definitions in ALANLIGHT programs. Furthermore, we give a formal specification of the dynamic semantics of ALANLIGHT, implying its guarantees, and demonstrating support for on-demand minimal effort calculation.

Preface

Several months ago, after mostly recovering from an accident, I considered it near impossible for me to ever finish my master's. The master thesis was the final report that I needed to write for the completion of my master's at Delft University of Technology. But, I was without a thesis supervisor, and unable to find someone to guide me to the completion of said thesis; my former supervisor had left the university at the time. As somewhat of a last resort I contacted Eelco, the chair of the Programming Languages Group. He referred me to Daco who soon became my thesis supervisor, and whom in retrospect I consider my thesis' saviour.

Therefore, first and foremost I want to thank Daco for supervising my thesis, and for the invaluable times we had, discussing not only my thesis but also his work, amongst many other shared interests. I want to thank Eelco for referring me to Daco, giving me a chance to finish my thesis. Furthermore, I want to thank Matthijs for being part of the thesis committee, and everyone else from the university that in any way contributed to the completion of my master's.

The thesis project was executed at M-industries, where I have worked for over five years now. I especially want to thank Corno, my company supervisor and founder of M-industries, for his valuable ideas, insights, and the great discussions we had and still have, and also for giving me the time and freedom to complete my thesis. Without him, this thesis would not have been possible. I also want to thank my other colleagues at M-industries for the interest they have shown in my work, for the fun times we had at the office, and for their valuable feedback on the design of ALAN and ALANLIGHT.

Last, but certainly not least, I want to thank my loving wife Corine and my two beautiful daughters Elena and Emma; my wife, for giving me unparalleled support and freedom, meals, tea, and coffee needed for finishing this work, and my daughters for their many hugs, kisses, and for being amazing reminders that there is more to life than work and study. Finally, I also want to thank my parents for their continued support and help in completing my master's, and other family members for pushing me towards completing this work.

G.J. Kunst
Delft, the Netherlands
January 19, 2018

Contents

List of Figures

Chapter 1

Introduction

In today's industry, machines are generally controlled automatically. We give instructions to machines and they autonomously produce requested results. To further automate control, we connect machines to each other. This enables them to exchange data for determining what they should do, without the need for human intervention.

Different machines require different kinds of data in different formats, and use that data for different purposes. Converting and using data typically involves running programs that perform complex calculations on large datasets which are subject to frequent modification.

It is of vital importance to industrial processes that these programs always produce correct results. That is, that these programs do not crash, and that they produce output that conforms to their *functional* specification. Furthermore, it is important that their running time is bounded, and that we can predict worst-case scenarios. The number of calculations that programs perform – and their execution time by extension – should not increase exponentially when the input changes. A program that crashes, produces incorrect results, or runs for long periods of time, can disrupt or halt industrial processes. Sometimes, long running times are inevitable. In such cases, it is important that we can predict worst-case running times, so that we can take these running times into account when configuring industrial processes.

Programs written in general purpose programming languages (such as C++ [67] and JAVA [17]) can crash or produce erroneous output. Furthermore, their running time is not bounded. Running times can increase exponentially depending on the input. Programs may not even terminate, as they can contain infinite loops. We can use program verification [32, 34] for proving that a single program does not crash and produces correct output (and thus terminates), but this requires additional effort. For guaranteeing sub-exponential running time bounds, some programs allow deriving time bounds automatically [33, 56]. But, for other programs we have to do this manually, as corresponding languages provide no such guarantee.

In this thesis, we present ALANLIGHT, a declarative data modeling language for expressing complex recursive calculations, while guaranteeing *type soundness* (where values are of a predefined indivisible type), *functional correctness* (including deterministic output), and *polynomial time complexity* in the size of user data. By a predefined indivisible type we mean that output values of calculations are not of a nullable type, error type, 'unknown' value type, option type [48, 68], or similar. The running time of ALANLIGHT programs is bounded by the *size* of the specification (the program) and the *size* of user data, enabling accurate worst-case running time predictions for changing data.

In addition, ALANLIGHT supports on-demand minimal effort calculation of derived (calculated) values. This means that an ALANLIGHT program performs exactly those calculations that lead to a final result that a user (or system) demands. Furthermore, ALANLIGHT programs perform calculations *at most once* during program execution.

Our research originates from the need to guarantee termination for calculations which are required for data-exchange among machines and supporting systems. The key idea behind our approach is ensuring that programs describe an acyclic data flow. To achieve this while enabling complex recursive calculations, ALANLIGHT supports expressing complex referential integrity constraints. Furthermore, the language includes an elegant, formally defined analysis over constraint and calculation definitions in programs.

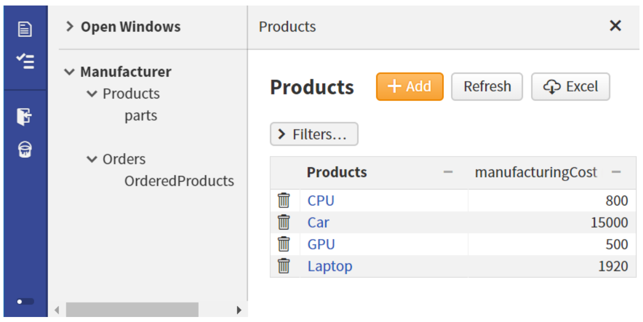
Existing approaches for supporting complex recursive calculations in databases either do not guarantee termination [25, 44, 47, 50], require manually bounding recursion to ensure termination [53], use methods involving unneeded calculations [21], do not (generally) guarantee functionally correct output [2, 5, 18, 23], cannot guarantee output of a predefined indivisible type [18, 53], or limit expressiveness in such a way that some typical problems become inexpressible [18, 38, 59]. Our approach aims to overcome these (and more) problems. In particular, our contributions are:

- ALANLIGHT, a hierarchical data modeling language with integrated support for expressing complex recursive calculations, while guaranteeing soundness, functional correctness, and bounded, predictable running times
- A formal static semantics for ALANLIGHT, defining an analysis over constraint and calculation definitions
- A formal operational semantics for ALANLIGHT, demonstrating support for on-demand minimal effort evaluation
- Complexity bounds for calculations that ALANLIGHT programs perform, demonstrating a polynomial time complexity in the size of user data, and an exponential time complexity in the size of a program
- An evaluation of the impact of referential integrity constraints with a corresponding analysis (ALANLIGHT's key concepts) on expressiveness and on guarantees for programs performing complex calculations

In addition, we

- give an overview of typical problems resulting from the use of existing programming languages;
- illustrate the tension between expressiveness and guarantees for recursive calculations;
- evaluate expressiveness and guarantees for different kinds of recursive calculations with approaches from related work.

```
Products: collection {
  parts: collection {
    quantity: integer
  }
  manufacturingCost: integer
}
Orders: collection {
  OrderedProducts: collection {
    cost: integer
  }
  totalCost: integer
}
```



Products	manufacturingCost
CPU	800
Car	15000
GPU	500
Laptop	1920

Figure 1.1: Hierarchical AlAN/AlanLight model for a manufacturer and a corresponding generated user interface.

Contrary to relational databases which store tabular data, AlanLight targets hierarchical data. We aim for expressing programs in a way that closely resembles hierarchical data and data flow. Staying close to the structure and flow of data makes it easier to discuss and implement customer requirements, which simplifies the software development process. Furthermore, our language is target (model) driven and is closely related to object oriented programming languages, lowering the barrier for typical software developers.

Requirements for AlanLight. We developed AlanLight at M-industries. At M-industries we use Alan, a more feature-rich version of AlanLight without the analysis which we present in this thesis. We use Alan as the principal programming language for developing data-intensive software systems for manufacturing companies. For example, one of our customers is a world player in the aluminium extrusion industry.

M-industries' customers typically have a large connected infrastructure of machines and supporting systems. At M-industries, we develop software connecting these machines and supporting systems for optimizing production processes. We have successfully deployed several systems at manufacturing sites all over Europe which are actively being used as integral parts of manufacturing processes.

In context of M-industries, our main goal with AlanLight is to provide a solution for complex (recursive) calculations in Alan, while guaranteeing soundness, functional correctness, and bounded, predictable running times. Successfully deployed Alan-based systems show that Alan's basic concepts are effective tools for the development of data-intensive software systems. These concepts include hierarchical data modeling, and the data types that Alan supports. To ensure that concepts from AlanLight apply to Alan, we impose the following functional requirements on AlanLight:

- AlanLight should support (text-based, declarative) hierarchical data modeling.
- AlanLight should support the data types from Alan, insofar they affect AlanLight's core concepts: complex referential integrity constraints, and a corresponding analysis.

The reasons for ALAN being a hierarchical data modeling language are twofold. First, our customers typically have hierarchical data. Second, hierarchical data models enable easy generation of (tree-style) graphical user interfaces that our customers are accustomed to. Figure 1 illustrates this.

The left side of the figure shows a (simplified) hierarchical ALAN/ALANLIGHT model for a manufacturer. The right side of the figure depicts a user interface that we generated from the model. The model expresses that the manufacturer has a collection of `Products` and `Orders`. `Products` have (a collection of) `parts`, which have a `quantity`. `Orders` consist of a collection of `OrderedProducts` with a `cost`. The left side of the user interface shows the collections in a tree view for navigating through the data. The right side of the user interface shows the `Products` of the manufacturer, which we get by clicking on 'Products' in the navigation bar. The first column of the data grid shows that each product has a unique key (e.g. CPU); that is, a `collection` is a map holding unique key-value pairs.

The data model and the graphical user interface are closely related. Our customers (with typically no prior programming experience) can typically understand the model because it resembles how they think about, and work with their data. This significantly simplifies discussing customer requirements, and translating customer requirements into a concrete implementation.

Problems that ALAN targets, require calculations that (recursively) traverse complex relations. For example, relations between `Products` and their `parts` (which are typically also `Products`). ALAN neither requires other types of recursive calculations, nor iterative calculations. From this, we extracted the following requirement for the expressiveness of ALANLIGHT:

- ALANLIGHT should support recursive calculations that (recursively) traverse complex relations.

From our research perspective, ALANLIGHT adds value if the language is either more expressive or provides more guarantees than existing languages. Therefore, we specifically focus on recursive calculations for which related work either provides no support or does not give aforementioned guarantees. We address problems found in practice, as these can effectively demonstrate how ALANLIGHT succeeds in going beyond the capabilities of existing languages in the context of data-intensive software development. Addressing problems found in practice gives an informal definition of expressiveness; we consider a formal definition of expressiveness beyond the scope of this thesis. As part of our research, we aim to demonstrate the tension between expressiveness and guarantees for recursive calculations with regards to both ALANLIGHT's concepts as well approaches from related work.

Outline. Chapter 2 gives a problem statement, addressing problems in existing languages. Chapter 3 introduces ALANLIGHT. Chapter 4 presents a static semantics for ALANLIGHT. Chapter 5 shows the intended meaning of concepts in ALANLIGHT in the form of a formal operational semantics, and presents complexity bounds for the full class of ALANLIGHT programs. Chapter 6 evaluates the key concepts of ALANLIGHT, demonstrates the tension between expressiveness and guarantees for complex calculations, and discusses the expressiveness and guarantees in context of related work. Chapter 7 more generally discusses how ALANLIGHT compares to related work. Finally, Chapter 8 presents conclusions about our work, and suggests some ideas for future work.

Chapter 2

Problem statement

In this chapter, we discuss different problems with existing languages. We use these problems to define our notion of soundness, functional correctness, and bounded, predictable running times. The problems that we address, are organized accordingly. The remainder of this chapter discusses how related work addresses problems that we present, focusing mainly on problems involving recursive computations. We conclude this chapter with the main research question for this thesis.

Soundness. Programs in languages such as C [29], JAVASCRIPT [12], JAVA [17], SCALA [49], and SQL [15] can produce runtime errors. For example, SCALA allows unsafe casts, which yield an exception on the JVM (Java Virtual Machine):

```
val a = "some text";
val b = a.asInstanceOf[Int];

//-> java.lang.ClassCastException:
//    java.lang.String cannot be cast to java.lang.Integer
```

Similarly, C and JAVA allow division by zero, which gives runtime exceptions as well:

```
int temp_low = 0;
int temp_avg = ( temp_max + temp_low ) / temp_low;

//-> java.lang.ArithmeticException: / by zero
```

Furthermore, JAVASCRIPT supports applying operators to values of incompatible types, yielding special implementation-defined values such as `NaN` (Not-a-Number):

```
temp = 200 - "ten"; //Incompatible types

//-> NaN (JavaScript)
```

These errors result from an unsound type system. Our notion of soundness requires that ‘well-typed models (programs) don’t go wrong’ [41, 55, 72]. ALANLIGHT aims for a complete absence of runtime errors: valid ALANLIGHT models are safe. Furthermore, ALANLIGHT guarantees values of a predefined indivisible type. That is, a strong notion of type soundness which excludes `null` values and values of an option or maybe type, preventing corresponding exceptions, such as:

```
public void nullPointerException() {
  Product prod = null;
  System.out.println(prod.getName());
}

//-> java.lang.NullPointerException
```

Typical relational database modeling and query languages, including SQL, have similar problems. They do not require specific multiplicities for the input to operations applying to sets or bags of values. Instead, they produce `null` values or special implementation-defined values on empty sets or bags:

```
SELECT MAX(price)
FROM products
WHERE (SELECT COUNT(price)) = 0;

//-> null //always null, because of the WHERE-clause.
```

Functional correctness. A strong soundness result ensures the absence of runtime errors. However, soundness does not guarantee that programs are correct with respect to the specification defining the intention of a program or system. Soundness depends on an operational semantics; an interpreter, that may include hacks or workarounds for ensuring correctly typed output. For example, a program may be well-typed, but only because of hacks in the form of 'fallback values' that a corresponding operational semantics defines; it may produce 0 when an integer object value is undefined.

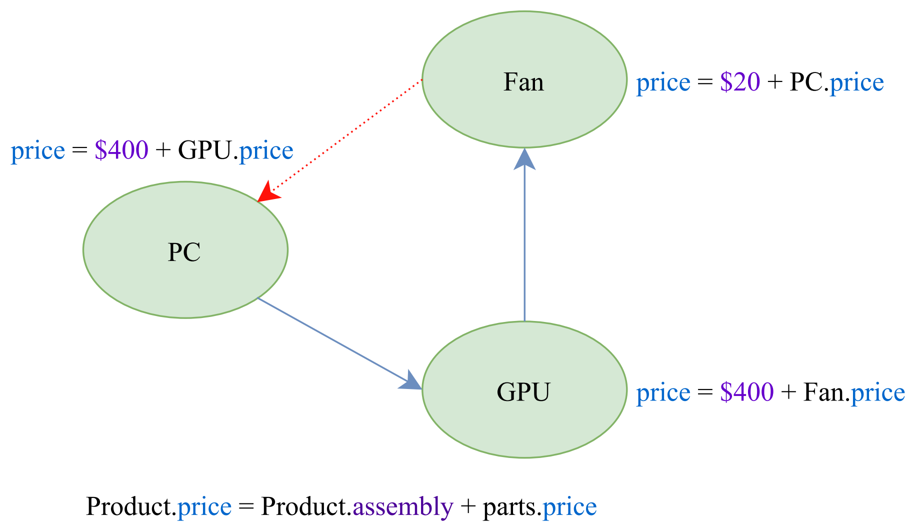
Functional correctness here means that for each input, a program produces expected output conform a (high-level) specification [6, 30, 34]. To explain this, consider the specification for the product price calculation from Figure 2.1. A program performing the price calculation is functionally correct – and its output always *correct* – if and only if the output *always* conforms to the specification of the price calculation. Our notion of functional correctness requires models to explicitly define every possible result of a computation in a deterministic manner, without falling back on values from an operational semantics (or the corresponding implementation of an interpreter). As we also require that programs terminate, we cannot guarantee this for the program from the figure (as product price values are infinite); the compiler should reject the program.

Languages such as SQL do not guarantee the above. For example, SQL queries can produce non-deterministic output [7]:

```
SELECT TOP(1) price
FROM products;

//-> 5 OR -500
```

The `TOP` clause limits the number of price values (rows) that the query produces. `TOP(1)` produces the `price` of a single product from the table (if the table holds data). As the order



Fan
price = \$20 + PC.price
price = \$400 + GPU.price
PC
GPU
price = \$400 + Fan.price
Product.price = Product.assembly + parts.price

Figure 2.1: Recursive product price calculation, where products consist of parts. The arrows indicate product→part relations. The red, dotted arrow indicates an incorrect relation which creates a cyclic dependency among products.

of the `products` is undefined, subsequent runs of the query can yield (different) price values from different products.

Queries or programs exhibiting such behaviour can produce expected output during testing, but can yield incorrect output in a production environment. Similar non-deterministic behaviour can occur when comparing pointers to memory locations in C or C++.

Another typical problem where a program does not produce functionally correct output is overflow (and also underflow):

```
int x = 987654321;
int y = x * x;

//-> y = -238269855
```

Instead of `y` holding the square of `x`, it holds a negative number value in JAVA and C. In SQL, we get a runtime error instead: `Data truncation`.

Bounded, predictable running times. Soundness and functional correctness ensure safe programs that give correct output; that is, assuming that programs produce any output at all. Programs in Turing-complete languages such as JAVA can have unpredictably long running times. Moreover, they may not terminate.

To illustrate the above, take the following two JAVA functions which can run infinitely:

```
public void recurse() {
  recurse();
}

public void loop() {
  while (1) { doSomething(); }
}
```

Programs calling functions such as these can stall production processes indefinitely. Thus, it is essential that we ensure that running times are bounded; that programs terminate.

However, when guaranteeing program termination, programs may still have long running times; they can delay production processes for unpredictably long periods of time. To illustrate this, consider the following part of a JAVA program:

```
while (x >= threshold) {
  x = decrease(x);
}
```

If the function `decrease` produces a value less than `x`, we know that the loop terminates. However, we cannot predict when `x` will drop below the `threshold`. Moreover, suppose that the running time for a specific value of `x` and `threshold` is 10 seconds. Then, even minor changes to the values can significantly impact the running time of the program, possibly turning 10 seconds into 10 years.

The problem is that (calculated, numerical) variable values bound the running time of the program. The size of the datasets (program input) of M-industries' customers typically changes slowly and in a predictable manner, while values are subject to frequent, unpredictable modifications. Therefore, if running times depend on values, they are unpredictable. For running times that depend on the size of customer data, we can give more accurate predictions. As the datasets can grow very large, it is also important that running times do not grow exponentially in the size of user data. Therefore, our notion of predictable running times requires polynomial running time bounds in the size of user data.

Towards a solution. Different approaches exist for ensuring bounded running times. A simple solution is to limit the expressiveness of a language: we can drop looping constructs and preclude recursion. However, complex data exchange and transformation processes typically require recursion. In fact, the main goal of our approach is to support complex recursive calculations.

A naive approach for ensuring bounded recursive calculations is a time limit. For example, PHP supports limiting execution times as follows:

```
set_time_limit(5);

$i = 0;
while (true) {
  $i++;
}

//-> $i = 1 or 2 or 3 or 4 ...
```

Similarly, SQL [61, 47] also lets developers bound the recursion depth of recursive queries:

```
SELECT SUM(quantity)
FROM recursive_sql_query
OPTION (MAXRECURSION 2);

//-> result after 2 iterations
```

However, such approaches either produce non-deterministic output or incomplete output (by only partially evaluating recursive computations).

A fixpoint computation (or fixed-point computation) is a more sophisticated approach that can ensure bounded running times. A fixpoint computation halts when no new information (facts, tuples, a value) is computed during an iteration of a fixpoint algorithm. For example, recursive DATALOG [18] uses a fixpoint algorithm that halts when it can no longer derive new tuples describing previously unknown relations among objects.

To further explain this, consider the product graph from Figure 2.1. Recursive DATALOG supports computing the transitive closure of the product graph (the set of all direct and indirect relations among products). From the graph, we can only derive a finite set of relations among products, at which point (the fixpoint) the fixpoint algorithm terminates.

Languages such as DATALOG^{FS} [37] suggest this approach for calculating values such as the product price from Figure 2.1. The approach ensures termination; the fixpoint algorithm produces finite price values because part prices only count once towards the final result. However, such price values are incorrect for the cyclic product graph from the figure. The values do not conform to the functional specification of the price calculation. Finite price values can give users and systems that receive them a false sense of correctness. It is important that we prevent that.

As mentioned for the product-part relations, a cyclic dependency should be prevented in the first place. In fact, many different kinds of relations that computations traverse recursively, require the absence of a cycle among them. ALANLIGHT is based on this idea. ALANLIGHT focuses on enabling complex recursive calculations – under aforementioned guarantees – using complex referential integrity constraints, which include acyclicity constraints for relations that computations traverse recursively.

The idea of introducing acyclicity constraints on relations is not new [21, 66, 71]. The need for acyclicity follows immediately from the fact that termination cannot be guaranteed for calculations which unboundedly traverse circular relations. Furthermore, acyclicity is a natural requirement for many relation types; for example, for *product-part* relations, *version* relations, and different kinds of *time bound* relations.

As shown above, data modeling languages and query languages that do not guarantee acyclic relation traversal for recursive calculations, require a different approach to ensuring termination. A naive approach is a time limit or a recursion depth limit [53]. More sophisticated approaches discussed in Abiteboul et al. [1] and Green et al. [18] limit expressiveness instead. For example, they support recursively calculating *minima* and *maxima*, but do not support recursive *summation*. (Note that *minimum* inside recursion yields a unique value (a fixpoint), whereas *sum* may infinitely increase in the presence of cycles.)

For more expressiveness, some of these approaches store results from earlier recursive calls. They use these results to prevent reevaluation of identical recursive calls [3, 64]. However, they do not guarantee functionally correct output. To our knowledge, no existing data modeling language guarantees our notion of soundness, functional correctness, and bounded, predictable running times, while providing similar expressive power for expressing recursive computations. The research question for this thesis is: *to what extent can we support complex recursive calculations, while guaranteeing soundness, functional correctness, and bounded, predictable running times?*

Chapter 3

AlanLight

This section introduces the key concepts of AlanLight: hierarchical data modeling, data flow modeling, derived values, and constraints on data. We discuss typical data modeling problems in the context of data-intensive software development and show how AlanLight addresses these problems. As running example we use part of an enterprise resource planning (ERP) system for a company that manufactures and distributes electronics. The manufacturer has customers, products, purchase orders and sales orders. Figure 3.1 shows a feature model (extended with crow's foot notation) for a subset of the manufacturer's data. Figure 3.2 presents the AlanLight version of this model.

3.1 Hierarchical data models

AlanLight models are hierarchical models specifying hierarchical data. Our solution for achieving soundness, functional correctness, and boundedness, uses this hierarchical aspect of AlanLight models.

An AlanLight model is a hierarchy of *nested types* with a single *root type*:

```
root { ... /* attributes */ ... }
```

The root type is a *complex type* that nests other (complex) types. Types are surrounded by curly braces, and their identification is a path which starts from the root type. We refer to an instance of a type as a *node*. Types contain *attributes*; their members. Attributes have a unique id and a built-in attribute type. AlanLight supports five different attribute types: *text*, *integer*, *natural*, *collection*, and *state group*. Figure 3.2 shows the use of these attribute types for the electronics manufacturer.

Text, *integer*, and *natural* are primitive attribute types. Text attributes hold an unbounded string value. Integers attributes hold an integer value, and natural attributes hold an integer value greater than zero. Primitive attribute types in AlanLight use the following notation:

```
last_name    : text
price        : integer /* Alan: eurocent */
release_date : natural /* Alan: date-time */
```

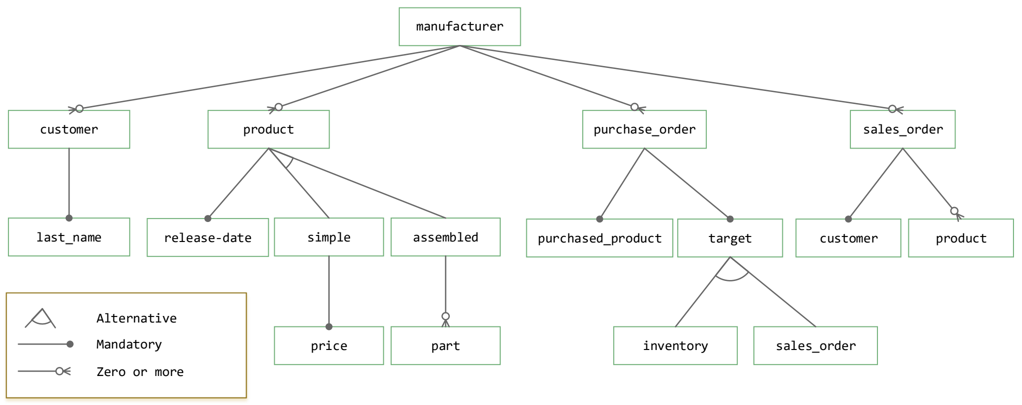


Figure 3.1: Feature model of an electronics manufacturers' data.

```
root {
  customers: collection {
    last_name: text
  }
  products: collection {
    release_date: natural
    product_type: stategroup (
      simple {
        price: integer
      }
      assembled {
        parts: collection {
          part_price: integer // derived from part key
        }
      }
    )
  }
  purchase_orders: collection {
    purchased_product: text
    target: stategroup ( // purchase for inventory or sales order fulfillment?
      inventory { }
      sales_order {
        sales_order: text
      }
    )
  }
  sales_orders: collection {
    customer: text
    products: collection { }
    order_price: integer // derived from products
  }
}
```

Figure 3.2: ALANLIGHT representation of an electronics manufacturers' data. The model corresponds to the feature model from Figure 3.1. The feature model omits attributes holding derived values.

For ensuring that number values have an explicit predefined accuracy, ALANLIGHT does not support number values with a fractional component. The super-language ALAN expresses the accuracy with unit type annotations, such as `date-time` or `eurocent`.

A *collection* attribute holds a map of key-value pairs. Keys are string values that have to be unique such that we can reference them unambiguously. Keys are implicitly defined for collection attributes; values are nodes of an inline defined type:

```
customers: collection { ... /* attributes */ ... }
```

A *state group* attribute holds a value indicating a *state*. States are alternatives to a property that a *state group* attribute indicates. For example, red, orange, or green for a 'color' attribute of a traffic-light. The type of the attribute value corresponds to one out of multiple predefined state types, such as `simple` or `assembled`:

```
product_type: stategroup (
  simple { ... /* attributes of this type */ ... }
  assembled { ... /* attributes of this type */ ... }
)
```

Languages such as ML[42] and HASKELL[68] refer to state group attributes as tagged unions (also: variant records or sum types). State group attributes enable modeling types with similar traits as a single type, whilst still being able to express their differences. Without a state group attribute such as `product_type`, we would need two different collections to distinguish between simple and assembled products. Because the types of the products from these collections would differ, we cannot treat them equally in (type safe) query expressions.

3.2 Derived values

Some attributes in ALANLIGHT models derive their values from elementary values or from other derived values. We refer to these attributes as *derived value attributes* (derived attributes or derivations in short). All attribute types that ALANLIGHT supports, have a set of operations for deriving values of their respective types. This means that ALANLIGHT supports deriving primitive values as well as complex relations and nodes.

The manufacturer from our running example needs derived values for gaining insight into his data. For example, he needs the `discounted_price` of his sales orders for gaining insight into their net sales value:

```
discounted_price: integer = switch (this.discount) as dc (
  | coupon = this.order_price - dc.coupon_value
  | none   = this.order_price )
```

The notation for derived value attributes in ALANLIGHT is mostly similar to the notation for elementary value attributes. The difference is an additional `=`-sign and expression for calculating the attribute value.

The previous example derives a primitive value: an integer. Suppose now that the manufacturer wants an overview of all his `simple products`. For this purpose, AlanLight supports deriving a collection with a complex type as follows:

```
simple_products: collection = this.products as p
  where(p.product_type|simple) = smpl
{
  price: integer = smpl.price
}
```

The navigation expression `this.products` selects a collection of nodes to serve as keys of items in the derived collection. The type of `simple_products` has derived value attributes. For deriving the values of these attributes, they use separate expressions determining their own values.

The keywords `this`, `parent`, and `key` are reserved words with a special meaning. The keyword `this` references the node that an expression is part of. The keyword `parent` references the parent node. Contrary to other programming languages, parent navigation in AlanLight is explicit because we have a nested hierarchy of types. Furthermore, it is important to the AlanLight runtime, which we further explain in subsequent chapters. The keyword `key` references the key of a collection entry. The `key` value is a string for basic elementary data, and a node if a collection is derived.

Figure 3.3 presents several other types of operations for deriving values of derived value attributes. This includes the `sum` aggregate operation and operators `*` and `-` for deriving integer values, and `empty` for deriving a state of a state group attribute.

3.3 Data flow representation in hierarchies

An AlanLight data model represents the flow of data, top-down in a textual format (left-to-right in Figure 3.1). Organizing types and their attributes in a top-down fashion (following the flow of data) is not a prerequisite. However, navigation expressions only use attributes and types that are fully defined before them:

```
net_sales_fw: integer = sum(this.sales_orders.discounted_price)
// Error: sales_orders references later defined derived attribute
sales_orders: collection {
  discounted_price: integer = ...
}
net_sales_bw: integer = sum(this.sales_orders.discounted_price)  // OK
```

Similar behaviour is found in languages such as C that require declaration (of e.g. functions or structs) before use.

Because navigation expressions require definition before use, it is important that types and attributes are organized in the way we need them for derivations. This leads to the three different layers of data that Figure 3.4 presents. The first layer, imported data, does not concern derivations. We combine the second layer, application data, with imported data to construct useful derivations for application users. From a mix of imported data and application data, we derive the third layer: views for external systems.

```
root {
  ...
  products: collection {
    description: text
    product_type: stategroup (
      simple { price: integer }
      assembled {
        parts: collection {
          amount: natural
          part_price: integer // derived from part (a product)
          price: integer = this.part_price * this.amount
        }
      }
    )
    has_parts: stategroup = empty(this.parts) as nonemptyparts
      | true  = no
      | false = yes (nonemptyparts)
    (
      yes (nonemptyparts) {
        max_part_price: integer = max(nonemptyparts.price)
      }
      no { }
    )
    product_price: integer = switch (this.product_type) as pt (
      | simple    = pt.price
      | assembled = sum(pt.parts.price) )
  }

  incorrectly_configured_products: collection = this.products as p
    where(p.product_type|assembled)
    where(p.has_parts|no)
  {
    description: integer = this>key.description
  }

  simple_products: collection = this.products as p
    where(p.product_type|simple) = smpl
  {
    price: integer = smpl.price
  }
  ...
  sales_orders: collection {
    products: collection {
      product_price: integer // derived from product
    }
    order_price: integer = sum(this.products.product_price)

    discount: stategroup (
      coupon { coupon_value: integer }
      none { }
    )
    discounted_price: integer = switch (this.discount) as dc (
      | coupon = this.order_price - dc.coupon_value
      | none   = this.order_price )
  }
  net_sales: integer = sum(this.sales_orders.discounted_price)
}
```

Figure 3.3: ALANLIGHT model of the electronics manufacturer extended with expressions for deriving attribute values.

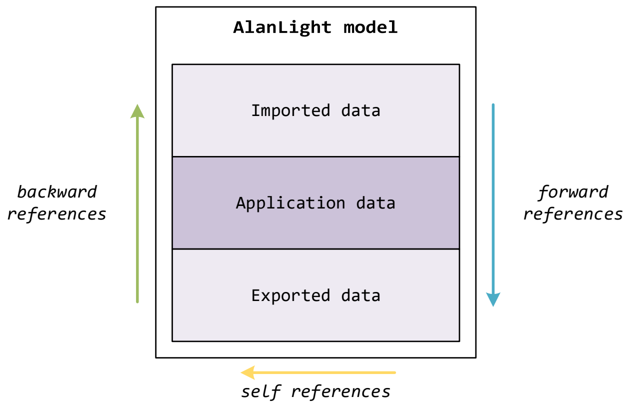


Figure 3.4: Data flow representation in ALANLIGHT models, and navigation directions for navigation expressions that follow this data flow. Backward reference expressions are the default, and follow the flow of data that a model expresses. We extend these with forward and self reference expressions to increase expressiveness.

3.4 Reference constraints

Sometimes, derived value calculations require following a reference. For example, an order price calculation requires following a reference from a `purchase_orders` item to the `purchased_product` that has a price attribute. For enabling such calculations, ALANLIGHT supports reference constraints. ALANLIGHT models specify reference constraints as follows:

```
products: collection { ...
  product_price: integer ...
}
purchase_orders: collection {
  // we need the purchased product reference...
  purchased_product: ~ ref(this.parent.products)
  amount: integer
  // for calculating this:
  total_price: integer = this>purchased_product.product_price * this.amount
}
```

The `purchased_product` attribute is a special attribute holding values which reference `products`. The keyword `~` indicates a constraint or fact about data. The keyword `ref` means reference constraint, and the navigation path defines the reference: a reference to a product from a specific `products` collection. Finding the correct `products` collection at runtime, requires evaluating the navigation path. The reference constraint ensures that the `total_price` calculation does not yield undefined behaviour.

Keys of `collection` items can also reference items in another collection:

```
purchased_products: collection ~ ref(this.parent.products) { ... }
```

For such collections, the navigation step `key` is for navigating to the referenced node. When following a reference, attribute values of the referenced node are available for use in constraints and calculations. ALANLIGHT precludes navigating to parent nodes after following a reference. This prevents verbose alternatives to certain navigation expressions:

```
my_product   : ~ ref(/*this:   */ this.products)
other_product: ~ ref(/*equals: */ this>my_product.parent.products)
```

Bidirectional references. By default, references in ALANLIGHT are *unidirectional*. Sometimes we need *bidirectional* references for deriving values such as a reference count. To this end, ALANLIGHT supports storing inverse references on referenced nodes:

```
products: collection {
  orders: root.purchase_orders = inv-refs(>purchased_product)
}
purchase_orders ...
```

An `inv-refs` expression specifies the references for which an inverse reference attribute holds the inverse: `orders` holds the inverse of `purchased_product` references.

Forward references. The expressions above use attributes that are undefined at the expression. ALANLIGHT supports this for elementary bidirectional references as well as for elementary unidirectional references. This enables expressing calculations for nodes that require values from equally typed sibling nodes:

```
base_currency: ~ ref(this.currencies) // forward reference
currencies: collection {
  exchange_rate_pound: integer
  exchange_rate_base: integer =
    this.exchange_rate_pound / this.parent.base_currency.exchange_rate_pound
}
```

Every currency has an exchange rate in pounds. For calculating exchange rates in terms of a configurable `base_currency`, `exchange_rate_base` calculations use the `base_currency` reference for retrieving its exchange rate in pounds.

Because derivation expressions only use earlier defined attributes, the `base_currency` is defined before `currencies`. We refer to references pointing to nodes of later defined types as *forward references*. In contrast, *backward references* point to nodes of earlier defined types. Equivalent to backward references, forward references can use earlier defined forward and backward references:

```
fwref1    : ~ ref(this.col1)
fwref2    : ~ ref(this>fwref1.col2)
fwref3    : ~ ref(this>fwref1>ref1.col3)
  // Error: cannot use later defined reference
col1 : collection {
  col2 : collection {
    col3 : collection { ... }
  }
  ref1 : ~ ref(this.parent.col2)
}
bwref3 : ~ ref(this>fwref1>ref1.col3)
  // OK: same expression as fwref3, but fine here
```

Derived value calculations use forward references for deriving attribute values. After following a forward reference, calculations can use elementary values of the referenced node. They can also use derived values of the referenced node; that is, if the type of the referenced node is defined before the derived value attribute expressing the calculation.

Sometimes, calculations also require values of derived value attributes defined after the expression. For example, take the following per-product revenue calculation for the manufacturer from our running example:

```
products: collection {
  product_price: integer ...
  orders        : root.purchase_orders = inv-refs(>purchased_product)
  orders_count  : integer = count(this<orders)
  revenue       : integer = sum(this<orders.total_price)
    // Error: cannot use derived attribute of later defined orders
}
purchase_orders: collection {
  purchased_product: ~ ref(this.parent.products)
  total_price: integer = .../*some expression for deriving the value*/...
}
```

The revenue calculation requires a later defined derived total_price value. In ALANLIGHT, we typically solve this by placing the calculation (for total_price) on the referencing node:

```
products: collection {
  product_price: integer ...
  orders        : root.purchase_orders = inv-refs(>purchased_product)
  orders_count  : integer = count(this<orders)
  revenue       : integer = this.orders_count * this.product_price
}
```

Sometimes, a calculation cannot be moved to the referencing node. For this reason, ALAN also supports using later defined derived values after a following a reference. But, we omit this for ALANLIGHT, as it does not contribute to demonstrating ALANLIGHT's core concepts (while it significantly increases the complexity of the language's semantics). Note that this does not mean that ALANLIGHT cannot solve problems requiring later defined derived values. Instead of using the values directly, calculations for these problems require introducing an additional derived collection:

```
products: collection {
  orders: root.purchase_orders = inv-refs(>purchased_product)
}
purchase_orders: collection {
  total_price: integer = ...
  purchased_product: ~ ref(this.parent.products)
}
per_product_revenue: collection = this.products {
  revenue: integer    = sum(this>key<orders.total_price)
}
```

As the `per_product_revenue` collection is defined after the `purchase_orders` collection, `revenue` calculations can use `total_price` values of `purchase_orders`.

3.5 Graph constraints

Section 3.1 presented a model for an electronics manufacturer (Figure 3.2). The manufacturer has two different kinds of products: simple products and assembled products. Assembled products consist of `parts` that are either `simple` or `assembled` products themselves. Keys of `parts` point to items in a context (ancestor) collection; they are neither backward nor forward references. For expressing this, ALANLIGHT supports *self* references (Figure 3.4). *Self* references point to items in context (ancestor) collections:

```
products: collection {
  parts: collection ~ ref(this.parent.products) { ... } // self reference
}
```

Self reference constraints ensure referential integrity for `parts`, but introduce a new problem for derived value calculations: circular references. A product can reference itself via `parts`, either directly or indirectly. If it does, calculating the functionally correct price yields an infinite loop.

For preventing circular references, – thus to ensure that calculations finite and their results are unique, functionally correct, and of a predefined indivisible type, – we express a graph constraint on a collection's keys:

```
products: collection
  ~ parts_graph = acyclic-graph // graph constraint
{
  parts: collection
    ~ ref(this.parent.products[parts_graph]) // graph participation
  {
    amount: natural
    price: integer = switch (this>key.product_type) as refpt (
      | simple    = refpt.price * this.amount
      | assembled = sum(refpt.parts.price) * this.amount
    )
  }
}
```

Graph constraints ensure that entities (collection items) form a uniquely identified acyclic graph, such as a `parts_graph`. A graph consists of *nodes*, connected via *edges*. Entities are the *nodes* of a graph. References determine the edges of the graph, but graph constraint expressions do not specify which references. Instead, reference constraint expressions define to which graphs references belong.

For deriving values, calculations use earlier defined self references. As we discussed before, this also applies to backward references and forward references. After following a self reference, earlier defined attribute values (with respect to the target attribute) are available for use:

```
products: collection ~ version_graph = acyclic-graph {
  description: text // Earlier defined attribute w.r.t. price
  has_new_version: stategroup (
    no { }
    yes {
      successor   : ~ ref(this.parent.parent.products[version_graph])
      description: text = this>successor.description // OK
      price       : integer = this>successor.product_price
        // Error: undefined attribute product_price
    }
  )
  product_price: integer = // Later defined derived attribute w.r.t. price
}
```

Recursive calculations. For recursively calculating values, expressions use *themselves* after a step for following a self reference:

```
products: collection ...
  latest_version: root.products = switch (this.has_new_version) as v (
    | no  = this
    | yes = v>successor>latest_version ) // uses the expression itself
```

The navigation step >`successor` follows a self reference to another product. After that step, the expression for calculating the `latest_version` points to itself.

Enabling expressions to use themselves after self reference steps, yields potentially unbounded calculations. For example, consider the `UpgradeKit` products from Figure 3.6. `UpgradeKits` circularly depend on each other. `UpgradeKit V1` implicitly references `V3` as a successor, and `V3` includes `V1` as a part (extending `V1` with a `GPU`). Without additional constraints on navigation, the following code compiles:

```
// Last known product price over all last known prices of parts:
last_known_product_price: integer = switch (this.has_new_version) as v (
  | yes = v>version.last_known_product_price
  | no  = switch (this.product_type) as pt (
    | simple    = pt.price
    | assembled = sum(pt.parts>key.last_known_product_price)
  )
)
```

For `UpgradeKit V1`, the `last_known_product_price` (`lnpp`) calculation yields infinite recursion. `UpgradeKit V1` requires calculating the `lnpp` for `V2`; `V2` depends on the `lnpp` from `V3`. But, the calculation for `V3` requires the original `lnpp` from `V1`. Guaranteeing bounded `lnpp` calculations requires an additional constraint: a constraint ensuring that `parts` key

```
root {
  customers: collection { last_name: text }
  products: collection
    ~ parts_graph   = acyclic-graph
    ~ version_graph = acyclic-graph
    {
      product_type: stategroup (
        simple { price: integer }
        assembled {
          parts: collection ~ ref(this.parent.parent.products[parts_graph]) {
            amount: natural
            price: integer = switch (this>key.product_type) as refpt (
              | simple    = refpt.price * this.amount
              | assembled = sum(refpt.parts.price) * this.amount
            )
          }
        }
      )
      has_new_version: stategroup (
        no { } //this is the latest version of the product
        yes { successor: ~ ref(this.parent.parent.products[version_graph]) }
      )
      latest_version: root.products = switch (this.has_new_version) as v (
        | no  = this
        | yes = v>successor>latest_version
      )

      // price of a product based on prices of its parts
      product_price: integer = switch (this.product_type) as pt (
        | simple    = pt.price
        | assembled = sum(pt.parts.price)
      )

      // maximum price of the product, starting at this version
      max_product_price: integer = switch (this.has_new_version) as v (
        | no  = this.product_price
        | yes = max(v>successor.max_product_price, this.product_price)
      )
    }
  sales_orders: collection ~ order_sequence = acyclic-graph {
    has_preceding_order: stategroup (
      yes { order: ~ ref(this.parent.parent.sales_orders[order_sequence]) }
      no { }
    )
    products: collection ~ ref(this.parent.products) { }
    order_price: integer = sum(this.products>key.price)
  }
  has_sales_orders: stategroup = empty(this.sales_orders) as nonemptyorders
    | true  = no
    | false = yes (nonemptyorders)
  (
    no { }
    yes (orders) { // non-empty orders collection for this state type
      max_price: integer = max(orders)
    }
  )
}
```

Figure 3.5: ALANLIGHT model for the electronics manufacturer expressing graph constraints, self reference constraints, and recursive calculations.

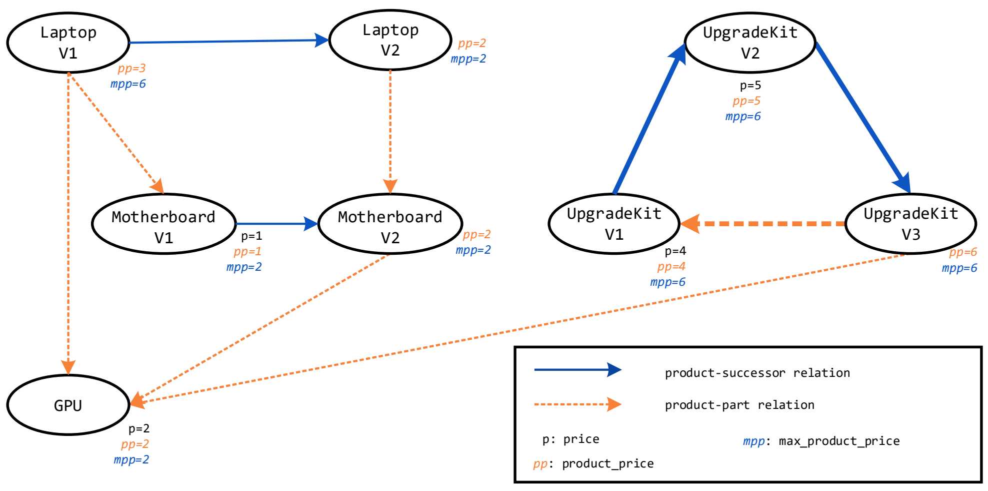


Figure 3.6: Products with version and part relations, and derived values that use these relations for calculating their values. The bold edges together form a cyclic graph. Figure 3.5 presents the expressions for calculating the `product_price` and the `max_product_price`.

references and `successor` references *together* form an acyclic graph. This prevents the circular dependency among the `UpgradeKits` from Figure 3.6, ensuring bounded `lnpp` calculations. The next paragraph discusses this in more detail.

Conditions for recursion. ALANLIGHT implements two conditions for ensuring that models contain all required constraints for polynomially bounded recursive calculations, and functionally correct calculation results of a predefined indivisible type. The key idea is that for each recursive calculation, at least one acyclic graph bounds the recursion.

The first condition is: *'subsequently followed self references that precede a recursive navigation step, partake in at least one identical graph'*. To explain this condition, consider the following `price` calculation for `parts` of `products`:

```
parts: collection ~ ref(this.parent.parent.products[parts_graph]) {
  price: integer = switch (this>key.product_type) as refpt (
    | simple    = refpt.price
    | assembled = sum(refpt.parts.price)
  )
}
```

Retrieving `price` values from `parts`, requires following `parts` key references to products. The `parts_graph` constraint ensures that parts do not reference themselves. Thus, `price` calculations are bounded; the `parts_graph` bounds the recursion. After following a `parts` key reference to a product, calculations can follow that products' `parts` key references. In fact, calculations can follow any self reference, as long as it partakes in the `parts_graph`:

```
part1: ~ ref(this.parent.parent.products[parts_graph,version_graph])
part2: ~ ref(this.parent.parent.products[parts_graph,fun_graph])
part3: ~ ref(this.parent.parent.products[parts_graph,great_graph])
sub_sub_part_price: integer = switch (this>part1.product_type) as refpt1 (
  | simple    = refpt1.price
  | assembled = switch (refpt1>part2.product_type) as refpt2 (
    | simple    = refpt2.price
    | assembled = switch (refpt2>part3.product_type) as refpt3 (
        | simple    = refpt3.price
      | assembled = refpt3.sub_sub_part_price
    )
  )
)
```

Again, the `parts_graph` bounds the recursion. It is essential that all followed self references partake in one identical graph; the `parts_graph`. This is because combinations of graphs can contain cycles; see also Figure 3.6.

Sometimes, expressions consist of multiple independent subexpressions; different possible branches in a calculation. To achieve ALANLIGHT's guarantees, the language implements a second condition for such expressions: *'all first followed self references that precede a recursive navigation step, partake in at least one identical graph'*. The 'first followed self reference' of a navigation path, is the first self reference step after a `this` step (for navigating to a context node). For explaining the second condition, consider the following example where two subexpressions for calculating a `preferred_product` use the attribute itself:

```
products: collection
  ~ g_versions = acyclic-graph
  ~ g_alts = acyclic-graph
{
  has_new_version: stategroup (
    no { } // latest version of the product
    yes { successor: ~ ref(this.parent.parent.products[g_versions]) } )
  has_alternative: stategroup (
    no { } // no alternative to this product
    yes { alternative: ~ ref(this.parent.parent.products[g_alts]) } )
  preferred_product: root.products = switch (this.has_new_version) as v (
    | no  = switch (this.has_alternative) as a (
      | no  = this
      | yes = a>alternative>preferred_product ) // alternative product S1
    | yes = v>successor>preferred_product )     // latest version      S2
}
```

The navigation paths of subexpressions `S1` and `S2` are different. Subexpression `S1` uses the self reference `successor`, whereas `S2` uses the self reference `alternative`. As the self references do not partake in least one identical graph, `preferred_product` calculations may run infinitely. For example, if an `UpgradeKit V1` points to `V3` as its `successor` and `V3` points to `V1` as its `alternative` (Figure 3.6), `preferred_product` calculations will unboundedly traverse the cyclic references among `UpgradeKits`. For precluding this, ALANLIGHT requires that either `successor` references partake in the `g_alts` graph, or `alternative` references partake in the `g_versions` graph.

Note that the two conditions for recursion do not require using equivalent graphs. Different subexpressions may traverse different graphs and contain recursion. For instance, suppose that for a cost price analysis, the manufacturer requires the absolute minimum cost of the parts of his assembled products. The absolute minimum cost is the minimum cost over all versions and alternatives of a product:

```
products: collection
  ~ g_parts = acyclic-graph
  ~ g_versions = acyclic-graph
  ~ g_alts = acyclic-graph
{
  has_new_version: stategroup (
    no  {}
    yes { successor: ~ ref(this.parent.parent.products[g_versions]) } )
  has_alternative: stategroup (
    no  {}
    yes { alternative: ~ ref(this.parent.parent.products[g_alts]) } )
  product_type: stategroup (
    simple { cost: integer }
    assembled {
      parts: collection
        ~ ref(this.parent.parent.products[g_parts,g_versions,g_alts])
      {
        amount: integer
        abs_min_cost: integer = min(
          switch (this>key.has_new_version) as nv (
            | yes = switch (nv>successor.product_type) as nvass (
              | simple    = nvass.cost
              | assembled = sum(nvass.parts as p mapto(p.amount * p.abs_min_cost)))
            | no  = switch (this>key.product_type) as ass (
              | simple    = ass.cost
              | assembled = sum(ass.parts as p mapto(p.amount * p.abs_min_cost))))
          ,switch (this>key.has_alternative) as alt (
            | yes = switch (alt>alternative.product_type) as altass (
              | simple    = altass.cost
              | assembled = sum(altass.parts as p mapto(p.amount * p.abs_min_cost)))
            | no  = switch (this>key.product_type) as ass (
              | simple    = ass.cost
              | assembled = sum(ass.parts as p mapto(p.amount * p.abs_min_cost)))))
      }
    }
  )
}
```

Expanding the variables from the navigation expressions gives three unique expressions:

```
this>key[g_parts,g_versions,g_alts]
  .has_new_version|yes>successor[g_versions]
  .product_type|assembled.parts.abs_min_cost
this>key[g_parts,g_versions,g_alts]
  .product_type|assembled.parts.abs_min_cost
this>key[g_parts,g_versions,g_alts]
  .has_alternative|yes>alternative[g_alts]
  .product_type|assembled.parts.abs_min_cost
```

The graph-sets between brackets hold graphs that immediately preceding references are part of. The first expression contains recursion on the product versions graph `g_versions`, while the third expression contains recursion on the alternatives graph `g_alts`. This is valid ALANLIGHT code, as it meets both conditions for recursion. Every subsequently (second) followed self reference partakes in an identical graph; and, the first followed self references partake in an identical graph.

Implicit graph participation: references. Sometimes, recursive calculations require following derived self references. For example, suppose that the manufacturer sometimes upgrades product parts to their latest versions. Determining if these upgrades are cost-effective requires calculating the cost price reduction:

```
products: collection
  ~ g_versions = acyclic-graph
  ~ g_parts = acyclic-graph
{
  has_new_version: stategroup (
    no { }
    yes {
      successor: ~ ref(this.parent.parent.products[g_versions])
      latest: root.products
        = switch (this>successor.has_new_version) //successor[g_versions]
          as nv (
          | no  = this>successor // successor[g_versions]
          | yes = nv>latest )    // nv[g_versions]
    }
  )
  product_type: stategroup (
    simple { cost: integer }
    assembled {
      parts: collection
        ~ ref(this.parent.parent.products[g_parts,g_versions])
      {
        latest_version: root.products
          = switch (this>key.has_new_version) as pnv (
          | no  = this>key      // [g_parts,g_versions]
          | yes = pnv>latest ) // [g_versions]
        latest_version_cost: integer
          = switch (this>latest_version.product_type) as pt (
          | simple    = pt.cost
          | assembled = sum(pt.parts>key.latest_version_cost) )

        current_cost: integer
          = switch (this>key.product_type) as pt (
          | simple    = pt.cost
          | assembled = sum(pt.parts>key.current_cost) )

        total_cost_reduction: integer
          = this.current_cost - this.latest_version_cost
      }
    }
  )
}
```

The attribute `total_cost_reduction` uses the `latest_version_cost`. This is where it gets interesting: latest_version_cost calculations are recursive; they follow derived latest_version references before recursing. That is, even though latest_version references do not explicitly partake in a graph. In addition, latest_version references use recursively calculated latest references themselves, which point to the latest version of a product.

For supporting latest_version_cost calculations while ensuring boundedness, the language determines the (implicitly defined) sets of graphs that reference attributes partake in. This set is the intersection of the graph-sets at the end of subexpressions. For the latest_version attribute, this is the set containing only g_versions. We call this set the *follow-set* of the latest_version attribute.

For explaining how ALANLIGHT determines a *follow-set*, consider the expression for calculating the latest product version. Expanding the variable nv gives the following set of subexpressions determining latest reference values (annotated with graphs):

```
this>successor[g_versions]
this>successor[g_versions].has_new_version|yes>latest
```

Both subexpressions implicitly define traversal of a g_versions graph. Therefore, a latest reference is an edge in the transitive closure of g_versions. That is, the latest reference is also (implicitly) part of the graph.

A latest_version reference uses a latest reference and parts key reference for deriving its value. Expanding the variable pnv gives the following set of subexpressions determining the result of the expression:

```
this>key[g_parts,g_versions]
this>key[g_parts,g_versions].has_new_version|yes>latest[g_versions]
```

The key references partake in both the g_parts graph as well as the g_versions graph. But, latest references only (implicitly) partake in the g_versions graph. Therefore, it is only guaranteed that the latest_version reference partakes in the g_versions graph. This means that the follow-set of the latest_version attribute is the set containing only g_versions. Chapter 4 gives a formal description of follow-set (δ) calculations.

Implicit graph participation: parameters. Above, we discussed follow-sets for derived references. These hold graphs that derived references are implicitly part of. State types optionally have parameters; we also need to calculate follow-sets for these parameters. For instance, suppose that the electronics manufacturer requires insight into the availability of a replacement for each of his products. Furthermore, suppose that he wants to know the minimum replacement cost, given that a replacement is available.

The minimum replacement cost is the minimum cost over a replacement and replacements of the replacement itself:

```
products: collection
  ~ g_versions = acyclic-graph
  ~ g_alts     = acyclic-graph
{
  cost: integer
  has_new_version: stategroup (
    no { }
    yes {
      successor: ~ ref(this.parent.parent.products[g_versions])
    }
  )
  has_alternative: stategroup (
    no { }
    yes {
      alternative: ~ ref(this.parent.parent.products[g_alts,g_versions])
    }
  )
  replacement_available: stategroup
    = switch (this.has_new_version) as nv (
    | no  = switch (this.has_alternative) as alt (
      | no  = no
      | yes = yes (replacement = alt>alternative) )
    | yes = yes(replacement = nv>successor)
  ) (
    yes (replacement /*: root.products*/) {
      min_replacement_cost: integer
        = switch (replacement.replacement_available) as rra (
        | yes = min(replacement.cost, rra.min_replacement_cost)
        | no  = replacement.cost )
    }
    no { }
  )
}
```

The state type yes is a parametrized state type with one parameter: the `replacement`. The possible values for `replacement` follow from the expression for calculating if a replacement is available. A `replacement` is either a newer version of a product or an alternative to a product.

A `min_replacement_cost` calculation requires following a `replacement` parameter value, and recursively using its `min_replacement_cost` value. To ensure correct output, and efficient on-demand calculation, ALANLIGHT requires that following the `replacement` parameter involves acyclic graph traversal. We ensure this in the same way as for references: we calculate a follow-set for the `replacement` parameter. For calculating the follow-set, ALANLIGHT essentially expands the expressions that determine the possible values for the parameter `replacement`:

```
this.has_new_version|yes>successor[g_versions]
this.has_alternative|yes>alternative[g_parts,g_versions]
```

The follow-set of the `replacement` parameter is the set containing (only) `g_versions`. Thus, the graph `g_versions` bounds recursive `min_replacement_cost` calculations.

Expression significance. Some recursive calculations require ignoring specific subexpressions when calculating follow-sets. Above, our follow-set calculation strategy already hinted towards this. In follow-set calculations, we only use subexpressions that yield the actual values of derived references (including referencing parameters).

For showing that ignoring specific subexpressions is useful, suppose that the manufacturer requires an overview of all parts per product and the `total_cost` per part:

```
products: collection ~ g_parts = acyclic-graph {
  product_type: stategroup (
    simple { cost: integer }
    assembled {
      parts: collection ~ ref(this.parent.parent.products[g_parts]) {
        amount: integer
      }
      all_parts: collection
        = this.parent.parent.products
          in(this.parts>key) = direct
          in(this.parts>key.product_type|assembled.all_parts>key) = indirect
      {
        total_amount: integer = sum(direct.amount) + sum(indirect.amount)
        total_cost: integer = this.total_amount *
          switch (this>key.product_type) as pt (
          | simple = pt.cost
          | assembled = sum(pt.all_parts.total_cost)
        )
      }
    }
  )
}
```

The `in` clauses in the figure aggregate references to `products`. For each unique referenced node, the derived collection `all_parts` holds an item. At runtime, ALANLIGHT groups referencing nodes by corresponding referenced nodes, and passes corresponding groups of referencing nodes as variable values to the derived collection items.

The expression for the `all_parts` collection selects all parts and subparts of a product, recursively. Thus, the `all_parts` collection of a product is the transitive closure of a rooted subgraph of `g_parts`; the product itself being the root of the subgraph. Because of this property, calculations using key references of `all_parts` items, never arrive back at the product they start at. This ensures boundedness for `total_cost` calculations.

For enabling such calculations, ALANLIGHT selectively ignores subexpressions when calculating follow-sets. For example, when collection key expressions contain `in` clauses, ALANLIGHT ignores the navigation expression selecting the source collection (after the `=`-sign). Furthermore, ALANLIGHT ignores the part between braces for `switch (...)` expressions, as it does not yield the actual value of an attribute.

Context navigation. Sometimes, a navigation path containing a single `this` step – optionally succeeded by `parent` steps – determines the value of a derived reference. Note that we cannot follow such references and immediately proceed with a recursive navigation step. Doing so can yield an infinite loop. But, suppose that the manufacturer requires an overview of all latest alternatives to the latest versions of his products:

```
products: collection ~ g_alts = acyclic-graph {
  latest_version: root.products = switch (...) as v (
    | no  = this // a "this" step can determine the latest_version
    | yes = ... )
  has_alternative: stategroup (
    no { }
    yes {
      alternative: ~ ref(this.parent.parent.products[g_alts])
      latest_alternatives: collection = this.parent.parent.products
        in(this.parent>latest_version.has_alternative|yes>alternative)
        in(this.parent>latest_version.has_alternative|yes>alternative
           .has_alternative|yes.latest_alternatives>key) //recursion
      { ... }
    }
  )
}
```

The second `in` clause expresses recursion *after* a step for following the `latest_version` reference. This is safe because the navigation expression also contains a step for following an `alternative` reference, before the recursive step. The `alternative` reference partakes in the `g_alts` graph; we traverse this graph when executing the expression.

For supporting this while guaranteeing bounded calculations, follow-sets store additional information. Follow-sets for references pointing to `this` or a `parent` node store the fact that the expression does not traverse a graph. Follow-sets containing such facts require following an additional reference (partaking in a graph) before a recursive step. In the above example, the `alternative` reference fulfills this requirement.

Nested graphs. So far, we have presented different forms of recursion over a single `collection` with a single set of graphs. But, because of the hierarchical nature of ALAN-LIGHT models, graphs can also be nested. Sometimes, calculations require traversing graphs on different hierarchical levels.

To illustrate this, suppose that the manufacturer is expanding his business. As Figure 3.7 expresses, the manufacturer now has multiple semi-independent manufacturing sites. Each of these sites produces specific (parts of) products. For instance, one site produces cooling fans, whereas an other site produces graphics cards. The sites depend on each other: the site that produces graphics cards needs cooling fans for these cards. Because of this, calculating values such as the cost of an assembled product requires following references to `products` from other manufacturing sites.

All `extprod` references implicitly partake in the `g_ext_parts` graph; the expression for `extprod` uses the reference attribute `site`. Figure 3.8 presents an instance of the model, showing relations among products, parts, and sites. Calculating the `total_cost` for the product `Desktop` requires following the references to the `GraphicsCard` and `CPUFan`. Both references partake in the `g_ext_parts` graph. This prevents circular dependencies among products of different manufacturing sites. After following a reference to another site, there is no need for restricting recursion on products of this site. We can first calculate all `total_cost` values for the `Cooling` site, then for the `Graphics` site, and finally for the `Computers` site. In general, after following a reference in an ancestor graph to an ancestor node, calculations use its child nodes without restriction.

```
sites: collection ~ g_ext_parts = acyclic-graph {
  products: collection ~ g_parts = acyclic-graph {
    product_type: stategroup (
      simple { cost: integer }
      assembled {
        parts: collection {
         amount: integer
         supplier: stategroup (
          self { prod: ~ ref(this.parent.parent.parent.parent.products[g_parts]) }
          other_site {
            site: ~ ref(this.parent.parent.parent.parent.parent.sites[g_ext_parts])
            extprod: ~ ref(this>site.products)
            transport_cost: integer
          }
         )
         total_cost: integer = this.amount * switch (this.supplier) as s (
          | self = switch (s>prod.product_type) as pt (
            | simple    = pt.cost
            | assembled = sum(pt.parts.total_cost) )
          | other_site = s.transport_cost + switch(s>extprod.product_type) as pt(
            | simple    = pt.cost
            | assembled = sum(pt.parts.total_cost)))
        }
      }
    )
  }
}
```

Figure 3.7: ALANLIGHT model for the electronics manufacturer, expressing multiple semi-independent manufacturing sites depending on each others parts.

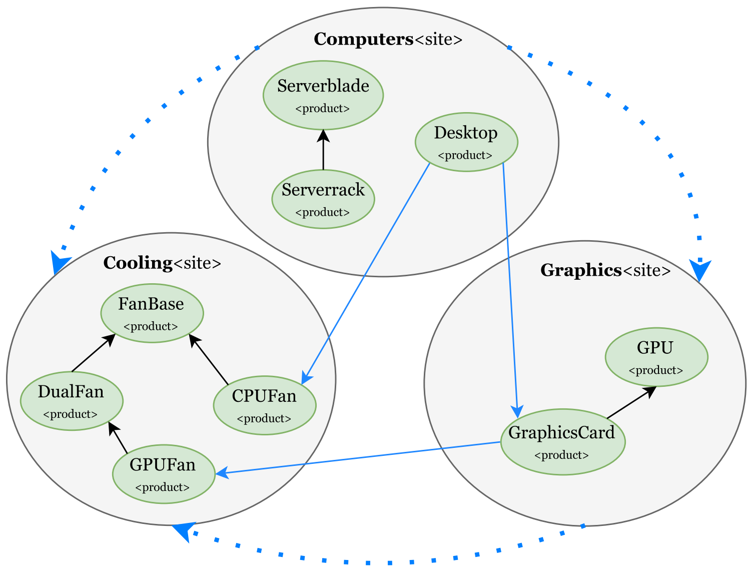


Figure 3.8: An instance of the model from Figure 3.7, showing multiple manufacturing sites with their own collections of products and relations among the products. Solid lines represent product-part relations. Blue dotted lines represent the transitive relations among sites, which follow from product-part relations.

Supporting recursion on hierarchically organized graphs in a single expression requires extending the presented conditions for recursive calculations. Namely, expanding the recursive subexpressions for calculating the `total_cost`, we get:

```
this.supplier|self>prod[g_parts]
  .product_type|assembled.bill_of_materials.total_cost
this.supplier|other_site>prod[g_ext_parts]
  .product_type|assembled.bill_of_materials.total_cost
```

The graph-sets for the subexpressions are different; the expression does not meet the second condition. In the first subexpression, the `g_parts` graph bounds recursion; in the second subexpression, the external parts graph `g_ext_parts` bounds recursion. ALANLIGHT supports this, while guaranteeing type soundness, functional correctness, and boundedness.

To achieve this, ALANLIGHT applies the second condition to all first followed self references *per level in the hierarchy* instead. For the `prod` reference from an `other_site`, the collection of sites determines its level in the hierarchy. This is its most significant (highest up) dependency in the model hierarchy. We reformulate the second condition as follows: *'all first followed self references, per level in the hierarchy, that precede a recursive navigation step, partake in at least one identical graph'*. That is, while ignoring less significant self references after following more significant ones.

Other calculations on hierarchically organized graphs require extending the first condition for recursive calculations as well. For instance, suppose that the manufacturer uses different sites for manufacturing different versions of a product. For each product he wants to see the latest version; Figure 3.9 expresses the calculation. The sites have their own self-manufactured latest version of a product. The latest version of a product is usually the self-manufactured latest version: `own_latest`. But, if another site manufactures a newer version of a sites' `own_latest` version, then the latest version of the other sites' product is the actual latest version of the product.
The two recursive subexpressions for calculating the `latest_version` expand to:

```
this.other_site_has_new_version|yes>successor[g_ext_versions]
  >latest_version /* L1 */
this.has_new_version|yes>latest[g_versions]
  .other_site_has_new_version|yes>successor[g_ext_versions]
  >latest_version /* L2 */
```

The first condition requires that the subsequently followed self references partake in the same graph. This is not the case for subexpression `L2`. However, ALANLIGHT supports this when self references traverse graphs on different hierarchical levels. We reformulate the first condition, as follows: 'subsequently followed self references *from the same hierarchical level* that precede a recursive navigation step, partake in at least one identical graph'. That is, with the additional property that we ignore less significant self references after following more significant ones. Thus, after following a self reference in an ancestor graph, calculations can freely use any self reference to a child node. We deduced the validity of this property when discussing Figure 3.8.

From the examples and extensions to the two conditions for recursive calculations follows that ALANLIGHT supports different kinds of recursion on hierarchies of graphs. These

```
sites: collection ~ g_ext_versions = acyclic-graph {
  products: collection ~ g_versions = acyclic-graph {
    has_new_version: stategroup (
      no {}
      yes {
        successor: ~ ref(this.parent.parent.products[g_versions])
        own_latest: = switch (this>successor.has_new_version) as nv (
          | no  = this>successor
          | yes = nv>own_latest )
      } )
    other_site_has_new_version: stategroup (
      no {}
      yes {
        site: ~ ref(this.parent.parent.parent.sites[g_ext_versions])
        successor: ~ ref(this>site.products)
      } )
    latest_version: root.sites.products = switch (this.has_new_version) as nv (
      | no  = switch (this.other_site_has_new_version) as osnv (
        | no  = this
        | yes = osnv>successor>latest_version ) /* L1 */
      | yes = switch (nv>own_latest.other_site_has_new_version) as osnv (
        | no  = nv>own_latest
        | yes = osnv>successor>latest_version )) /* L2 */
  }
}
```

Figure 3.9: ALANLIGHT model for the electronics manufacturer, expressing a `latest_version` calculation for a `product`. The latest version of a product for a manufacturing site is either the self-manufactured `own_latest` version of a product, or the `latest_version` from another site.

put different requirements on the sets of graphs for self references. The following recursion patterns summarize ALANLIGHT's support for recursion on hierarchies:

```
/* following multiple references succeeded by recursion */
best_product : root.sites.products =
  /* P1:  more significant reference succeeded by less significant reference: */
  ..>site(sites[g_sites])>prod(products[])>best_product..
  /* P2:  less significant reference succeeded by more significant reference: */
  ..>prod(products[])>site(sites[sites_g])>best_product..

  /* multiple recursive navigation steps */
  /* P3:  recursion after more significant reference
     succeeded by recursion after less significant reference: */
  ..>site(sites[g_sites])>best_product>prod(products[])>best_product..

  /* P4:  recursion after less significant reference,
     succeeded by recursion after more significant reference: */
  ..>prod(products[g_parts])>best_product>site(sites[g_sites])>best_product..
```

The patterns show expanded versions of expressions that ALANLIGHT supports. The expressions derive `best_product` references. The sea-green annotation includes the collection attribute name together with the graphs of the references.

Above, we presented two expanded subexpressions for deriving `latest_version` references. Subexpression `L2` matches pattern `P2`. `L2` defines following two self references on different hierarchical levels: `latest` and `successor`. Before recursion, the subexpression defines following the – less significant – `latest` reference, succeeded by the – more significant – `successor` reference.

Patterns `P1` and `P2` express that the most significant self references preceding a recursive step, require graph participation. For less significant self references (`prod`), AlanLight does *not* require graph participation. This summarizes the additional property for the extended first condition.

Patterns `P3` and `P4` demonstrate support for successive recursive steps in a navigation expression. `P3` expresses calculating the `best_product` by first following a self reference to another site. For example, a site that produces a newer product version. Subsequently, the calculation uses the `best_product` according to that site, and follows its reference to another product `prod`. For example, a product alternative. From this product, it selects the `best_product`.

`P4` follows the `prod` reference first, before the `site` reference. `P4` expresses recursion on a `prod` reference, followed by recursion on a more significant `site` reference. In `P4`, both the most significant `site` reference *as well as* the less significant `prod` reference require graph participation. This requirement ensures a bounded computation; without the requirement, the first recursive step may run infinitely.

Supporting recursion on hierarchies of graphs affects follow-set calculations. The previous paragraph discussed these follow-sets: sets of graphs that self references partake in, implicitly. The previous paragraph considered graphs on a single hierarchical level; from one specific collection. Therefore, follow-sets only included graphs from one collection.

For self references that traverse graphs on multiple hierarchical levels, follow-sets hold zero or more graph-sets. Each graph-set in a follow-set corresponds to a different hierarchical level. Below we explain how AlanLight calculates the follow-sets. We distinguish two cases: separate expressions (consisting of a single navigation path) and composite expressions (consisting of multiple subexpressions).

The follow-set for separate expressions contains the most significant traversed graphs (from the most significant followed self references). For example, for the expression

```
this.has_new_version|yes>latest[g_versions]
  .other_site_has_new_version|yes>successor[g_ext_versions]
  >latest_version /* L2 */
```

the follow-set contains only the graph `g_ext_versions`. If a follow-set contains less significant graphs, we ignore these when following references traversing more significant graphs. For instance, if expressions start with pattern `P1` or `P2` from the example above, we ignore the set of graphs from the `products` collection. We can ignore this graph-set because the expression traverses a more significant graph, from the `sites` collection.

For composite expressions, we combine the follow-sets from their subexpressions. For example, for two subexpressions

```
this>product_ref(products[parts_graph])
this>external_product_ref(sites[external_parts_graph])
```

the follow-set contains both sets of graphs:

```
{ products: [parts_graph], sites: [external_parts_graph] }
```

Generalizing conditions for recursion. Sometimes, calculations require following backward references and forward references before recursive steps. For supporting this, AlanLight stores follow-sets for these references as well. This is necessary for ensuring termination, as the references may (implicitly) traverse ancestor graphs that are also ancestor graphs with respect to the recursive expression.

Thus, the two conditions for recursion apply to references in general, rather then being specific to self references. The final two conditions for recursive calculations are:

1. Subsequently followed references, from the same hierarchical level, that precede a recursive navigation step, partake in at least one identical graph.
2. All first followed references of subexpressions, per level in the hierarchy, that precede a recursive navigation step, partake in at least one identical graph.

With the following property applying to both conditions: when following references that (implicitly) navigate via ancestor collections, we ignore less significant collections and their graphs. This means that we ignore follow-set elements (graph-sets) for less significant collections, instead of ignoring less significant references themselves.

Follow-sets form the basis for AlanLight's static semantics (Chapter 4). AlanLight's static semantics uses follow-sets for determining both implicit graph participation of derived references, as well as for checking recursive navigation steps.

3.6 Syntax

AlanLight's grammar follows directly from the concepts and examples we discussed in this chapter. Figure 3.10 presents the grammar in EBNF notation, using a style that borrows from regular expression syntax.

The names a, s, v, and g are attribute, state type, expression result, and graph names, respectively. For $NavMem$ steps, a is either a `text`, `integer`, `natural`, `stategroup`, or `collection` attribute. For $NavRef$ steps, a is a reference attribute, and for $NavInvRef$ it is an inverse reference attribute.

Variable assignment with $IntlVarAss$ is optional. But, before checking the static semantics of an AlanLight program, we inject a fresh name when variable assignment is omitted. For AST nodes $ExtlVarAss$ we do this as well. The next chapter presents AlanLight's static semantics.

```
Root          ::= root Type
Type          ::= { (a : Attr)* }
Attr          ::= text (= TextExp)?
                | integer (= NumExp)?
                | natural (= NumExp)?
                | collection Key? GraphConstr* Type
                | stategroup (= SgExp)? State+
                | TypePath? (RefConstr | RefExp)
                | TypePath * = inv-refs ( NavRef )
State         ::= s (( v (, v)* ))? Type
Key           ::= RefConstr | ColExp

RefConstr     ::= ~ ref ( NavThis NavMem [ g (, g)* ] )
                | ~ ref ( NavExp )
GraphConstr   ::= ~ g = acyclic-graph

TextExp       ::= NavExp
                | switch ( NavExp ) ( IntlVarAss ( (| s = TextExp)* )
NumExp        ::= NavExp
                | ( NumExp )
                | NumExp NumBinOp NumExp
                | NumBinFn ( NumExp , NumExp )
                | NumAggFn ( NavExp (IntlVarAss mapto ( NumExp ))? )
                | switch ( NavExp ) IntlVarAss ( (| s = NumExp)* )
NumBinOp      ::= - | + | / | * | %
NumBinFn      ::= min | max | diff
NumAggFn      ::= sum | count | min | max | avg
ColExp        ::= = NavExp IntlVarAss
                  ((where ( NavExp ) ExtlVarAss)*
                  | (in ( NavExp (NavRef | NavInvRef) ) ExtlVarAss)*)
SgExp         ::= s (( NavExp (, NavExp)* ))?
                | empty ( NavExp ) IntlVarAss | true = SgExp | false = SgExp
                | switch ( NavExp ) IntlVarAss ( (| s = SgExp)* )
RefExp        ::= NavExp
                | switch ( NavExp ) IntlVarAss ( (| s = RefExp)* )

IntlVarAss    ::= (as v)?
ExtlVarAss    ::= (= v)?

TypePath      ::= root | TypePath NavMem | TypePath NavState
NavExp        ::= NavStart | NavExp (NavMem | NavRef | NavInvRef | NavState)
NavStart      ::= NavThis | NavVar
NavThis       ::= this | NavThis . parent
NavVar        ::= v
NavState      ::= | s
NavMem        ::= . a
NavRef        ::= > (key | a)
NavInvRef     ::= < a
```

Figure 3.10: EBNF grammar for ALANLIGHT.

Chapter 4

Static semantics

This chapter presents ALANLIGHT's static semantics. We mainly focus on rules for achieving ALANLIGHT's boundedness properties. The static semantics also includes non-trivial name binding rules and some type checking rules, but we generally assume that name binding and type checking is done before running our checker.

Framework. The inference rules defining ALANLIGHT's static semantics have the following signature (in sequent notation):

$$\boxed{\tau\Omega \vdash \textit{expression} \ @ \ \langle\alpha,\gamma\rangle\Psi\delta}$$

Variables before the turnstile are environments. Variables after the @-sign are results. The *expression* is either a small piece of code or constructor found in the grammar. Rules omit environments or results when they do not apply.

Figure 4.1 presents the types of the environments and the results. τ is the current position of the semantics checker when traversing an *abstract syntax tree* (AST). The current position is either an attribute or a key. The checker uses τ for recursion detection and definition before use checking. Ω is a list of ancestor attributes (collections and state groups) and their corresponding *Type* AST nodes, with respect to τ. Ω includes the *Root* and its *Type* AST node as well. $\langle\alpha,\gamma\rangle$ is a tuple of attributes, keys, or the root with corresponding *Type* AST nodes. The tuple captures the attribute and type that a navigation expression yields. Similar to τ, we use the tuple for recursion detection and definition before use checking. In addition, we use it for checking type equivalence for subexpressions. For expressions yielding an irrelevant attribute or type, we sometimes use *None* as value for α and γ instead.

G is a set of graphs which is used in follow-sets and first-sets. δ is the follow-set that we discussed in Section 3.5. It is a set of mappings from collection attributes to subsets

$\tau : \textit{Attr} \mid \textit{Key}$	$G : \text{Set}\ \{\textit{GraphId}\}$
$\Omega : \text{List}\ \langle\alpha,\gamma\rangle$	$\delta : \text{Set}\ \{\textit{CollectionAttr} \mapsto G \mid \textit{None}\}$
$\langle\alpha,\gamma\rangle : \langle \textit{Root} \mid \textit{Attr} \mid \textit{Key} \mid \textit{None},\ \textit{Type} \mid \textit{None}\rangle$	$\Psi : \text{Map}\ \{\textit{CollectionAttr} \mapsto G\}$

Figure 4.1: Environment and return type definitions for all rules.

$$\boxed{\Omega \vdash a : e \mathbin{@} \checkmark}$$

$$\frac{}{\Omega \vdash a : \texttt{text} \mathbin{@} \checkmark}\ [\text{Text1}]$$

$$\frac{}{\Omega \vdash a : \texttt{integer} \mathbin{@} \checkmark}\ [\text{Integer1}] \qquad \frac{}{\Omega \vdash a : \texttt{natural} \mathbin{@} \checkmark}\ [\text{Natural1}]$$

$$\frac{\begin{array}{l}\forall State(s,\emptyset,t) \in S \\ \Omega \cup \{\langle a,t \rangle\} \vdash t \mathbin{@} \checkmark\end{array}}{\Omega \vdash a : \texttt{stategroup}\ S \mathbin{@} \checkmark}\ [\text{StateGroup1}] \qquad \frac{\Omega \cup \{\langle a,t \rangle\} \vdash t \mathbin{@} \checkmark}{\Omega \vdash a : \texttt{collection}\ g^{*}\ t \mathbin{@} \checkmark}\ [\text{Collection1}]$$

$$\boxed{\vdash e \mathbin{@} \checkmark} \qquad \boxed{\Omega \vdash e \mathbin{@} \checkmark}$$

$$\frac{\{\langle Root,t \rangle\} \vdash t \mathbin{@} \checkmark}{\vdash Root(t) \mathbin{@} \checkmark}\ [\text{Root}] \qquad \frac{\forall a \in a^{*} \quad \Omega \vdash a \mathbin{@} \checkmark}{\Omega \vdash Type(a^{*}) \mathbin{@} \checkmark}\ [\text{Type}]$$

Figure 4.2: Rules for checking a model hierarchy, applying to *Root*, *Type*, and *Attribute* AST nodes without derivation or reference expressions, as presented in Section 3.1.

of their graphs and sometimes also to *None*. We use the follow-set for calculating the sets of graphs that references partake in. Furthermore, we use the follow-set for checking the *first condition* from Section 3.5: at every recursive step we check that every item from δ maps to a non-empty set of graphs. δ contains mappings from collection attributes to *None* for expressions ending in `this` or `parent` navigation steps. Ψ is the first-set for checking the *second condition* that we presented in the previous chapter. Namely, that references preceding a recursive navigation step partake in an identical graph. Ψ maps unique collection attributes to subsets of their graphs.

Root, Type, and Attribute rules. Figure 4.2 presents the rules for the root, types, and attributes holding elementary values; introduced in Section 3.1. The labels of the rules follow constructor names and rule alternatives, typically found on separate lines in the grammar. The bottom left side of the figure shows the root rule, applying to the root and its *Type*. For checking the *Type* of the root, the [Root] rule emplaces the *Root* and its *Type* AST node in the environment Ω. The rule [Type] checks the attributes of a *Type* AST node. We assume that the checker evaluates attributes in the order in which they occur in an Alan-Light model. The remaining rules match attributes that hold elementary values. The rules [StateGroup1] and [Collection1] call the [Type] rule for checking the attributes of their *Type* nodes, recursively.

In Section 3.2 and 3.4 we introduced attributes with expressions for deriving and constraining values. Figure 4.3 presents rules for checking these attributes. In addition to checking *Type* nodes, the rules check expressions for deriving and constraining values. For the rules [Text2], [Integer2], and [Natural2], the rules for the expressions e do all required checking. Figure 4.4 presents the rules for these expressions.

For collection attributes with an expression for deriving or constraining keys, [Collection2]

$$\boxed{\Omega \vdash a : e \ @ \checkmark}$$

$$\frac{a\Omega \vdash e \ @ \ \_\,\_\,\_}{\Omega \vdash a : \texttt{text}\ e \ @ \checkmark}\ [\mathsf{Text2}]$$

$$\frac{a\Omega \vdash e \ @ \ \_\,\_\,\_}{\Omega \vdash a : \texttt{integer}\ e \ @ \checkmark}\ [\mathsf{Integer2}] \qquad \frac{a\Omega \vdash e \ @ \ \_\,\_\,\_}{\Omega \vdash a : \texttt{natural}\ e \ @ \checkmark}\ [\mathsf{Natural2}]$$

$$\frac{\begin{array}{l} k\Omega \vdash k \ @ \ \langle \alpha,\gamma \rangle \Psi\delta \\ \alpha \prec a \vee \alpha \succ a \\ \Omega \cup \{\langle a,t \rangle\} \ \vdash t \ @ \checkmark \end{array}}{\Omega \vdash a : \texttt{collection}\ k\ g^*\ t \ @ \checkmark}\ [\mathsf{Collection2}] \qquad \frac{\begin{array}{l} a\Omega \vdash p \ @ \ \langle \alpha,\gamma \rangle \ \_\,\_ \\ a\Omega \vdash e \ @ \ \langle \alpha,\gamma \rangle \Psi\delta \\ e \in \{RefConstr, RefExp\} \end{array}}{\Omega \vdash a : p\ e \ @ \checkmark}\ [\mathsf{Ref}]$$

$$\frac{\begin{array}{l} a\Omega \vdash e \ @ \ \_\,\_\,\_ \\ \forall State(s,V,t) \in S \\ a\Omega \vdash e \,\#\, Z \quad Z(s) \subseteq \Gamma_s \\ \Omega \cup \{\langle a,t \rangle\} \ \vdash t \ @ \checkmark \end{array}}{\Omega \vdash a : \texttt{stategroup}\ e\ S \ @ \checkmark}\ [\mathsf{StateGroup2}] \qquad \frac{\begin{array}{l} a\Omega \vdash p \ @ \ \langle \alpha_1,\gamma_1 \rangle \Psi\delta \quad \gamma_1 = Type(A) \\ type\text{-}of(r) = \Omega.last() \\ r = \texttt{key} \vee r \in A \end{array}}{\Omega \vdash a : p \ \texttt{*=}\ \texttt{inv-refs}(\,>r\,) \ @ \checkmark}\ [\mathsf{InvRefs}]$$

Figure 4.3: Rules for checking attributes with constraint or derivation expressions, as presented in Section 3.2 and Section 3.4.

calls a rule yielding a collection attribute α and a *Type* AST node γ. The rule also yields a first-set Ψ and a follow-set δ. The second premise of the rule ensures that keys of collection items do not reference other instances of themselves. It also ensures that children of an entity (collection item) do not define its identity; its key. The precedence operator $\prec$ checks if the left-hand attribute precedes the right-hand attribute; ancestor attributes precede (and do not succeed) child attributes. Thus, in the rule the operator checks if α is *defined before* a. The operator $\succ$ checks if α is *fully defined after* (succeeds) attribute a. Note that the operators are not each others inverse; child attributes do *not* succeed ancestor attributes.

We assume that after executing the rule for the *Key* k, we can always retrieve its result with the function def, followed by an @ sign: $def(k)$ @ $\langle \alpha, \gamma \rangle \Psi\delta$. This also applies to (inverse) reference attributes and reference variables (pointing to *Type* AST nodes). For example, $def(a)$ @ $\langle \alpha, \gamma \rangle \Psi\delta$ retrieves results for reference attributes a.

The rule [Ref] checks references with constraint expressions, as well as references with an expression for deriving their value. The path p indicates the attribute and type that a reference attribute points to; that is, a tuple $\langle \alpha, \gamma \rangle$. Notice that in the grammar, the $TypePath$ p is optional for reference attributes. We assume that before running our checker, the tuple is calculated for p from the expression e when p is omitted. This means that recursive expressions have at least one branch that determines the result type. Furthermore, it means that non-recursive branches yield equivalent attributes and types. This also holds for the *Key* AST nodes of collection attributes.

[InvRef] uses the referenced attribute and type tuple of a reference for ensuring that it points to the context attribute and type of the attribute a. For retrieving the tuple, it uses the function $type\text{-}of$. For retrieving the context type, it uses the function $last()$ on the list of ancestors of attribute a. Both function calls should always yield an equivalent result. The

$$\boxed{\tau\Omega \vdash e \;@\; \langle\alpha,\gamma\rangle\Psi\delta}$$

$$\frac{\begin{array}{l}\tau\Omega \vdash e \;@\; \langle\alpha,\gamma\rangle\Psi\delta \\ \mathit{Exp} \in \{\mathit{TextExp}, \mathit{NumExp}\}\end{array}}{\tau\Omega \vdash \mathit{Exp}(e) \;@\; \_\;\Psi\delta}\;[\text{TextExp1}, \text{NumExp1}, 2]$$

$$\frac{\begin{array}{l}\tau\Omega \vdash e_1 \;@\; \langle\alpha_1,\gamma_1\rangle\Psi_1\delta_1 \\ \tau\Omega \vdash e_2 \;@\; \langle\alpha_2,\gamma_2\rangle\Psi_2\delta_2 \\ \Psi_3\delta_3 = \mathit{cac}\text{-}\Psi\delta(\{\Psi_1,\Psi_2\},\{\delta_1,\delta_2\})\end{array}}{\begin{array}{l}\tau\Omega \vdash e_1\; \mathit{NumBinOp}\; e_2 \;@\; \_\;\Psi_3\delta_3 \\ \tau\Omega \vdash \mathit{NumBinFn}\; e_1\; e_2 \;@\; \_\;\Psi_3\delta_3\end{array}}\;[\text{NumExp3}, 4]$$

$$\frac{\tau\Omega \vdash e \;@\; \langle\alpha,\gamma\rangle\Psi\delta}{\tau\Omega \vdash \mathit{NumAggFn}\; e \;@\; \_\;\Psi\delta}\;[\text{NumExp5.1}]$$

$$\frac{\begin{array}{l}\tau\Omega \vdash e_1 \;@\; \langle\alpha_1,\gamma_1\rangle\Psi_1\delta_1 \\ \tau\Omega \vdash e_2 \;@\; \langle\alpha_2,\gamma_2\rangle\Psi_2\delta_2 \\ \Psi_3\delta_3 = \mathit{cac}\text{-}\Psi\delta(\{\Psi_1,\Psi_2\},\{\delta_1,\delta_2\})\end{array}}{\tau\Omega \vdash \mathit{NumAggFn}\; e_1\; v\; e_2 \;@\; \_\;\Psi_3\delta_3}\;[\text{NumExp5.2}]$$

$$\frac{\begin{array}{l}\forall w \in W \quad \tau\Omega \vdash w \;@\; \langle\alpha_w,\gamma_w\rangle\Psi_w\delta_w \\ \mathit{def}(s) = \mathit{State}(s,V,\_) \\ \Psi_r\delta_r = \mathit{cac}\text{-}\Psi\delta(\bigcup_{w\in W}\{\Psi_w\},\bigcup_{w\in W}\{\delta_w\})\end{array}}{\tau\Omega \vdash \mathit{SgExp}(s,W) \;@\; \_\;\Psi_r\delta_r}\;[\text{SgExp1}]$$

$$\frac{\begin{array}{l}\tau\Omega \vdash e \;@\; \langle\alpha_e,\gamma_e\rangle\Psi_e\delta_e \\ \tau\Omega \vdash s_t \;@\; \langle\alpha_t,\gamma_t\rangle\Psi_t\delta_t \\ \tau\Omega \vdash s_f \;@\; \langle\alpha_f,\gamma_f\rangle\Psi_f\delta_f \\ \Psi_3\delta_3 = \mathit{cac}\text{-}\Psi\delta(\{\Psi_e,\Psi_t,\Psi_f\},\{\delta_t,\delta_f\})\end{array}}{\tau\Omega \vdash \texttt{empty}\; e\; v\; s_t\; s_f \;@\; \_\;\Psi_3\delta_3}\;[\text{SgExp2}]$$

$$\frac{\begin{array}{l}\tau\Omega \vdash e \;@\; \langle\alpha_e,\mathit{None}\rangle\Psi_e\delta_e \\ \tau\Omega \vdash s \;@\; \langle\alpha_s,\gamma_s\rangle\Psi_s\delta_s \quad \forall s \in S \\ \Psi_3\delta_3 = \mathit{cac}\text{-}\Psi\delta(\bigcup_{s\in S}\{\Psi_s\}\cup\{\Psi_e\},\bigcup_{s\in S}\{\delta_s\}) \\ \mathit{def}(\alpha_e) = \alpha_e : \texttt{stategroup}\;\_^*\; X \quad \forall s \in S(\exists! x \in X)\end{array}}{\tau\Omega \vdash \texttt{switch}\;(e)\; v\; S \;@\; \langle\alpha_s,\gamma_s\rangle\Psi_3\delta_3}\;[\text{TextExp2}, \text{NumExp6}, \text{SgExp3}, \text{RefExp2}]$$

Figure 4.4: Rules for derivation expressions producing values.

last premise of the rule [InvRef] checks if r is actually a member of the type that p points to. Inverse reference attributes in ALANLIGHT hold the inverse of elementary references.

For state group attributes with an expression e for deriving their value, we apply two rules to the expression. The topmost premise of rule [StateGroup2] checks the expression, ignoring its result. The third premise is a rule call for retrieving results for the parameters v of the possible states. We use the results when checking state types t. Figure 3.3 presents an example of a parameter: the `orders` parameter, with `nonemptyorders` as its value.

The result from executing the third premise of [StateGroup2] is Z. Z captures parameter-specific result sets ($\langle\alpha,\gamma\rangle\Psi\delta$), for all parameters of all possible states. Figure 4.6 defines the signature of the rule. $Z(s)$ is the result set for all parameters of a state s. This set is a subset of the – further omitted – default environment Γ for the *Type* AST node t of a state.

Expression rules. Figure 4.4 presents rules for derivation expressions that produce values (in contrast to references which produce nodes). The first rule applies to *TextExp* and *NumExp* AST nodes having a single *NavExp* AST node. By convention, result types are checked before executing the rules. Therefore, we omit type checking the resulting attribute and corresponding type, $\langle\alpha,\gamma\rangle$. Because further checking $\langle\alpha,\gamma\rangle$ is not required, the rule ignores the result tuple (as the _ indicates). The same holds for other rules ignoring the result tuple. Expression e in rule [TextExp1] has a multiplicity bound of $[1,1]$. That is, when evaluated at runtime, the expression yields exactly one value. The same holds for

$$\mathit{combine}\text{-}\delta(B : \{\delta\}) = \{c \mapsto \cap_{G \in \{G_b | c \mapsto G_b \in b \in B\}} G \mid c \mapsto G_c \in b \in B\} \cup \{c \mapsto \mathit{None} \in b \in B\} \qquad \text{[combine-}\delta\text{]}$$

$$\mathit{combine}\text{-}\Psi(B : \{\Psi\}) = \{c \mapsto \cap_{G \in \{G_b | c \mapsto G_b \in b \in B\}} G \mid c \mapsto \_ \in b \in B\} \qquad \text{[combine-}\Psi\text{]}$$

$$\mathit{cac}\text{-}\Psi(B : \{\Psi\}) = D = \mathit{combine}\text{-}\Psi(B) \qquad \forall c \mapsto G \in D \quad G \neq \emptyset \qquad \text{[cac-}\Psi\text{]}$$

$$\mathit{cac}\text{-}\Psi\delta(A : \{\Psi\}, B : \{\delta\}) = \mathit{cac}\text{-}\Psi(A)\ \mathit{combine}\text{-}\delta(B) \qquad \text{[cac-}\Psi\delta\text{]}$$

Figure 4.5: Key functions for recursion checking, part one. The functions combine first-sets and follow-sets from different subexpressions.The third function checks for non-empty sets of graphs; the second condition from Section 3.5.

expressions e_1 and e_2 in rule [NumExp3, 4], e_2 in rule [NumExp5.2], and e in the rule for `switch` statements.

Rule [NumExp3, 4] has premises for checking two subexpressions e_1 and e_2. The `parts price` attribute from Figure 3.3 depicts such an expression. For calculating the new first-set and follow-set from the sets of the two subexpressions, we combine their results with the function *cac*-$\Psi\delta$ (***c****ombine*-***a****nd*-***c****heck*). Figure 4.5 defines this function. The function combines multiple first-sets and follow-sets, and checks the new combined first-set. First we discuss combining and checking first-sets, and then we discuss combining follow-sets.

Checking the new combined first-set means checking the second condition for recursive calculations from Section 3.5 (last paragraph). The condition is: all first followed references of subexpressions, per level in the hierarchy, that precede a recursive navigation step, partake in at least one identical graph. First-sets of subexpressions contain the graph-sets of first followed references preceding a recursive step, per hierarchical level. Therefore, for checking the second condition we intersects the first-sets of subexpressions, and check that all sets of graphs from the combined first-set are not empty. If the sets of graphs are not empty, identical graphs exist; as per the requirement of the second condition.

For combining follow-sets, we also intersect the sets of graphs from the same hierarchical levels. Follow-sets also contain mappings from collections to *None*. We add the unique occurrences from the follow-sets of subexpressions to the combined follow-set. We need these mappings for subexpressions consisting of a single `this` step (page 30). In our other rules, these mappings ensure that valid expressions always define graph traversal before recursion; see also the first condition.

The remaining rules from Figure 4.4 are similar to rule [NumExp3, 4]. All four rules combine first-sets and follow-sets, and check newly constructed first-sets. But, there are three important details to consider.

First, the conclusions for rule [NumExp5.2] and the last three rules, include a variable assignment v; an AST node *IntlVarAss*. The default environment for e_2 in [NumExp5.2] (after the keyword `mapsto`) captures v (and only v, to simplify time complexity bounds); the value of v is the result from checking e_1. For the rule *SgExp*2, only the default environment for s_f contains v. This is because at runtime, the result of the expression e is an empty set for s_t; the state when `empty` matches `true`. For the rule matching `switch` statements, the default environments for rule calls checking the states s, include v. The value of v is the

$$\boxed{\tau\Omega \vdash e \,\#\, \{s \mapsto \{v \mapsto \langle\alpha,\gamma\rangle\Psi\delta\}\}}$$

$$\frac{\begin{array}{l}\forall w \in W \quad \tau\Omega \vdash w \;@\; \langle\alpha_w,\gamma_w\rangle\Psi_w\delta_w \\ |W| = |V| \quad def(s) = State(s,V,\_) \\ z = \{V(i) \mapsto \langle\alpha_{W(i)},\gamma_{W(i)}\rangle\; \Psi_{W(i)}\delta_{W(i)} \mid 1 \le i \le |W|\}\end{array}}{\tau\Omega \vdash SgExp(s,W) \,\#\, \{s \mapsto z\}} \quad \text{[SgExp1b]}$$

$$\frac{\begin{array}{l}\forall o \in O \quad \tau\Omega \vdash o \,\#\, Z_o \quad Q = \bigcup_{o\in O}\{Z_o\} \\ S = \{s \mid s \mapsto \_ \in Z \in Q\} \quad def(s) = State(s,V,\_)\} \\ C(s,v) = \bigcup_{\{s\mapsto V\}\in Z\in Q}\{\langle\alpha_i,\gamma_i\rangle \mid V(v) \mapsto \langle\alpha_i,\gamma_i\rangle\,\_\,\_\} \quad =_{\langle\alpha_C,\gamma_C\rangle\in C(s,v)} \gamma_C \\ \Psi(s,v) = combine\text{-}\Psi(\bigcup_{\{s\mapsto V\}\in Z\in Q}\{\Psi_v \mid V(v) \mapsto \_\;\Psi_v\;\_\}) \\ \delta(s,v) = combine\text{-}\delta(\bigcup_{\{s\mapsto V\}\in Z\in Q}\{\delta_v \mid V(v) \mapsto \_\,\_\;\delta_v\}) \\ Z_S = \{s \in S \mapsto \{v \in V \mapsto \langle\alpha,\gamma\rangle \in C(s,v)\; \Psi(s,v)\; \delta(s,v)\}\end{array}}{\tau\Omega \vdash \texttt{empty}\,\_\,\_\,O \,\#\, Z_S \qquad \tau\Omega \vdash \texttt{switch}\,\_\,\_\,O \,\#\, Z_S} \quad \text{[SgExp2b, SgExp3b]}$$

Figure 4.6: Additional navigation rules for parametrized state types.

result from evaluating e. But, for each state $s \in S$, *None* is replaced with its corresponding state type.

Second, the last two rules ignore follow-set δ_e when combining follow-sets. In Section 3.5 (page 30) we demonstrated that some recursive calculations require this. We can safely ignore follow-sets from subexpressions that do not produce the final result of a calculation. For example, suppose that an expression uses a reference matching rule [RefExp2]. Then, subexpressions for the states S determine the referenced node. When following the reference, we navigate to the referenced node. Therefore, we traverse graphs that the subexpressions for the states S traverse. We do not traverse graphs that the expression e from rule [RefExp2] traverses. Because the follow-set for a reference determines the graphs we traverse when following it, we can ignore graphs from the follow-set for e. Therefore, the rule ignores the follow-set δ_e when constructing the resulting follow-set for the `switch` statement.

Third, the last line of the rule for `switch` statements has two premises that are specifically tailored to `switch` statements. The first one uses the function $def(\alpha)$ followed by an =-sign for retrieving the definition of an attribute. We also use this for retrieving definitions from names of states and attribute labels. The second one ensures that `switch`-statements handle every state of a state group; it prevents undefined and nondeterministic behaviour.

Figure 4.6 presents the rules for calculating the first-sets and follow-sets of state type parameters. We discussed these in the previous paragraph. The signature of the rules differs from the signature of the other rules constituting our static semantics. Instead of producing a single result set $\langle\alpha,\gamma\rangle\Psi\delta$, the rules produce a result set per parameter, per state type. [SgExp1b] matches the end of state group expressions, where we choose a state. There, we also define the arguments W to the parameters of the state. The first premise calculates the result sets for the arguments. Subsequently, the second premise checks if every parameter of the state has a corresponding argument. Finally, the premise on the third line constructs z: a mapping for all parameters to their specific result sets. The result of the rule is a mapping from the state to z.

$$\boxed{\tau\Omega \vdash e \;@\; \langle\alpha,\gamma\rangle\Psi\delta}$$

$$\frac{\begin{array}{l}\tau\Omega \vdash e \;@\; \langle\alpha_e,\gamma_e\rangle\Psi_e\delta_e\\ \tau\Omega \vdash w \;@\; \langle\alpha_w,\gamma_w\rangle\Psi_w\delta_w \quad \forall w\in w^*\\ \Psi_c=\{\Psi_e\}\cup\bigcup_{w\in w^*}\{\Psi_w\}\\ \Psi_3=\mathit{cac}\text{-}\Psi(\Psi_c)\end{array}}{\tau\Omega \vdash \texttt{=}\; e\; v\; w^* \;@\; \langle\alpha_e,\gamma_e\rangle\Psi_3\delta_e}\;[\text{ColExp1.1a}]$$

$$\frac{\begin{array}{l}\tau\Omega \vdash e \;@\; \langle\alpha_e,\gamma_e\rangle\Psi_e\delta_e\\ \tau\Omega \vdash i \;@\; \langle\alpha_e,\gamma_e\rangle\Psi_i\delta_i \quad \forall i\in i^*\neq\emptyset\\ \Psi_c=\{\Psi_e\}\cup\bigcup_{i\in i^*}\{\Psi_i\}\\ \Psi_3=\mathit{cac}\text{-}\Psi(\Psi_c)\\ \delta_4=\mathit{combine}\text{-}\delta(\bigcup_{i\in i^*}\{\delta_i\})\end{array}}{\tau\Omega \vdash \texttt{=}\; e\; v\; i^* \;@\; \langle\alpha_e,\gamma_e\rangle\Psi_3\delta_4}\;[\text{ColExp1.1b}]$$

$$\frac{\tau\Omega \vdash e \;@\; \langle\alpha,\gamma\rangle\Psi\delta}{\tau\Omega \vdash \texttt{where(}\; e\;\texttt{)}\; v \;@\; \langle\alpha,\gamma\rangle\Psi\delta}\;[\text{ColExp1.2}]$$

$$\frac{\begin{array}{l}\tau\Omega \vdash e_1 \;@\; \langle\alpha_1,\gamma_1\rangle\Psi_1\delta_1\\ \tau\Omega \vdash e_1\; r \;@\; \langle\alpha_2,\gamma_2\rangle\Psi_2\delta_2\end{array}}{\tau\Omega \vdash \texttt{in(}\; e_1\; r\;\texttt{)}\; v \;@\; \langle\alpha_2,\gamma_2\rangle\Psi_2\delta_2}\;[\text{ColExp1.3}]$$

$$\frac{\begin{array}{l}\tau\Omega \vdash e_1\; e_2 \;@\; \langle\alpha,\gamma\rangle\Psi\delta_1\\ \langle\alpha,\gamma\rangle\in\Omega \quad \delta_2=\{\alpha\mapsto G\}\end{array}}{\tau\Omega \vdash \texttt{\~{}ref(}e_1\; e_2\;\texttt{[}\; G\;\texttt{])} \;@\; \langle\alpha,\gamma\rangle\Psi\delta_2}\;[\text{RefConstr1}]$$

$$\frac{\tau\Omega \vdash e \;@\; \langle\alpha,\gamma\rangle\Psi\delta}{\tau\Omega \vdash \texttt{\~{}ref(}\; e\;\texttt{)} \;@\; \langle\alpha,\gamma\rangle\Psi\delta}\;[\text{RefConstr2}]$$

$$\frac{\tau\Omega \vdash e \;@\; \langle\alpha,\gamma\rangle\Psi\delta}{\tau\Omega \vdash \mathit{RefExp}(e) \;@\; \langle\alpha,\gamma\rangle\Psi\delta}\;[\text{RefExp1}]$$

Figure 4.7: Rules for reference constraint expressions and derivation expressions producing references.

Sometimes, different subexpressions of an *SgExp* expression map to the same state. Result sets for equivalent parameters may vary. Therefore, the rule with labels [SgExp2b] and [SgExp3b] combines result sets, per parameter. The first premise evaluates the subexpressions. Q captures all result sets for further processing. S is the set of unique states that occur Q. The three immediately following function premises together construct the combined result set for a parameter v of a state s. The first function $C(s,v)$ determines the parameter specific result tuple $\langle\alpha,\gamma\rangle$. For type safety concerns, ALANLIGHT requires type equivalence for different arguments matching the same parameter v. The equivalence check after the function ensures this. The second function $\Psi(s,v)$ calculates the combined first-set. We purposely omit checking the combined first-set, as the rules [SgExp1], [SgExp2], and [SgExp3] already do that. The third function $\delta(s,v)$ calculates the combined follow-set. Z_4 uses the functions for calculating the result of the inference rule. It maps all states in S to corresponding parameter mappings, and maps each parameter to its specific result set.

Reference rules. The previous paragraph introduced one rule that applies to references as well as expressions producing values: $RefExp2$. In addition the paragraph introduced rules applying to variables, which behave similarly to references. Figure 4.7 presents all other rules for expressions yielding references.

The rule [ColExp1.1a] applies to expressions for deriving collections using (optional) filters. The first premise checks the expression e for selecting a unique collection of nodes. The expression requires exactly one collection navigation step at the end of the expression in order to guarantee unique keys (unique string key values). The second premise calls a rule for checking the `where` clauses w^* that serve as filters. The default environment for where clauses includes the variable v; the result from e. Similar to expressions from the previous paragraph, the rule combines results from all subexpressions. But, it uses the function *cac*-Ψ from Figure 4.5 directly, instead of *cac*-$\Psi\delta$. δ_e already determines the combined follow-set; that is, the follow-set for the navigation step `>key`. Every `where` clauses has a separate result set $\langle\alpha_w,\gamma_w\rangle\Psi_w\delta_w$. We use these for checking expressions with a *NavVar* step referencing the name v of the `where` clause. Figure 3.3 shows such a step with an expression for retrieving the `price` of `simple_products`.

Section 3.5 (page 30) presented `in` clauses for deriving collections by aggregating references. The rule labeled [ColExp1.1b] applies to expressions with such clauses. The rule constructs a follow-set for the `key` using the results from `in` clause checking. We explained this in the previous paragraph, when discussing the last rule from Figure 4.4. Notice that the rule requires equivalent result tuples $\langle\alpha_e,\gamma_e\rangle$ for e and all `in` clauses.

[ColExp1.2] presents the rule for `where` clauses. It has a single premise for checking the navigation expression e. The variable v stores the result from the expression. The rule [ColExp1.3] has two premises for checking `in` clauses. The variable v stores the result set of the first premise. The second premise determines the actual result set from the `in` clause. Child attributes of the collection attributes' *Type* can use the variable results from in clauses. The figure from page 30 demonstrates this.

Rule [RefExp1] from Figure 4.7 checks expressions for deriving references. In Section 3.5 we gave several examples of such expressions. Examples included expressions for calculating `preferred_product` references and `latest` product version references. For checking the expressions we use a single premise that evaluates the *NavExp* AST node e.

The rules [RefConstr1] and [RefConstr2] check reference constraint expressions. Reference constraint expressions require a single collection navigation step at the end of the expression. The rule [RefConstr1] matches explicit self references that define graph participation. We first discussed these references in Section 3.5 when introducing graphs. The follow-sets of such references contain one element: a mapping from the collection attribute – referenced at the end of the expression e – to the predefined set of respected graphs G. [RefConstr2] applies to all other reference constraint expressions, not partaking in graphs. In the next paragraph we discuss the rules for navigation expressions.

Navigation rules. Figures 4.8, 4.10, and 4.11 present the rules for navigation expressions; the bottom section of the grammar: *NavExp*, *TypePath*, and their parts. The four rules from Figure 4.8 check the heads of navigation expressions: the AST node *NavStart*. At the start of a navigation expression, we construct the initial attribute-type tuple and the initial first-set and follow-set.

The ancestor list Ω stores the ancestors of the current position τ. Rule [TypePath1] uses the function *first* for retrieving the first item from this list: the root and its *Type* AST node. Expressions starting at the **`root`** may not be recursive; the root has no ancestor graphs to traverse. For guaranteeing boundedness, the follow-set resulting from a `root` step stores a mapping from the root to *None*. If a follow-set contains *None*, recursion is not allowed. Sometimes we ignore less significant items from follow-sets. But, the root is by definition the most significant item in any follow-set, ensuring no recursion after a `root` step. The rule

$$\boxed{\tau\Omega \vdash e \mathbin{@} \langle\alpha,\gamma\rangle\Psi\delta}$$

$$\frac{\langle\alpha_1,\gamma_1\rangle = \Omega.\mathit{last}() \quad \langle\alpha_2,\gamma_2\rangle = \Omega.\mathit{entity\text{-}of}(\alpha_1) \quad \delta = \{\alpha_2 \mapsto \mathit{None}\} \quad \Psi = \emptyset}{\tau\Omega \vdash \texttt{this} \mathbin{@} \langle\alpha_1,\gamma_1\rangle\Psi\delta}\ \text{[NavThis1.1]}$$

$$\frac{\tau\Omega \vdash e \mathbin{@} \langle\alpha_1,\gamma_1\rangle\Psi\delta_1 \quad \langle\alpha_2,\gamma_2\rangle = \Omega.\mathit{parent\text{-}of}(\alpha_1) \quad \langle\alpha_3,\gamma_3\rangle = \Omega.\mathit{entity\text{-}of}(\alpha_2) \quad \delta_2 = \mathit{next}\text{-}\delta(\delta_1, \{\alpha_3 \mapsto \mathit{None}\})}{\tau\Omega \vdash e.\texttt{parent} \mathbin{@} \langle\alpha_2,\gamma_2\rangle\Psi\delta}\ \text{[NavThis1.2]}$$

$$\frac{\mathit{def}(v) \mathbin{@} \langle\alpha,\gamma\rangle\Psi\delta}{\tau\Omega \vdash v \mathbin{@} \langle\alpha,\gamma\rangle\Psi\delta}\ \text{[NavVar]}$$

$$\frac{\langle\alpha,\gamma\rangle = \Omega.\mathit{first}() \quad \delta = \{\alpha \mapsto \mathit{None}\} \quad \Psi = \emptyset}{\tau\Omega \vdash \texttt{root} \mathbin{@} \langle\alpha,\gamma\rangle\Psi\delta}\ \text{[TypePath1]}$$

Figure 4.8: Rules for *NavStart* AST nodes and their children (bottom section of grammar), and the `root` step in a *TypePath*.

[NavVar] retrieves the result set for a variable v from the default environment. A variable v is a variable that an AST node *IntlVarAss* or *ExtlVarAss* defines, or a *State* parameter.

The rule [NavThis1.1] checks a `this` step. For a `this` step, the rule retrieves the *last* tuple from the list of ancestors. This tuple contains the immediate parent attribute and type of the current position τ in the AST; this is the result tuple of the `this` step. The initial first-set is an empty set, as a navigation expression consisting of only a `this` step does not contain recursion. The third premise constructs the initial follow-set. The initial follow-set maps a single collection attribute to *None*. We shortly discussed the need for *None* in the paragraph on expression rules, and also in Section 3.5 (page 30). We use *None* for ensuring that an expression defines graph traversal before a recursive step. For constructing the follow-set, the second premise retrieves the context collection with the function *entity-of*. If α_1 is a collection attribute, the *entity-of* α_1 is α_1 itself.

Rule [NavThis1.2] checks the second alternative to *NavThis* AST nodes: `parent` steps. The first premise calls a rule for evaluating the expression e. For retrieving the parent attribute and type of the result from e, we use the function *parent-of*. For constructing the follow-set of the navigation *step* we retrieve the context collection attribute as well. For constructing the follow-set of the navigation expression e together with the parent step, we use the function *next*-δ. This function generalizes follow-set calculation for all navigation steps requiring it. Figure 4.9 defines *next*-δ. It calculates a new follow-set from two follow-sets; δ_1 and $\alpha_3 \mapsto \mathit{None}$ in the rule [NavThis1.2]. For the parent navigation step, the follow-set resulting from executing *next*-δ equals the follow-set of the step: $\alpha_3 \mapsto \mathit{None}$. This is because the grammar ensures that e only contains `this` steps and additional `parent` steps. Below we explain the function in more detail.

In Section 3.5 we extensively discussed follow-set construction. We can ignore less significant collections when (implicitly) navigating to more significant ones. To this end, the function *next*-δ calls the function *remove-LSC*. *remove-LSC* filters a follow-set on mappings for collections that are at least as significant as a collection c_{LSC}. It uses the earlier defined operator $\prec$ for checking if a collection is an ancestor of c_{LSC}. The c_{LSC} for follow-sets δ_1 and δ_2 are the least significant (*last*) ones from δ_2 and δ_1, respectively.

Sometimes, follow-sets contain *None*. In such cases, recursive steps in calculations

$$\textit{remove-LSC}(\delta, c_{LSC} : \textit{CollectionAttr}) = \{c_i \mapsto \_ \in \delta \mid c_i \prec c_{LSC} \lor c_i \equiv c_{LSC}\} \qquad \text{[remove-LSC]}$$

$$\begin{aligned}
&\textit{next-}\delta(\delta_1, \delta_2) = \textit{combine-}\delta(\{\delta_5, \delta_6\}) \text{ where}\\
&\quad c_1 = \delta_1.\textit{collections}().\textit{last}()\\
&\quad c_2 = \delta_2.\textit{collections}().\textit{last}()\\
&\quad \delta_3 = \textit{remove-LSC}(\delta_1, c_2)\\
&\quad \delta_4 = \textit{remove-LSC}(\delta_2, c_1)\\
&\quad \delta_5 = \delta_3 \setminus \{c_2 \mapsto \textit{None} \mid c_2 \mapsto \textit{None} \notin \delta_4 \land c_2 \mapsto G \in \delta_4\}\\
&\quad \delta_6 = \delta_4 \setminus \{c_1 \mapsto \textit{None} \mid c_1 \mapsto \textit{None} \notin \delta_3 \land c_1 \mapsto G \in \delta_3\} \qquad \text{[next-}\delta\text{]}
\end{aligned}$$

$$\textit{next-}\Psi(\Psi, \delta) = \Psi \cup \{c \in \delta \mid c \notin \Psi\} \qquad \forall c \mapsto P \in \delta \quad P \neq \textit{None} \land P \neq \emptyset \qquad \text{[next-}\Psi\text{]}$$

Figure 4.9: Key functions for recursion checking, part two. The first function constructs a new follow-set, dropping collections that are less significant than collection c_{LSC}. The second function calculates a new follow-set from two follow-sets; typically a current/context follow-set, and the follow-set of a navigation step.

require a leading navigation step following a reference in an acyclic graph. Enabling recursion after the reference navigation step, requires modifying the follow-set. Namely, recursive steps require that all items in a follow-set map to a non-empty set of graphs; they may not map to *None*. To this end, the function *next*-δ calculates δ_5 and δ_6. We subtract the *None* case for the least significant collection c_2 from δ_3. That is, if and only if δ_4 only maps c_2 to a set of graphs, and not to *None*. For calculating δ_6 we repeat the same process, but for δ_4. We subtract $\{c_1 \mapsto \textit{None}\}$ from δ_4, while checking mappings from δ_3. After constructing δ_5 and δ_6, we combine them with the function *combine*-δ from Figure 4.5. This yields the follow-set that *next*-δ returns. Notice that the follow-set $\{\alpha_3 \mapsto \textit{None}\}$ for rule [NavThis1.2] equals the result from calling the function.

Figures 4.10 and 4.11 define the rules for tails of navigation expressions. As they are always preceded by the head of a navigation expression, their first premise calls a rule for checking this head e. The rule [NavState] requires that e yields a tuple of a state group attribute and *None*. *None*, because a state step determines the type γ of the tuple; a state group has no default state type. All other remaining rules also call a rule for evaluating preceding navigation steps e. By convention, the resulting type γ_1 of the rule call is always a *Type* AST node. We also assume that every navigation step to an attribute or key is correct. That is, an attribute reference in a navigation step binds to an attribute of γ_1. Furthermore, a key step binds to the AST node *Key* of a collection attribute with type γ_1.

The rules [NavMem1.1] and [NavMem1.2] match navigation steps attributes a holding values. The rule [NavMem1.1] checks non-recursive steps, as the second line of the rule expresses. These are steps where τ is neither the attribute α_m nor the *Key* of collection attribute α_m. [NavMem1.2] checks recursive steps, where the opposite is true.

The third line of [NavMem1.1] ensures that α_m either precedes τ, or holds *elementary* values. We discussed this requirement in Section 3.4. Expressions use earlier defined *derived* value attributes or elementary value attributes. The premise on the fourth line ensures that reference constraints only depend on elementary values: either the referenced attribute α_m holds elementary values or the target τ holds derived values. The last premise

$$\boxed{\tau\Omega \vdash e \;@\; \langle\alpha,\gamma\rangle\Psi\delta}$$

$$\frac{\begin{array}{l}\tau\Omega \vdash e \;@\; \langle\alpha, None\rangle\Psi\delta\\ def(\alpha)=\alpha : \texttt{stategroup}\ \_\ S\\ s\in S \quad def(s)=State(s,\_,t)\end{array}}{\tau\Omega \vdash e\,|\,s \;@\; \langle\alpha,t\rangle\Psi\delta}\quad[\text{NavState}]$$

$$\frac{\begin{array}{l}\tau\Omega \vdash e \;@\; \langle\alpha_1,\gamma_1\rangle\Psi_1\delta_1 \quad \alpha_m = def(a)\\ \alpha_m\neq\tau \quad \alpha_m\neq a : \texttt{collection}\ \tau\ \_\ \_\\ \alpha_m\prec\tau \vee elementary(\alpha_m)\\ elementary(\alpha_m)\vee derived(\tau)\\ \gamma_2=\begin{cases} t & \text{if } \alpha_m = a : \texttt{collection}\ \_^{*}\ t\\ None & \text{otherwise}\end{cases}\end{array}}{\tau\Omega \vdash e\ .\ a \;@\; \langle\alpha_m,\gamma_2\rangle\Psi_1\delta_1}\quad[\text{NavMem1.1}]$$

$$\frac{\begin{array}{l}\tau\Omega \vdash e \;@\; \langle\alpha_1,\gamma_1\rangle\Psi_1\delta_1 \quad \alpha_m = def(a)\\ \alpha_m=\tau \vee \alpha_m = a : \texttt{collection}\ \tau\ \_\ \_\\ \gamma_2=\begin{cases} t & \text{if } \alpha_m = a : \texttt{collection}\ \_^{*}\ t\\ None & \text{otherwise}\end{cases}\\ \Psi_2 = next\text{-}\Psi(\Psi_1,\delta_1)\end{array}}{\tau\Omega \vdash e\ .\ a \;@\; \langle\alpha_m,\gamma_2\rangle\Psi_2\delta_1}\quad[\text{recursion} \mid \text{NavMem1.2}]$$

Figure 4.10: Rules for *NavExp* and *TypePath* AST nodes and their children for member access, after *NavStart* (bottom section of grammar).

$$\boxed{\tau\Omega \vdash e \;@\; \langle\alpha,\gamma\rangle\Psi\delta}$$

$$\frac{\begin{array}{l}\tau\Omega \vdash e \;@\; \langle\alpha_1,\gamma_1\rangle\Psi_1\delta_1 \quad \alpha_m = def(x)\\ \alpha_m\neq\tau \quad \alpha_m\prec\tau \quad elementary(a_m)\vee derived(\tau)\\ def(x) \;@\; \langle\alpha_2,\gamma_2\rangle\Psi_2\delta_2\\ \delta_{>} = \{c\mapsto\_\in\delta_2 \mid \langle c,\_\rangle\in\Omega\}\\ \delta_3 = next\text{-}\delta(\delta_1,\delta_{>})\end{array}}{\tau\Omega \vdash e > x \;@\; \langle\alpha_2,\gamma_2\rangle\Psi_1\delta_3}\quad[\text{NavRef1.1}]$$

$$\frac{\begin{array}{l}x\Omega \vdash e \;@\; \langle\alpha_1,\gamma_1\rangle\Psi_1\delta\\ type\text{-}of(x)=\langle\alpha_2,\gamma_2\rangle\\ \Psi_2 = next\text{-}\Psi(\Psi_1,\delta)\end{array}}{x\Omega \vdash e > x \;@\; \langle\alpha_2,\gamma_2\rangle\Psi_2\delta}\quad[\text{recursion} \mid \text{NavRef1.2}]$$

$$\frac{\begin{array}{l}\tau\Omega \vdash e \;@\; \langle\alpha_1,\gamma_1\rangle\Psi_1\delta_1 \quad \alpha_m = def(a)\\ \alpha_m\prec\tau \quad \alpha_m = a : p = \texttt{inv-refs}(\,> x\,)\\ def(p) \;@\; \langle\alpha_2,\gamma_2\rangle\Psi_2\delta_2\\ \delta_{<} = \{c\mapsto\_\in\delta_2 \mid \langle c,\_\rangle\in\Omega\}\\ \delta_3 = next\text{-}\delta(\delta_1,\delta_{<})\end{array}}{\tau\Omega \vdash e < a \;@\; \langle\alpha_2,\gamma_2\rangle\Psi_1\delta_3}\quad[\text{NavInvRef}]$$

Figure 4.11: Rules for *NavExp* AST nodes and their children for following references (bottom section of grammar).

determines the result type γ_2 of the rule. For collection attributes, γ_2 is the *Type t* of the collection; for other attributes, it is *None*. Notice that [NavMem1.1] does not modify first-sets and follow-sets, as it evaluates navigation to child attributes.

Rule [NavMem1.2] applies to recursive navigation steps; follow-sets determine validity of recursive navigation steps. The last premise calls the function [next-Ψ]. This function ensures that the resulting follow-set from e supports recursion; that the follow-set neither holds a mapping to *None*, nor a mapping to an empty graph-set. This conforms to the second condition from Section 3.5. The second condition concerns first followed self references per hierarchical level. For that reason, Ψ_2 consists of Ψ_1 combined with mappings from δ_2 for hierarchical levels (collection attributes) that are not found in Ψ_1. Function [next-Ψ] also produces a new first-set, Ψ_2 is the new first-set; the first-set after the recursive step.

The rules [NavRef1.1] and [NavRef1.2] check reference navigation steps. Similar to the rules for *NavMem* AST nodes, the rule [NavRef1.1] matches non-recursive steps, whereas [NavRef1.2] matches recursive steps. Reference steps are `key` steps or navigation steps to reference attributes. The variable x in rules [NavRef1.1] and [NavRef1.2] match both `key` steps as well as attribute steps a.

The second line from rule [NavRef1.1] expresses that the rule is not recursive. Furthermore, it defines that expressions always use earlier defined references. We discussed this in Section 3.4. The third premise from the third line ensures that elementary references only use (other) elementary references to constrain values. The remaining two lines from [NavRef1.1] define the follow-set after a reference step. To this end, we first construct $\delta_>$. $\delta_>$ filters the follow-set from the reference x on mappings for ancestors of the target attribute τ. We explained this in Section 3.5 (page 36). Namely, forward and backward reference may traverse ancestor graphs of the target attribute τ. For safely supporting recursion after following such references, the ancestor graphs are important. They determine the allowed recursive navigation steps for which ALANLIGHT can guarantee boundedness. For calculating the final follow-set after the reference step, we add the filtered follow-set to δ_1. For this purpose, we use the earlier presented function *next*-δ.

Valid recursively defined references match rule [NavRef1.2]. For checking the expressions defining such references, the first-set and follow-set are unknown during rule evaluation. To this end, the rule uses the function $type\text{-}of$ for retrieving the result attribute and type. Instead of explicitly retrieving the definition of x with def, we assume that the rule retrieves it automatically. Equivalent to [NavMem1.2], the last premise calls [next-Ψ], which checks the follow-set and constructs a new first-set.

The rule [NavInvRef] evaluates inverse reference attribute navigation steps. The first line of the rule evaluates the expression e and retrieves the definition of the inverse reference attribute. The second line ensures that expressions only use earlier defined inverse references, and that a points to an inverse reference attribute. x is the reference for which a_m stores the inverse. This reference is the `key` of a collection item or a reference attribute. The remaining two premises are equivalent to those from rule $NavRef1.1$.

Many modifications to our grammar and semantics are possible for supporting even more complex calculations. We discuss some of those in our recommendations for future work. In the next chapter we cover a dynamic semantics for ALANLIGHT.

Chapter 5

Dynamic Semantics

This chapter presents a straight-forward dynamic semantics for ALANLIGHT, in the form of a big step operational semantics. The purpose of our operational semantics is to demonstrate the intended interpretation of concepts found in ALANLIGHT. In addition, its purpose is to show that ALANLIGHT prevents duplicate calculations. Our operational semantics defines a recursive on-read strategy for calculating derived values, enabling on-demand minimal effort evaluation. We conclude this chapter showing that the number of calculations for ALANLIGHT programs is bounded above in terms of the size of the specification (the model) and input.

ALANLIGHT guarantees bounded calculations; constraints on data ensure this. Moreover, ALANLIGHT guarantees that expressions for an attribute yield a value of a corresponding predefined (or inferable) type. Thus, ALANLIGHT ensures that expressions meet all multiplicity bounds for operations. For example, the operation `max` requires a non-empty set of nodes, meaning that using the operation requires first checking for a non-empty set of nodes. For this purpose, ALANLIGHT requires using the `empty` expression for deriving the state of a state group attribute (Section 3.4). In addition, we assume that ALANLIGHT ensures that numerical expressions are valid, meaning that division by zero cannot occur. It also means that expressions for natural numbers produce a value greater than zero.

Initialization and modification. For the rules defining the operational semantics of ALANLIGHT, we assume the existence of an in memory data structure; a *store*. This store holds basic elementary data conforming to the data structure that an ALANLIGHT model defines.

After loading data into the store, we query all elementary references once for checking referential integrity (reference constraints and graph constraints). A dataset satisfies all constraints if every elementary reference yields a non-empty result (containing a node, an instance of a *Type* AST node), and every graph constraint is satisfied. We assume rechecking all constraints after updating elementary values in the store, discarding earlier calculated results. For querying derived values, our semantics supports calculating derived values on the fly, caching intermediate calculated values. In summary, we check all constraints on initialization and modification, and calculate derived values on demand.

Before applying rules defining ALANLIGHT's operational semantics, we assume that all required semantic information is available. For example, we assume that names are bound to corresponding definitions (and we can use them interchangeably). Furthermore, we assume that all names are unique. For attributes or keys with reference constraints, a set named $InvRefs$ holds inverse references attributes that point to it (with r in our static semantics).

5.1 Evaluation rules

The evaluation rules defining the operational semantics of ALANLIGHT, have the following signature:

$$\boxed{\Theta\sigma \vdash \textit{expression} \;/\; \Sigma \Downarrow \mu \;/\; \Sigma}$$

Similar to the static semantics rule signature, variables before the turnstile are environments. Variables after the $\Downarrow$-sign are return values; the values that the rule evaluates to. Σ is the store, storing user data and calculated values. All rules evaluate expressions in the context of a store and yield a new store (after the $\Downarrow$-sign). We omit the store in evaluation rules not explicitly using (i.e. reading or modifying) it. When omitted, we assume top-down left-to-right threading of the store through the different premises. For evaluation rules that are executed more than once (because of $\forall$ premises), we assume sequential threading of the store. That is, a subsequent rule evaluation uses the store resulting from its immediately preceding rule evaluation.

The store has the following signature:

$$\Sigma : \{Object \mapsto Member \mapsto Set\langle type\rangle\}$$

That is, the store maps objects to members to a set of equivalently typed objects, where

$$type \in \{\texttt{text}, \texttt{integer}, Type\}$$

The store holds a set for each member, such that evaluation rules can treat all attribute types identically. An instance of a text attribute in ALANLIGHT corresponds to a *Member* mapping to a *Set* storing a single `text` object; a string value. For integer and natural attributes, the object is of type `integer`; a number value. For state group attributes, it is an object conforming to a *Type* AST node of one of its states. For collection attributes, the set holds multiple objects conforming to its *Type* AST node.

The environment Θ maps variable names to a bag holding equivalently typed values: $\{VarName \mapsto Bag\langle type\rangle\}$. Navigation expressions use Θ for retrieving (sub)expression-scoped variable values that *IntlVarAss* AST nodes specify. The environment σ is a pointer to the context node (the `this` node) when evaluating an expression; it is an instance of a user defined *Type*. The evaluation rules use it for executing the `this` step in navigation expressions.

The result of an expression is μ, a *bag* of equivalently typed objects: $Bag\langle type\rangle$. Notice that Θ maps variable names to μ instances. The store holds sets that support interpretation as μ instances.

$$\boxed{\Theta\sigma \vdash \mathit{expression} \,/\, \Sigma \Downarrow \mu \,/\, \Sigma}$$

$$\frac{}{\Theta\sigma \vdash \texttt{this} \Downarrow \{\sigma\}}\ [\mathsf{NavThis1.1}] \qquad \frac{\Theta\sigma \vdash e \,/\, \Sigma \Downarrow \mu \,/\, \Sigma \quad \mu_p = \bigcup_{o\in\mu}\{\Sigma(o,\texttt{parent})\}}{\Theta\sigma \vdash e\texttt{.parent} \,/\, \Sigma \Downarrow \mu_p \,/\, \Sigma}\ [\mathsf{NavThis1.2}]$$

$$\frac{\Sigma(\sigma,v)=\mu}{\Theta\sigma \vdash v \,/\, \Sigma \Downarrow \mu \,/\, \Sigma}\ [\mathsf{NavVar1}] \qquad \frac{\Theta(v)=\mu}{\Theta\sigma \vdash v \Downarrow \mu}\ [\mathsf{NavVar2}]$$

Figure 5.1: Operational semantics for *NavStart*.

We discuss the operational semantics of ALANLIGHT by starting at the smallest syntactical components, working our way up. The organization of evaluation rules strongly corresponds to the organization of the rules defining ALANLIGHT static semantics. Labels of the rules follow constructor names and rule alternatives, typically found on separate lines of the grammar (Figure 3.10).

Navigation expression evaluation. Figure 5.1 presents the rules for *NavStart* components. Rule [NavThis1.1] evaluates to a bag holding the context object σ. Every instance μ of a *Type* AST node stores the location of its `parent` object. Rule [NavThis1.2] retrieves the `parent` for each object o in the bag μ.

Rule [NavVar1] evaluates references to state parameter values and to results from `in` or `where` clause of a collection expression. The context object σ stores the values; the rule retrieves the value of a variable v from the store. Rule [NavVar2] evaluates references to expression-scoped variables (assigned with an `as` statement). It retrieves the value of a variable v from the variables environment Θ. The static semantics of ALANLIGHT guarantees successful retrieval of a value for each variable reference.

Figure 5.2 presents rules evaluating navigation steps. [NavState] evaluates state navigation steps. The first premise evaluates the expression e, which yields a bag of nodes μ. Subsequently, it filters out nodes with state definitions not matching state s. All object instances of states store a member **_state**; a special member holding the state definition of a state group attribute instance. The rule retrieves the state definition **_state** of μ from the store for checking against s.

Rules [NavMem], [NavRef], and [NavInvRef] evaluate member navigation steps using a function **get** for retrieving a member a from every object in μ. Figure 5.3 defines different variants of this function, matching different types of members (attributes and keys). Below we explain these functions.

Rule [**get1**] retrieves *basic* elementary attribute values. The rule retrieves a set μ_a from the store: the value of member a from node o. Rule [**get2.1**] retrieves the calculated value of an attribute holding a *derived* value (or reference). Similarly, [**get3.1**] retrieves the calculated referenced node that a reference attribute or key points to. Note that all rules that are not related to a syntactic component use a bold format.

Sometimes, retrieving attribute or key values requires calculating them first. Rules [**get2.2**] and [**get3.2**] match these cases; when the store does not contain a value for the

$$\boxed{\Theta\sigma \vdash \mathit{expression} \,/\, \Sigma \Downarrow \mu \,/\, \Sigma}$$

$$\frac{\begin{array}{l}\Theta\sigma \vdash e \,/\, \Sigma \Downarrow \mu \,/\, \Sigma_e \\ \mu_s = \bigcup_{o\in\mu}\{o \mid \Sigma_e(o, \_\mathbf{state}) = s\}\end{array}}{\Theta\sigma \vdash e \mid s \,/\, \Sigma \Downarrow \mu_s \,/\, \Sigma_e}\ [\text{NavState}] \qquad \frac{\begin{array}{l}\Theta\sigma \vdash e \Downarrow \mu \\ \mu_a = \bigcup\{\mu_o \mid o \in \mu, o.\mathbf{get}(a) \Downarrow \mu_o\}\end{array}}{\Theta\sigma \vdash e \,.\, a \Downarrow \mu_a}\ [\text{NavMem}]$$

$$\frac{\begin{array}{l}\Theta\sigma \vdash e \Downarrow \mu \\ \mu_a = \bigcup\{\mu_o \mid o \in \mu, o.\mathbf{get}(a) \Downarrow \mu_o\}\end{array}}{\Theta\sigma \vdash e > a \Downarrow \mu_a}\ [\text{NavRef}] \qquad \frac{\begin{array}{l}\Theta\sigma \vdash e \Downarrow \mu \\ \mu_a = \bigcup\{\mu_o \mid o \in \mu, o.\mathbf{get}(a) \Downarrow \mu_o\}\end{array}}{\Theta\sigma \vdash e < a \Downarrow \mu_a}\ [\text{NavInvRef}]$$

Figure 5.2: Operational semantics for *NavExp* and *TypePath* AST node children.

$$\boxed{\Theta\sigma \vdash \mathit{expression} \,/\, \Sigma \Downarrow \mu \,/\, \Sigma}$$

$$\frac{\mathit{basic}(a) \quad \Sigma(o,a) = \mu_a}{o.\mathbf{get}(a) \,/\, \Sigma \Downarrow \mu_a \,/\, \Sigma}\ [\text{basic} \mid \mathbf{get1}] \qquad \frac{\begin{array}{l}\mathit{reference}(a) \quad \mathit{elementary}(a) \\ \Sigma(o,a) = \{o_a : \mathit{Type}\}\end{array}}{o.\mathbf{get}(a) \,/\, \Sigma \Downarrow \{o_a\} \,/\, \Sigma}\ [\text{calc} \mid \mathbf{get3.1}]$$

$$\frac{\mathit{derived}(a) \quad \Sigma(o,a) = \mu_a}{o.\mathbf{get}(a) \,/\, \Sigma \Downarrow \mu_a \,/\, \Sigma_a}\ [\text{calc} \mid \mathbf{get2.1}]$$

$$\frac{\begin{array}{l}\mathit{derived}(a) \quad \Sigma(o,a) \neq \mu_x \\ \{\}\ o \vdash a.\mathit{Exp} \,/\, \Sigma \Downarrow \mu_a \,/\, \Sigma_a \\ \Sigma_n = \Sigma_a, \{o \mapsto a \mapsto \mu_a\}\end{array}}{o.\mathbf{get}(a) \,/\, \Sigma \Downarrow \mu_a \,/\, \Sigma_n}\ [\text{uncalc} \mid \mathbf{get2.2}]$$

$$\frac{\begin{array}{l}\mathit{reference}(a) \quad \mathit{elementary}(a) \\ \Sigma(o,a) \neq \mu_x \quad \Sigma(o,\_a) = \{o_k : \texttt{text}\} \\ \{\}\ o\ o_k \vdash a.\mathit{RefConstr} \,/\, \Sigma \Downarrow \mu_a \,/\, \Sigma_e \\ \Sigma_2 = \Sigma_e, \{o \mapsto a \mapsto \mu_a\} \\ \Sigma_i = \bigcup_{o_a\in\mu_a}\{o_a \mapsto a_i \mapsto \Sigma_2(o_a,a_i) \cup \{o\} \\ \qquad\qquad\qquad\qquad \mid a_i \in a.\mathit{InvRefs}\} \\ \Sigma_3 = \Sigma_2, \Sigma i\end{array}}{o.\mathbf{get}(a) \,/\, \Sigma \Downarrow \mu_e \,/\, \Sigma_3}\ [\text{uncalc} \mid \mathbf{get3.2}]$$

$$\frac{\begin{array}{l}a : \texttt{collection} \quad \Sigma(o,a) = \mu_a \\ \exists o_e \in \mu_a[\Sigma(o_e, \_\mathsf{key}) = k]\end{array}}{o.\mathbf{getEntry}(a,k) \,/\, \Sigma \Downarrow \{o_e\} \,/\, \Sigma}\ [\text{found} \mid \mathbf{getEntry}] \qquad \frac{\begin{array}{l}a : \texttt{collection} \quad \Sigma(o,a) = \mu_a \\ \nexists o_e \in \mu_a[\Sigma(o_e, \_\mathsf{key}) = k]\end{array}}{o.\mathbf{getEntry}(a,k) \,/\, \Sigma \Downarrow \{\} \,/\, \Sigma}\ [\text{notfound} \mid \mathbf{getEntry}]$$

Figure 5.3: Operational semantics for getters that evaluation rules use.

attribute. Rule [**get2.2**] matches if a derived attribute value is uncalculated. The rule evaluates the expression *Exp* of an attribute holding derived values; e.g. the *NumExp* for an integer attribute. The third premise of the rule constructs a new store that includes the result from the expression evaluation.

Rule [**get3.2**] matches unresolved elementary references (members with a *RefConstr* AST node). For elementary reference attributes, objects holds a special member *_a* storing a text key value. This value should correspond to the key value of an object in a collection. For checking referential integrity, rule [**get3.2**] evaluates the constraint expression *RefConstr*. The rule passes an additional environment o_k to the rule evaluating *RefConstr*; the key of the object. The rule evaluating *RefConstr* uses the key for retrieving an object from a collection with the function **getEntry**. With the result from *RefConstr*, [**get3.2**] constructs a new store, Σ_2. Σ_2 holds a mapping from member a to the expression result. After adding a reference to the store, we also need to update inverse reference attributes on the referenced node. That is, if the result contains such a node. To this end, we construct Σ_i. Σ_i holds

$$\boxed{\Theta\sigma \vdash \mathit{expression} \,/\, \Sigma \Downarrow \mu \,/\, \Sigma}$$

$$\frac{\Theta\sigma \vdash e \Downarrow \{o\}}{\Theta\sigma \vdash \mathit{TextExp}(e) \Downarrow \{o\}}\ \textsf{[TextExp1]} \qquad \frac{\Theta\sigma \vdash e \Downarrow \mu}{\Theta\sigma \vdash \mathit{NumExp}(e) \Downarrow \mu}\ \textsf{[NumExp1,2]}$$

$$\frac{\Theta\sigma \vdash e_1 \Downarrow \{o_1\} \quad \Theta\sigma \vdash e_2 \Downarrow \{o_2\}}{\Theta\sigma \vdash e_1 \oplus e_2 \Downarrow \{\oplus(o_1,o_2)\}}\ \textsf{[NumExp3]} \qquad \frac{\Theta\sigma \vdash e_1 \Downarrow \{o_1\} \quad \Theta\sigma \vdash e_2 \Downarrow \{o_2\}}{\Theta\sigma \vdash \oplus\, e_1\, e_2 \Downarrow \{\oplus(o_1,o_2)\}}\ \textsf{[NumExp4]}$$

$$\frac{\Theta\sigma \vdash e \Downarrow \mu_1}{\Theta\sigma \vdash \uplus\, e \Downarrow \{\uplus(\mu_1)\}}\ \textsf{[NumExp5.1]} \qquad \frac{\begin{array}{l}\Theta\ \sigma \vdash e_1 \Downarrow \mu_1\\ \mu_3 = \bigcup_{o\in\mu_1}\{o_r \mid \{v \mapsto o\}\ \sigma \vdash e_2 \Downarrow \{o_r\}\}\end{array}}{\Theta\sigma \vdash \uplus\, e_1\, v\, e_2 \Downarrow \{\uplus(\mu_3)\}}\ \textsf{[NumExp5.2]}$$

$$\frac{\begin{array}{l}o_w = \{w \mapsto \mu_w \mid w \in W, \Theta\sigma \vdash e_w \Downarrow \mu_w\}\\ o_{new} = \{\textbf{\_state} \mapsto s\} \cup o_w\end{array}}{\Theta\sigma \vdash \mathit{SgExp}(s,W) \Downarrow \{o_{new}\}}\ \textsf{[SgExp1]} \qquad \frac{\begin{array}{l}\Theta\sigma \vdash e \Downarrow \mu_e \quad \mu_e = \{\}\\ \Theta\ \sigma \vdash s_t \Downarrow \mu_s\end{array}}{\Theta\sigma \vdash \texttt{empty}\ e\ \_\ s_t\ \_ \Downarrow \mu_s}\ \textsf{[SgExp2a]}$$

$$\frac{\begin{array}{l}\Theta\ \sigma \vdash e \,/\, \Sigma \Downarrow \mu_e \,/\, \Sigma_e\\ \Sigma_e(\mu_e, \textbf{\_state}) = s \in S\\ \Theta \cup \{v \mapsto \mu_e\}\ \sigma \vdash s.\mathit{Exp} \,/\, \Sigma_e \Downarrow \mu_s \,/\, \Sigma_s\end{array}}{\Theta\sigma \vdash \texttt{switch}\ e\ v\ S \,/\, \Sigma \Downarrow \mu_s \,/\, \Sigma_s}\ \begin{array}{r}\textsf{[TextExp2,}\\ \textsf{NumExp6,}\\ \textsf{SgExp3,}\\ \textsf{RefExp2]}\end{array} \qquad \frac{\begin{array}{l}\Theta\sigma \vdash e \Downarrow \mu_e \quad \mu_e \neq \{\}\\ \Theta \cup \{v \mapsto \mu_e\}\ \sigma \vdash s_f \Downarrow \mu_s\end{array}}{\Theta\sigma \vdash \texttt{empty}\ e\ v\ \_\ s_f \Downarrow \mu_s}\ \textsf{[SgExp2b]}$$

Figure 5.4: Operational semantics for expressions producing derived values.

the new values for inverse references attributes on the referenced node. The last premise of [**get3.2**] constructs a new store combining Σ_2 with Σ_i.

As mentioned above, the function **getEntry** retrieves an object with a specific key from a collection. If an object with a key k exists, the rule evaluates to a bag holding this object. Otherwise, it evaluates to an empty bag.

Derivation expression evaluation. For calculating derived attribute values, getters evaluate expressions of the attributes. Figure 5.4 presents rules for evaluating the expressions. The rules [TextExp1] and [TextExp2] evaluate to the result of the navigation expression e. For [TextExp1], ALANLIGHT's static semantics guarantees evaluation to a bag holding a single object o.

Rules [NumExp3] and [NumExp4] both evaluate subexpressions, combining results with the operator $\oplus$. Again, for the subexpressions ALANLIGHT guarantees evaluation to a bag holding a single object. The operator $\oplus$ calculates a result using mathematical operations corresponding to *NumBinOp* and *NumBinFn*. Rules [NumExp5.1] and [NumExp5.2] apply the operator $\uplus$ to a bag of results. This operator applies the operation corresponding to *NumAggFn* to a bag of `integer` objects. Rule [NumExp5.2] evaluates e_2 per object from the bag μ_1; the result from evaluating e_1. For each evaluation, the rule constructs a variable environment mapping v to a single object o from μ_1. Rules [SgExp2b] and [SgExp3b] also construct new variable environments for their subexpressions. Rule [SgExp3a] matches when the expression e evaluates to an empty set; [SgExp3b] matches the opposite case.

Unlike results from other rules in the figure, the result for rule [SgExp1] is special; it is a new object, a state node. The state node has a member storing its state definition **_state**,

$$\boxed{\Theta\sigma(k) \vdash \mathit{expression} \,/\, \Sigma \Downarrow \mu \,/\, \Sigma}$$

$$\frac{\begin{array}{l}\Theta\sigma \vdash e \Downarrow \mu_e \qquad \Theta_e(o) = \Theta \cup \{v \mapsto o\} \\ f(o) = \bigcup_{w \in w^*} \{w.v \mapsto \mu_w \mid \Theta_e(o)\,\sigma \vdash w \Downarrow \mu_w\} \\ g(o) = \{\texttt{key} \mapsto o\} \cup f(o) \\ \mu_{new} = \bigcup_{o \in \mu_e} \{g(o) \mid \forall v \mapsto \mu \in f(o),\ \mu \neq \{\}\}\end{array}}{\Theta\sigma \vdash \texttt{=}\ e\ \ v\ w^* \Downarrow \mu_{new}}\ \text{[ColExp1.1a]}$$

$$\frac{\begin{array}{l}\Theta\sigma \vdash e \Downarrow \mu_e \\ h(k,i) = \{i \mapsto \mu_i \mid \Theta\sigma k \vdash i \Downarrow \mu_i\} \\ g(o) = \{\texttt{key} \mapsto o\} \cup \{i.v \mapsto h(o,i) \mid i \in i^*\} \\ \mu_{new} = \bigcup_{o \in \mu_e} \{g(o) \mid \exists i \in i^*,\ h(o,i) \neq \{\}\}\end{array}}{\Theta\sigma \vdash \texttt{=}\ e\ v\ i^* \Downarrow \mu_{new}}\ \text{[ColExp1.1b]}$$

$$\frac{\Theta\sigma \vdash e \Downarrow \mu_e}{\Theta\sigma \vdash \texttt{where(}\ e\ \texttt{)}\ v \Downarrow \mu_e}\ \text{[ColExp1.2]}$$

$$\frac{\begin{array}{l}\Theta\sigma \vdash e \Downarrow \mu_e \\ \mu_v = \{o_v \in \mu_e \mid o_v.\mathbf{get}(r) \Downarrow \mu_k,\ k \in \mu_k\}\end{array}}{\Theta\sigma k \vdash \texttt{in(}\ e\ r\ \texttt{)}\ v \Downarrow \mu_v}\ \text{[ColExp1.3]}$$

$$\frac{\Theta\sigma \vdash e \Downarrow \mu_e}{\Theta\sigma \vdash \mathit{RefExp}(e) \Downarrow \mu_e}\ \text{[RefExp1]}$$

Figure 5.5: Operational semantics for expressions producing derived references.

which we described earlier. The state node also holds mappings for all parameters W to their corresponding values.

Figure 5.5 presents the operational semantics for expressions producing derived references. The rule signature indicates an additional optional environment k; an object, which rule [ColExp1.3] uses. Rules [ColExp1.1a] and [ColExp1.1b] construct new objects: derived entities. Rule [ColExp1.1a] interprets collection expressions with `where` clauses; rule [ColExp1.1b] interprets collection expressions with `in` clauses. The first premise of both rules interprets the expression e for selecting a collection, yielding a bag of nodes.

The second line of rule [ColExp1.1a] is a function, evaluating `where` clauses for a specific object o. The third line constructs a new object with a key mapping to o. The new object stores corresponding variable results from `where` clause evaluation. The last premise for rule [ColExp1.1a] constructs a new collection, holding a new object for each object in μ_e that satisfies all `where` clauses. Rule [ColExp1.2] evaluates `where` clauses, calling a rule for evaluating the navigation expression e and returning its result. Note that the rule ignores v. Rule [ColExp1.1a] retrieves v from w when calling the function f.

To explain rule [ColExp1.1b], we first explain rule [ColExp1.3], which evaluates separate `in` clauses. An `in` clause consists of two parts: a navigation expression e and a reference or inverse reference navigation step r. The navigation expression e produces a bag of objects μ_e, as the first premise of the rule expresses. The last premise filters the bag: μ_v holds objects for which the result from evaluating r holds a key object k.

The key object k is object for which [ColExp1.1b] creates a new object in a derived collection. That is, if and only if at least one `in` clause produces a bag holding an object: a node object which references k. The second premise of [ColExp1.1b] defines a function evaluating a single `in` clause for a specific object k. The third premise defines a function for constructing a new object, which holds a mapping from each `in` clause variable name v to a corresponding bag of objects. The last premise of [ColExp1.1b] construct the actual collection. The collection holds an object for each result from e where at least one `in` clause produces a non-empty bag of objects.

$$\boxed{\Theta\sigma k \vdash \mathit{expression} \,/\, \Sigma \Downarrow \mu \,/\, \Sigma}$$

$$\frac{\Theta\sigma \vdash e \Downarrow \{\}}{\Theta\sigma k \vdash \sim\!\texttt{ref}(e\;.a\,[\,G\,])) \Downarrow \{\}}\;[\text{RefConstr1a}] \qquad \frac{\Theta\sigma \vdash e \Downarrow \{\}}{\Theta\sigma k \vdash \sim\!\texttt{ref}(\,e.\;a\,) \Downarrow \{\}}\;[\text{RefConstr2a}]$$

$$\frac{\Theta\sigma \vdash e \Downarrow \{o\} \qquad o.\mathbf{getEntry}(a,k) \Downarrow \{\}}{\Theta\sigma k \vdash \sim\!\texttt{ref}(e\;.a\,[\,G\,])) \Downarrow \{\}}\;[\text{RefConstr1b}] \qquad \frac{\Theta\sigma \vdash e \Downarrow \{o\} \qquad o.\mathbf{getEntry}(a,k) \Downarrow \mu}{\Theta\sigma k \vdash \sim\!\texttt{ref}(\,e.\;a\,) \Downarrow \mu}\;[\text{RefConstr2b}]$$

$$\frac{\Theta\sigma \vdash e \Downarrow \{o_e\} \qquad \Theta\sigma \vdash e.\texttt{parent} \Downarrow \{o\} \qquad o.\mathbf{getEntry}(a,k) \Downarrow \{o_k\} \qquad o.\mathbf{addEdge}(\langle o_e,o_k\rangle,G) \Downarrow \mu}{\Theta\sigma k \vdash \sim\!\texttt{ref}(e\;.\texttt{parent}\;.a\,[\,G\,])) \Downarrow \mu}\;[\text{RefConstr1c}]$$

$$\frac{\Sigma_G = \Sigma, \bigcup_{g\in G}\{o \mapsto g \mapsto \Sigma(o,g) \cup \{\langle o_s,o_r\rangle\}\} \qquad \mu = \begin{cases}\{o_r\} & \text{if } \forall g \in G[\dagger(\Sigma_G(o,g))]\\ \{\} & \text{else}\end{cases}}{o.\mathbf{addEdge}(\langle o_s,o_r\rangle,G) \,/\, \Sigma \Downarrow \mu \,/\, \Sigma_G}\;[\mathbf{addEdge}]$$

Figure 5.6: Operational semantics for constraint expressions.

The remaining rule [RefExp1] evaluates an expression e for deriving a reference. In the next paragraph we discuss constraints and graph construction.

Constraint expression evaluation. ALANLIGHT's operational semantics for evaluating constraints consists of six rules (Figure 5.6). The first two rules match reference constraint expressions where the expression e evaluates to an empty bag. In such cases, the rule also yields an empty bag, meaning that the constraint is not satisfied.

Rule [RefConstr1b] matches self reference constraint expressions where the entry with key k does not exist. This rule also evaluates to an empty bag. Rule [RefConstr2b] matches other reference constraint expressions, yielding a bag holding zero or one item.

Rule [RefConstr1c] matches self reference constraint expressions where the entry with key k exists. If it exists, the rule adds an edge to all graphs G that the newly resolved reference partakes in. To this end, the rule calls the function **addEdge** on object o, which stores special graph members together with the collection member. Note that the we use o_e for the new edge; o_e is an entry from collection attribute a on o (and an ancestor of the referencing object σ, or otherwise σ itself). We use o_e because we only need to ensure acyclicity for entries from the collection a on object o.

Rule [**addEdge**] constructs a new store holding graphs containing a new edge. The function uses the operator † to check if all graphs in the new store are acyclic. If they are acyclic, the rule evaluates to a bag holding the referenced node; otherwise, it evaluates to an empty bag. A non-empty bag means satisfaction of all graph constraints G.

For checking constraints on initialization, we visit all elementary reference members once and execute the function **get** for the member. If the function yields an empty result, we reject the dataset: it is not conform the model. After processing updates, we do this again, rejecting the update when finding an empty result.

5.2 Complexity bounds

In this section, we discuss bounds on the number of calculations and the time complexity of ALANLIGHT programs. ALANLIGHT programs perform each calculation exactly once during program execution. Program execution means evaluating (all) expressions for any store Σ, storing elementary (user) data. The number of calculations for ALANLIGHT programs is bounded in the size of the specification and the size of the input data. The time complexity of ALANLIGHT programs is polynomially bounded (PTIME) in the size of the input data.

Calculations. According to our operational semantics, member values are calculated exactly once when checking constraints and calculating derived values. The evaluation rules ensure this: if a value is calculated, the getters retrieve it from the store. If an attribute value is uncalculated, we evaluate the expression for calculating it, storing the resulting value. An upper bound on the number of calculations of an ALANLIGHT program is (using *big-O notation* [65]):

$$O(\#Calcs) = O(m \cdot n^d) \tag{5.1}$$

where d is the maximum depth of a model hierarchy when taking only collection attributes into account, and m is the total number of attribute and key members. Note that $d \leq m$, but we distinguish d from m as the typical depth of a model hierarchy is significantly smaller than m. n is the maximum size of a collection in the input; n is at least one (for the root node). Below we derive this upper bound.

A data store consists of hierarchically organized nodes. The root level consists of a exactly one root node. Let m_0 be the number of members of the root node (level 0), and m_l the total number of members on level l, where $0 \leq l \leq d$. A member corresponds to a calculation, meaning that stores consisting of only a root node require at most m_0 calculations.

An expression for deriving a value yields at most a collection of size n; the size of largest input collection. Therefore, the root node holds at most m_0 collections of size n. The items of different collections $c_0 \in m_0$ are of different types; each collection attribute defines the *Type* of its items.

Let t_{c_0} be the number of members of such a *Type*; specifically, the number of members on the next hierarchical level ($l = 1$). Thus, each item i in a collection c_0 requires t_{c_0} calculations, not counting calculations for items in collections that i nests (i.e. only calculations on level $l = 1$). As m_1 is the total number of members on level $l = 1$, we know that:

$$\sum_{c_0 \in m_0} t_{c_0} = m_1$$

Because an item requires at most t_{c_0} calculations, a single collection c_0 on the root level requires at most $t_{c_0} \cdot n$ calculations for level $l = 1$. As the root level holds at most m_0 of these collections, the number of calculations for hierarchical level $l = 1$ is bounded above by:

$$\sum_{c_0 \in m_0} t_{c_0} \cdot n = m_1 \cdot n$$

ALANLIGHT supports manually repeating derived collection attributes, recursively. Every nested collection can store all n nodes from the largest input collection, yielding at most n^d *equally typed* nodes on the d-th level. Consequently, the total maximum number of calculations for the d-th level is:

$$\sum_{c_0 \in m_0} \sum_{c_1 \in t_{c_0}} \cdots \sum_{c_{d-1} \in t_{c_{d-2}}} t_{c_{d-1}} \cdot n^d = m_d \cdot n^d$$

Combining the total maximum number of calculations per level in a data store, we get the aforementioned upper bound on the number of calculations for any given ALANLIGHT data store:

$$O(m_0 + m_1 \cdot n + m_2 \cdot n^2 + \ldots + m_d \cdot n^d) = O(m \cdot n^d)$$

Note that the total number of members $m = m_0 + m_1 + \ldots + m_d$. We can statically determine m and d in trivial manner: counting members and the model depth; the height and the width of an ALANLIGHT model, respectively. We conclude that the dynamic number of calculations of every ALANLIGHT program is bounded above by a d-th order polynomial in the size of the input:

$$O(\#Calcs_{runtime}) = O(n^d) \tag{5.2}$$

where d is the constant, design-time calculated model-depth. Using this upper bound and ALANLIGHT's operational semantics, we can determine an upper bound on the time complexity of ALANLIGHT programs. Specifically, we focus on an upper bound on the time complexity for expressions calculating derived values in an ALAN program.

Time complexity. Calculating a derived value requires evaluating expressions. Expressions for deriving values consist of navigation expressions which ultimately determine the cost for deriving a value. An ALANLIGHT model defines the number of navigation expressions (subexpressions) needed for calculating a derived value.

Navigation expressions consist of navigation steps. Text attribute, integer attribute, natural attribute, state group attribute, state, and key steps yield (a bag holding) zero or one value. The same holds for `this` steps and `parent` steps. Collection navigation steps yield n nodes (where n is maximum collection size that we mentioned above). Inverse reference attributes store inverse references to nodes of a specific *Type* in a model. The maximum number of nodes of a specific type is n^d. Therefore, a subsequent navigation step multiplies a previous result by at most n^d values.

If every navigation step produces at most n^d values, then every fully expanded navigation expression produces at most $(n^d)^e = n^{de}$ values, where e is the maximum number of navigation steps of a fully expanded navigation expression. (Recall that fully expanded means that variables are replaced by their corresponding expressions.) For example, suppose that we have a model with only the following expression from Chapter 3:

```
latest_version_cost: integer
= switch (this>latest_version.product_type) as pt (   // e = 3
| simple    = pt.cost                                  // e = 3 + 1 (inline pt)
| assembled = sum(pt.parts>key.latest_version_cost) ) // e = 3 + 3 (inline pt)
```

For the expression, the maximum e-value is $3+3=6$. That is, the e-value for the fully expanded navigation expression for the `assembled` state. Note that for producing n^{de} values, a fully expanded navigation expression has to consist of e inverse reference attribute navigation steps. As every navigation expression starts with a `this` navigation step and optional `parent` navigation steps, the actual maximum number of values is strictly less than n^{de}. Furthermore, ALAN models typically contain at most one inverse reference step (and also, inverse reference steps occur infrequently).

From the bound on the number of values that navigation expressions produce, we can derive an upper bound on their time complexity. Assuming that the retrieval of a single value from memory requires constant time, the first navigation step requires $O(n^d)$ time. For subsequent navigation steps $2..e$, we have to multiply this by the maximum number of values from the previous result. This gives the following upper bound on the time complexity of a navigation expression:

$$T_{NavExp} = O(n^d + (n^d)^2 + ... + (n^d)^e) = O((n^d)^e) = O(n^{de}) \tag{5.3}$$

An expression for deriving a value consists of at most s subexpressions (which is 3 for the `latest_version_cost` above). Taking the subexpressions into account, we get the following upper bound on the time complexity of navigation expressions:

$$T_{NavExps \in Exp} = O(s \cdot n^{de}) \tag{5.4}$$

This is also an upper bound on the time complexity of an expression for calculating a derived value, such as an integer. We explain this next.

Separate evaluation rules for deriving values (Figure 5.4) and references (Figure 5.5), either produce (at most) n^{de} values (the upper bound on the number of values for a navigation expression), or a single value by value aggregation (e.g. with `sum`). The rules [TextExp1], [NumExp1], [RefExp1], and [ColExp1.2] evaluate a single navigation expression. Therefore, corresponding expressions require $O(n^{de})$ time, as shown above.

The remaining rules from Figure 5.4 and [ColExp1.3] evaluate one or more subexpressions (which can be navigation expressions), and perform operations on these values (except for rule [NumExp5.2] which we discuss separately). The rules perform either (1) a fixed number of basic operations on a single value per subexpression they nest, or (2) a basic operation on at most n^{de} values (which is the upper bound mentioned above). To this end, the rules use the operators $\oplus$ and $\uplus$; for example, to multiply or sum values. Alternatively, the rules use basic equality checks; for instance, for checking if a bag is empty. Because of this, the maximum number of values for which the rules perform an operation, is bounded above by:

$$Values_{op} = O(s + n^{de}) = O(n^{de}) \tag{5.5}$$

for large enough n, and $d, e \geq 1$.

Assuming that the operators $\oplus$ and $\uplus$ require constant time per value, the rules require $O(n^{de})$ time for operations such as `sum`. The maximum number of subexpressions for which we have to perform such operations, is s. If we also include s for the full expression for deriving a value, we get the following upper bound on the time complexity of expressions deriving a value:

$$T_{Exp} = O(s \cdot n^{de}) \tag{5.6}$$

This bound is identical to the presented complexity bound for navigation expressions.

Unlike other rules from Figure 5.4, rule [NumExp5.2] evaluates a subexpression e_1, and subsequently evaluates another subexpression e_2 for each value that e_1 produces. The following figure shows an expression matching [NumExp5.2]:

```
sum(this.products as p mapto(p.amount * p.cost))
```

Subexpression `this.products` matches e_1; the subexpression after `mapto` matches e_2. A useful property of e_2 is that – conform to our static semantics (Chapter 4) – the expression only uses the variable p (where p is a single `product` value in e_2). Furthermore, `mapto` guarantees a single output value per input value (a `products` value). Therefore, if we omit `* p.cost`, the resulting expression is equivalent to:

```
sum(this.products.amount)
```

For this expression we derived a time complexity of $O(n^{de})$. Note that the `mapto` construct essentially just extends a navigation expression, supporting (recursively nested) numerical operations beside common navigation steps with the dot- and $>$-operator. An important difference with regard to the time complexity is that `mapto` supports multiple subexpressions s_{mapto}. Because of this, the time complexity of [NumExp5.2] is $O(s_{mapto} \cdot n^{de})$. For instance, for the above figure expressing the product of `amount` and `cost`, we have:

$$O(n^{de_p} * (n^{d(e-e_p)} + n^{d(e-e_p)}) = O(2 \cdot n^{de_p} * n^{de-de_p}) = O(s_{mapto} \cdot n^{de})$$

Variable e_p corresponds to the length of e_1 (the expression `this.products`); we multiply n^{de_p} by the complexity of e_2 as we evaluate e_2 n^{de_p} times. Because we map n^{de_p} on a per value basis, p corresponds to one value in e_2. Therefore, a single evaluation of e_2 requires $O(n^{d(e-e_p)})$ time. By applying some basic rewrite rules, we get the above-mentioned upper bound.

However, the total number of subexpressions s for calculating a derived value already includes s_{mapto}. Consequently, the bound $O(s \cdot n^{de})$ also holds for expressions defining a `mapto` operation.

Figure 5.5 presents two rules that we have not discussed so far: the main rules for deriving collections. The rules behave in similar manner: they both evaluate a navigation expression e, which produces at most n objects from one specific collection. A derived collection holds at most one new object for each of these (n) objects; the n objects are the potential candidates for becoming keys of the new objects. For determining the objects for which to create a corresponding object in a derived collection, the rules evaluate `where` clauses and `in` clauses per potential key object. Above, we showed that evaluating separate clauses requires $O(n^{de})$ time. Therefore, evaluating all clauses for a potential key object requires $O(s \cdot n^{de})$ time. Consequently, the two rules require $O(n \cdot s \cdot n^{de})$ time for calculating a derived collection; an additional n for number of potential key candidate objects. However, this bound does not affect the upper bound on the time complexity for calculations that an ALANLIGHT program performs; we show this next.

For deriving an upper bound on the time complexity for calculations that an ALAN-LIGHT program performs, recall that we showed a bound of $O(m \cdot n^{d})$ on the number of calculations. The variable m represents the total count of all members: attributes plus key members (one per collection). Let k be the total number of key members; we know that $k <= m$. Furthermore, let a be the total number of attributes; thus $m = a + k$. Combining this with the upper bound on the number of calculations, we get $O(k \cdot n^{d})$: an upper bound on the number of keys. As deriving a single key value requires $O(s \cdot n^{de})$ time, an upper bound on the time complexity for all expressions determining key values is

$$O(k \cdot n^{d} \ \cdot \ s \cdot n^{de}) \tag{5.7}$$

For collections, we now only need to consider the expression determining key candidates (the above bound includes the expressions for key candidates). Therefore, we get a similar upper bound for attributes, as calculating a derived attribute value requires $O(s \cdot n^{de})$ time:

$$O(a \cdot n^{d} \ \cdot \ s \cdot n^{de}) \tag{5.8}$$

Combining these two bounds gives the following upper bound on the time complexity for calculations that ALAN programs perform:

$$\begin{aligned} T_{Calcs} &= O(a \cdot n^{d} \cdot s \cdot n^{de} \ + \ k \cdot n^{d} \cdot s \cdot n^{de}) \\ &= O((a+k) \cdot n^{d} \cdot s \cdot n^{de}) \\ &= O(m \cdot n^{d} \cdot s \cdot n^{de}) \\ &= O(m \cdot s \cdot n^{d(e+1)}) \end{aligned} \tag{5.9}$$

Again, we can statically determine m and d by counting members and the model depth; the height and the width of an ALANLIGHT model, respectively. We can also statically determine s and e by counting subexpressions and navigation steps, respectively. This gives us the following PTIME bound for ALANLIGHT programs:

$$T(n) = O(n^{c}) \tag{5.10}$$

where c is a design-time constant, calculated from d and e.

5.3 Discussion

Note that for actual ALANLIGHT models, we can automate the computation of tighter bounds using information about attribute types. Specifically, we can use the information that collection attribute navigation steps multiply a previous result by at most n, inverse reference attributes multiply a previous result by at most n^d, and other navigation steps multiply a previous result by a constant value of 1.

With regard to the dynamic semantics, note that ALANLIGHT does not commit to a specific calculation strategy for derived values. Harkes et al. [22, 23] discuss different strategies for calculating and maintaining derived values. This includes on-read, on-write, eventual, incremental calculation, including mixing of different calculation strategies. Their research focuses on the language IceDust, but we can apply these strategies to ALANLIGHT as well. In addition to strategies from their research, ALANLIGHT also supports deterministic top-down calculation of derived values. But, we consider evaluating calculation strategies beyond the scope of this thesis. In the next chapter we evaluate ALANLIGHT. We compare it against other approaches, focusing on the main contributions.

Chapter 6

Evaluation

In our problem statement (Chapter 2), we presented the following research question: *to what extent can we support complex recursive calculations, while guaranteeing soundness, functional correctness, and bounded, predictable running times?* In previous chapters, we showed how ALANLIGHT achieves the guarantees from this question. Specifically, we demonstrated that ALANLIGHT guarantees:

- *soundness*, where values are always of a *predefined indivisible type*,
- *functional correctness* (which includes deterministic output),
- *polynomial time complexity* in the *size* of user data, and
- a *combined complexity* in the *size* of (trivially countable properties of) the specification.

To our knowledge, no existing data modeling language simultaneously (1) guarantees the above properties, (2) uses explicit acyclicity constraints (graph constraints) from models in their type system as a condition for enabling complex recursive calculations, (3) achieves the same level of expressiveness as ALANLIGHT, and (4) integrates expressing calculations.

In this chapter, we evaluate ALANLIGHT's support for complex recursive calculations. To this end, Section 6.1 addresses problems and solutions involving recursive calculations, found in related work. We present modeling problems requiring complex calculations and discuss how existing approaches to solving these problems compare to ALANLIGHT. Section 6.2 discusses potential threats to the validity of our approach.

6.1 Expressiveness and Guarantees

Data modeling and query languages typically make a trade-off between expressiveness and guarantees such as termination, specific running time bounds, soundness, functional correctness, and simplicity. For example, (safe, stratified) DATALOG has a simple syntax, a well-understood semantics, and queries can be evaluated in polynomial time [18], but the language cannot express (recursive) aggregation.

Methodology. The goal of ALANLIGHT is to support expressing complex (recursive) calculations, while providing aforementioned guarantees. ALANLIGHT adds value if the language is either more expressive or provides more guarantees than existing languages. The question is: *to what extent does* ALANLIGHT *succeed in going beyond the capabilities of existing languages?*

To address this question, we compare ALANLIGHT to existing languages that restrict expressiveness, for providing guarantees such as termination. Because of ALANLIGHT's goals, we direct our comparison towards languages that at least have rudimentary support for recursion and address boundedness. We start with more restrictive languages, continuing towards languages providing more expressive power or different guarantees.

We focus on the *concepts* of ALANLIGHT which form the basis for its semantics (Chapter 4 and 5): graph constraints and reference constraints. Therefore, extensions to ALANLIGHT are fine, if (and only if) they do not undermine the fundamental concepts constituting the language's semantics.

ALANLIGHT is a data modeling language with integrated support for modeling constraints and derivations in a declarative manner. But, many approaches do not have such integrated support. Furthermore, ALANLIGHT strictly separates models from conforming data, while other approaches do not make this separation [31, 18, 38]. More similar approaches allow for a more detailed comparison. As such, we discuss more similar approaches in more detail than other (less similar) approaches.

To discuss expressiveness and guarantees of data modeling languages and query languages, we start by introducing a typical problem involving recursion in the field of manufacturing: the bill of materials. This problem typically involves calculating all (basic) parts for a product that recursively consists of multiple parts. The problem is identical to the more general *all pairs reachability* problem (see also: *summarized explosion query*). Calculating all parts of a product is equivalent to calculating the *transitive closure* of a graph.

Basic relational recursion. SQL features an extensive set of operators and other language constructs for performing complex calculations [40]. But, originally, SQL did not support recursive queries, and could thus not express recursive transitive closure computations. The SQL99 standard introduced support for recursive queries with *common table expressions* (CTEs) [53]. Nowadays, most major database systems support this, including PostgreSQL [44] and Microsoft SQL Server [25]. Other database systems have implemented different (proprietary) solutions for supporting recursion. For example, Oracle Database supports recursive SQL queries with `CONNECT BY` statements [45]. The following CTE calculates the transitive closure of a product graph:

```
WITH RECURSIVE transitive_closure(product_id, part_id) AS
( SELECT product_id, part_id FROM products_parts // Anchor
    UNION ALL
  SELECT tc.product_id, i.part_id                // Recursive Member
  FROM products_parts AS i JOIN transitive_closure AS tc
    ON i.product_id = tc.part_id
)
SELECT product_id, part_id FROM transitive_closure
GROUP BY product_id, part_id;
```

A recursive CTE (transitive_closure in the figure) starts with one or more *anchors*; the first SELECT statement in the figure. The anchor from the figure produces all rows from the product_parts table with columns (product_id,part_id), where each row describes a reference from a product to a part. The statement UNION ALL combines the result from the anchor with results from subsequently defined recursive members of the CTE.

The anchor provides the input to the first recursive call. The second SELECT statement uses the result from the anchor (with JOIN transitive_closure) for a JOIN. The second statement should be interpreted as:

```
select all (x=TC.product_id,z=I.part_id)
  given (x,y=TC.part_id) and (y=I.product_id,z)
```

The result of the statement is passed as input to the recursive member for the next iteration. The recursion ends when the select statement no longer produces (new) rows. Oracle SQL supports expressing the query using a CONNECT BY statement, as follows:

```
SELECT product_id, part_id
FROM products_parts
CONNECT BY PRIOR part_id = product_id;
```

We ran the query on five known SQL implementations, including PostgreSQL 9.6, Microsoft SQL Server 2014, Oracle DB 11gR2, MySQL 8.0 [47], and SQLLite 3.18 [51]. With acyclic data, the implementations produced expected output.

However, with *cyclic* data, all five implementations yielded an error or exception, such as 'out of bounds' or 'timeout'. To prevent that, developers can manually limit the number of iterations [2]. But, such limits can elicit incomplete results (when calculations halt because of them). Alternatively, developers can create database triggers or write custom code ensuring acyclic relations before accepting database transactions. But, this places the burden of checking and ensuring termination and functional correctness on the developer, while ALANLIGHT guarantees it.

Stratification and fixpoint recursion. Although the transitive closure query fails on the SQL implementations mentioned above, guaranteeing termination and even correct output for cyclic graphs is entirely possible. We explained this in Chapter 2: the query will – at some point – stop generating new rows: the output converges to a fixpoint. Note that this requires filtering out duplicate rows; that is, a set semantics (rather than a bag semantics, where duplicates may occur). Using a stratified fixpoint semantics, we can guarantee that the transitive closure query terminates and produces correct output.

Aranda et al. [2] made similar observations, leading to the development of R-SQL [2, 58]. R-SQL extends SQL, implementing ideas from the field of deductive database systems, including stratification and fixpoint computation. The basic idea is to restrict the expressions of a program in such a way that a fixpoint always exists; the restriction being that the program (set of query expressions or logical predicates) is stratifiable.

A stratification of a program is an *ordered partitioning* of its expressions. Stratified negation and stratified aggregation determine the conditions for this ordered partitioning, as Green et al. [16, 18] define. Negation stratified programs do not contain recursion through negation; aggregation stratified programs do not contain recursion through aggregation. To explain this, consider the following recursive DATALOG program for counting the *distinct* parts of a product:

```
cpart(X,Y)                   <- part(X,Y)           //r1 ; stratum S1
cpart(X,Y)                   <- cpart(X,Z),part(Z,Y) //r2 ; stratum S1
numDistinctParts(X,count<X>) <- cpart(X,_)           //r3 ; stratum S2
```

This program is aggregation stratified. Strata S1 and S2 indicate a possible stratification (an ordered partitioning) of the clauses. Clauses r1 and r2 are part of the first stratum: S1; clause r3 is part of a second stratum: S2. Clause r3 cannot occur in stratum S1. This is because stratification requires that predicates occurring in the right-hand side of an aggregate rule are strictly defined in an earlier stratum. Thus, that the cpart predicate is defined in an earlier stratum (S1) than numParts (defined in stratum S2).

A stratified fixpoint semantics ensures termination for transitive closure computations. Specifically: a stratified fixpoint semantics in which the algorithm terminates when it is unable to produce new rows or tuples. To explain this, suppose that a manufacturer has products. When releasing a new version of a product, he registers this new version on the old product. Sometimes, after the release of a product, it appears to be unreliable and is marked as such. Customers can put in a request for a free replacement if a product fails. When the manufacturer deems the product unreliable, the manufacturer sends the first reliable version as a replacement. If a reliable version does not exist, he sends the latest version as a replacement. The following ALANLIGHT model expresses the replacement calculation:

```
products: collection ~ g_versions = acyclic-graph {
  new_version: stategroup (
    no  { }
    yes {
      successor: ~ ref(this.parent.parent.persons[g_versions])
    }
  )
  is_unreliable: stategroup (
    yes {
      replacement: = switch (this.parent.new_version) as nv (
        | no  = this.parent
        | yes = switch (nv>successor.is_unreliable) as ur (
          | no  = nv>successor
          | yes = ur>replacement
        )
      )
    }
    no {}
  )
}
```

Suppose now that the manufacturer just registered ten products in his brand-new system. All ten products are different versions of a `DustBuster` product; we assume that they form a linked list. Only one of these `DustBusters` is deemed unreliable: `DustBusterV1`; the oldest one, as it has a bad battery. For calculating the `replacement` for `DustBusterV1`, a fixpoint algorithm will iteratively attempt to find a possible replacement for `DustBusterV1`. For example, in the first iteration it traverses the `successor` relation for `DustBusterV1`. Subsequently, it traverses the `successor` relation for `DustBusterV2`, and so on.

At every iteration, a fixpoint algorithm checks if a fixpoint is reached: it essentially checks if it already traversed the path from one `DustBuster` to another. For this purpose, the algorithm stores information about each traversed relation (generated tuple or row). The check ensures termination in the presence of cycles. Different fixpoint computation strategies exist (e.g. top-down, starting at the target and bottom-up, starting at the known – user provided – references). But, the basic idea is the same; the algorithms check at every iteration if a fixpoint is reached.

As a natural consequence of its semantics, ALANLIGHT does not require such checks. Note that the checks are redundant when relations are acyclic, which may negatively affect efficiency. But, we consider investigating this beyond the scope of this thesis; we merely establish that ALANLIGHT uses a more elegant approach. Instead of performing checks during computations, ALANLIGHT requires acyclic graph traversal. For ensuring acyclic graph traversal, ALANLIGHT checks graph constraints before accepting changes to user data. (Note that graph constraints often also serve to ensure data integrity, and are thus not solely meant for the purpose of enabling specific recursive calculations.)

Because ALANLIGHT requires acyclic graph traversal, the language precludes computations that recursively traverse potentially cyclic relations. For example, ALANLIGHT does not support computing the shortest path between nodes in a cyclic graph. A typical fixpoint semantics supports such computations, but imposes specific restrictions on operations for ensuring boundedness. Furthermore, graph constraints are often essential for ensuring correct output. The next paragraph discusses this in more detail.

Monotonic aggregation and arithmetic. Although a stratified fixpoint computation for the transitive closure query terminates, it fails when introducing even basic arithmetic. For example, the bill-of-material problem typically involves calculating required quantities per part of a product. For expressing this, we introduce the $*$ operator, and use it in the recursive member of the transitive closure query:

```
WITH RECURSIVE transitive_closure_q(product_id, part_id, quantity) AS
( SELECT product_id, part_id, quantity FROM products_parts
    UNION ALL // combine with:
  SELECT tc.product_id, i.part_id, tc.quantity * i.quantity
  FROM products_parts AS i JOIN transitive_closure_q AS tc
  ON i.product_id = tc.part_id
)
SELECT product_id, part_id, SUM(quantity)
FROM transitive_closure_q
GROUP BY product_id, part_id;
```

Each recursive call potentially produces new rows with a different quantity. The calculated quantity is an infinitely increasing number in the presence of a cycle (if the quantity is greater than zero). Therefore, a fixpoint computation for this query does not necessarily terminate.

The field of deductive databases proposes several solutions to this problem [18]. However, the community seems not to have converged on any specific proposal. One of the proposals is *monotonic aggregation*, presented by Ross and Sagiv [57]. Monotonic aggregation has received much attention in the field of deductive databases [11, 18, 38, 57]. Several concrete implementations exist, including DATALOG^{FS} [38], DEALS [63], and BIGDATALOG [62]. The semantics of monotonic aggregates in the latter two is based on the semantics of DATALOG^{FS}.

The following program from Mazuran et al. [38] expresses the quantity calculation in DATALOG^{FS}:

```
cassb(Prod, Part) : Qty  <- prodpart(Prod, Part, Qty).           // r1
need(Part, Part) : 1     <- prodpart(_, Part, _).                // r2
need(Prod, Part) : K     <- K:[cassb(Prod, P1), need(P1, Part)].  // r3
total(Prod, Part, K)     <- K =![need(Prod, Part)].               // r4
```

The program uses the three extensions to DATALOG that DATALOG^{FS} introduces: multi-occurring predicates (`r1`), running-FS goals (`r3`), and final-FS goals (`r4`). The basic idea behind the extensions is to add a frequency (a counter) to clauses, and to sum (or count) frequencies, producing a result equivalent to that of an aggregate such as `sum` in recursion.

Rule `r1` indicates that a product-part relation with a quantity `Qty` gives a predicate `cassb` with a frequency support of `Qty`. Rule `r2` and `r3` constitute the recursive part of the quantity calculation. Rule `r2` ends the recursion. It indicates that a part needing itself counts as 1 towards the final result. The body of rule `r3` contains a running-FS goal; the bracketed part is called a b-expression. At every recursive iteration of `need`, the frequency of `cassb` is multiplied by the frequency of a matching `need` predicate, yielding a `need` tuple with a new `K`-value.

Mazuran et al. define the formal semantics of a running-FS goal by rewriting it into a negation-free logic program. The basic idea behind the semantics is that a b-expression can be rewritten as follows:

```
cassb(Prod,P1),cassb(P1,P2),cassb(P2,P3),..,cassb(PX,PN), P1≠P2≠P3..≠PN
```

This rewriting indicates that a `cassb` tuple is never used twice when evaluating the running-FS goal. The maximum recursion depth equals the cardinality of the set of `cassb` instances, and the recursive `need` query always terminates.

Note that every `prodpart` item initially produces a `need` tuple with a frequency support of 1. Thus, rule `r3` in fact generates all possible `need` tuples with a frequency of 1 up to the maximum value for `K` for which the running-FS goal holds. The maximum value for `K` corresponds to the result we require; the final-FS goal from rule `r4` is needed for retrieving it. Its semantics is defined as follows:

```
total(Prod, Part, K)      <- K :[need(Prod, Part)], ¬morethan(Prod,Part,K).
morethan(Prod,Part,N)     <- N1:[need(Prod, Part)], N1 > N.
```

Thus, `r4` retrieves the `need` tuple with the highest frequency K for every distinct product-part pair. The expression ¬`morethan` ensures that K is indeed the highest frequency. To this end, `morethan` checks K against the frequencies of other `need` tuples for identical product-part pairs.

This DATALOG^{FS} program for the quantity calculation poses several concerns, three of which we address below. First, in the presence of a cycle, the program produces finite `total` quantities. From a logical (DATALOG's least-fixpoint semantics') perspective, this result is 'correct' output to the program. But, from a functional point of view it is incorrect; the correct solution to the problem is an infinite number. Note that in this particular case, a cycle should not be allowed to exist in the first place.

Second, programs such as the above, use complex combinations of optimization techniques needed for achieving efficient evaluation; for example, magic set optimization, copy rule optimization, max-based optimization, differential fixpoint optimization, Monotonic Aggregate Semi-naive Evaluation (MASN), Eager Monotonic Aggregate Semi-naive evaluation (EMSN), and Constraint Pushing into Recursion (CPR) [38, 63, 75]. But, different optimizations apply to different kinds of calculations, and can lead to erroneous behaviour of other calculations, or otherwise require very specific restrictions in the compiler. For example, suppose that we use the max-based optimization for evaluating a running-FS goal. Then, for querying product-part pairs where products require 3 or more instances of the part, we can use: `K:[need(Prod,Part)], K >= 3`. However, for querying product-part pairs where products require less than 3 instances of the part, we cannot use the natural counterpart: `K:[need(Prod,Part)], K < 3`. This running-FS goal produces every product-part pair instead of the ones with at most two instances of a part. This is because of the semantics of the running-FS goal: a `need` tuple is generated for frequencies 1 up to `K` for all `prodpart` tuples.

In contrast, ALANLIGHT imposes no specific restrictions on operators. More importantly: ALANLIGHT guarantees correct results and in addition has a simple yet efficient evaluation strategy. Efficient, as ALANLIGHT only calculates values that lead to the final result; that is, on-demand and on a per-attribute-instance basis.

So far, we have shown how to express a bill of material calculation (with quantities) in other languages, highlighting the main differences between these languages and ALANLIGHT. The following model expresses the bill of material calculation with quantities in ALANLIGHT:

```
products: collection ~ g_parts = acyclic-graph {
  parts: collection ~ ref(this.parent.products[g_parts]) {
    quantity: integer
    sub_parts: collection = this.parent.parent.products
      in(this>key.parts>key) = direct
      in(this>key.parts.sub_parts>key) = indirect
    {
      quantity: integer = this.parent.quantity *
        (sum(direct.quantity) + sum(indirect.quantity))
    }
  }
  all_parts: collection = this.parent.products
    in(this.parts>key) = direct
    in(this.parts.sub_parts>key) = indirect
  {
    quantity: integer = sum(direct.quantity) + sum(indirect.quantity)
  }
}
```

The sub_parts calculation recursively aggregates sub_parts of parts, and calculates the quantity per subpart. Subsequently, the all_parts calculation combines the results. Note that with some minor modifications to ALANLIGHT we can omit the sub_parts collection, making the expression more concise. For example, we can track the parts items leading to a specific indirect part, enabling us to express the calculation as follows:

```
products: collection ~ g_parts = acyclic-graph {
  parts: collection ~ ref(this.parent.products[g_parts]) {
    quantity: integer
  }
  all_parts: collection = this.parent.products
   in(this.parts>key) = direct
   in(this.parts as part in part>key.all_parts>key) = (indirect,part) //tuple
  {
    quantity: integer = sum(direct.quantity)
      + sum(part.quantity) * sum(indirect.quantity)
  }
}
```

Language extensions such as these do not affect the core concepts constituting the (static) semantics of ALANLIGHT. The same holds for language extensions adding support for features such as *mutual recursion*. For supporting mutual recursion, we can simply inline expressions before running our static semantics checker [26]. The next paragraph uses some basic extensions for further discussing other important properties of ALANLIGHT.

```
module lms (incremental)
entity Student {
  name     : String
}
entity Assignment (eventual) {
  name     : String
  question : String?
  deadline : Datetime?
  minimum  : Float
  avgGrade : Float?    = avg(submissions.grade)
  passPerc : Float?    = count(submissions.filter(x=>x.pass)) / count(submissions)
}
entity Submission {
  name     : String    = assignment.name + " " + student.name        (on-demand)
  answer   : String?
  deadline : Datetime? = assignment.deadline <+ parent.deadline      (default)
  finished : Datetime?
  onTime   : Boolean   = finished <= deadline <+ true
  grade    : Float?    = if(conj(children.pass)) avg(children.grade) (default)
  pass     : Boolean   = grade >= assignment.minimum && onTime <+ false
}
relation Submission.student    1 <-> * Student.submissions
relation Submission.assignment 1 <-> * Assignment.submissions
relation Assignment.parent     ? <-> * Assignment.children
relation Submission.parent     ? =
  assignment.parent.submissions.find(x => x.student == student)
                  <-> * Submission.children
```

Figure 6.1: ICEDUST model of a simplified learning management system [23].

Bidirectional recursion. Similar to ALANLIGHT, ICEDUST [23] is data modeling language with integrated support for declaratively modeling constraints and derivations as an integral part of an object specification. ICEDUST also admits recursion, but neither guarantees functionally correct output nor termination. For demonstrating the capabilities of ICEDUST, Harkes et al. [23] use a running example containing a complex calculation requiring bidirectional recursion on a graph (by recursively following references and their inverse).

ALANLIGHT supports recursion on graphs that references explicitly partake in. ALANLIGHT also supports inverse references, but the presented static semantics (Chapter 4) does not cover recursive traversal of inverse references. However, a minor extension to ALANLIGHT solves this problem; our semantics checker implements this extension. For demonstrating this, we use the running example from a paper presenting ICEDUST [23]. We also use this example to highlight differences between ALANLIGHT and ICEDUST, and to further evaluate important properties of ALANLIGHT.

The paper presenting ICEDUST uses a simplified learning management system as running example. The system consists of `Students` submitting `Submissions` to `Assignments`. `Assignments` form trees (they recursively consist of sub-assignments), and it is *assumed* (not guaranteed by the ICEDUST model) that `Submissions` mirror these trees. Figure 6.1 shows the ICEDUST model from the paper.

The recursive calculation determines if `Submissions` pass, and includes determining

```
root {
  Students: collection { name: text }
  Assignments: collection ~ ass_g = acyclic-graph {
    question: text
    minimum: natural
    hasParent: stategroup (
      yes { paren: ~ ref(this.parent.parent.Assignments[ass_g]) }
      no {} )
    hasDeadline: stategroup ( yes { deadline: natural } no { } )
    children: root.Assignments = inv-refs (.hasParent|no>paren)
    Submissions: collection ~ ref(this.parent.Students) {
      answer: text
      finished: natural
      gradeType: stategroup ( manual { grade: natural } computed {} )
      deadlineType: stategroup ( custom { deadline: natural } default {} )
      hasDeadline: stategroup = switch (this.deadlineType) as dT (
        | custom = yes (date = dT.deadline)
        | default = switch(this.parent.hasDeadline) as hD (// assignment deadline?
          | yes = yes (date = hD.deadline)
          | no  = switch (this.parent.hasParent) as pA ( // assignment has parent?
            | yes = empty (pA>paren.Submissions[this>key]) as pS
              | true = switch (pA>paren.hasDeadline) as pAD (//ass paren deadline?
                | yes = yes (date = pAD.deadline)
                | no  = no )
              | false = switch (pS.hasDeadline) as pSD ( // parent subm. deadline?
                | yes = yes (date = pSD@date)
                | no  = no )
            | no  = no ) ) )
      ( yes (date: natural) { deadline: natural = date } no {} )
      onTime: stategroup = switch (this.hasDeadline) as dl (
        | yes = match (this.finished <= dl.deadline) | true = yes | false = no
        | no  = yes )
      ( yes {} no {} )
      pass: stategroup = switch (this.onTime) (              // finished on time?
        | yes = switch (this.gradeType) as gt (
          | manual = match (grade >= this.parent.minimum)   // grade above minimum?
            | true  = yes (grade = gt.grade) | false = no
          | computed
            = empty (this.parent<children.Submissions[this>key].pass|yes) as cpass
            | true  = no
            | false = match(count(cpass) == count(this.parent<children))
              | true  = match (avg(cpass@grade) >= this.parent.minimum)// avg pass?
                | true  = yes (grade = avg(cpass@grade))
                | false = no
              | false = no )
        | no = no
      ) (
        yes (grade: natural) { finalGrade: natural = grade }
        no {}
      )
    }
    avgGrade: natural = empty (this.Submissions.pass|yes) as s
      | true = one  | false = avg(s.finalGrade)
    passPerc: integer = empty (this.Submissions) as s
      | true = zero | false = count(s.pass|yes) / count(s)
  }
}
```

Figure 6.2: ALANLIGHT implementation of the ICEDUST model from Figure 6.1.

grades for `Submissions`. The calculation essentially consists of two parts: (1) calculating if submissions are `onTime` using user-provided references and (2) calculating if submissions `pass` corresponding requirements using derived inverse references.

As for the first part, `Assignments` have `deadlines`. These `deadlines` are optional; `Submissions` inherit them if they exist. If they do not exist, `Submissions` use the deadline from their parent submission, if it exists. Assignment instructors can also override `deadlines` of `Submissions` with custom submission-specific `deadlines`.

For the second part of the calculation, instructors grade leaf `Submissions` by assigning a `grade`. Leaf `Submissions` `pass` when finished before the `deadline` passes and their grade is greater than or equal to the `minimum` assignment grade. Other (composite) `Submissions` pass if their `children` pass and the average grade of their `children` is greater than or equal to the `minimum` assignment grade.

Figure 6.2 presents an ALANLIGHT implementation of the learning management system. The figure uses minor extensions to ALANLIGHT, including boolean comparison, a match expression matching boolean comparison results, key lookup in a collection, and the option to reference a state input parameter (e.g. `@grade`) inside an expression.

More importantly, the model uses the inverse reference attribute `children` in the recursive `pass` calculation. If a reference partakes in a graph, its inverse implicitly partakes in the inverse of that graph (which is also acyclic). For ensuring termination, ALANLIGHT's extended semantics creates inverse graph-sets G for inverse references. Furthermore, it performs one additional operation when combining graph-sets (from first-sets and follow-sets): if one graph-set contains a graph and another contains its inverse, the combined graph-set contains neither.

ICEDUST models consist of a list of objects types, and relations among these object types. Attributes in ICEDUST hold atomic values. Therefore, `Assignments`, `Submissions`, `Students`, and the relations among them are defined as a list. In contrast, ALANLIGHT supports complex (nested) object structures, and requires expressing these as such. That is, `Assignments` in the ALANLIGHT model nest `Submissions` and relations among the object types. This is important, as ALANLIGHT's static semantics requires this for the recursive calculations. (We can extend ALANLIGHT to support recursion without the nesting requirement as well, but we consider this beyond the scope of this thesis.)

The ICEDUST example *assumes* that `Submissions` trees mirror `Assignments` trees. ICEDUST requires this assumption for ensuring consistency. In contrast, ALANLIGHT requires no such assumption; `Submissions` are added when `Students` submit them. The `finished` attribute values correspond to the submit date.

As mentioned before, the recursive calculation consists of two parts: calculating if a submission is `onTime` and calculating if it is a `pass`. For calculating if a `Submission` is `onTime`, we recursively calculate its `deadline`. The `deadline` calculation uses the `paren` reference for retrieving a parent submission. That is, if the submission has a `default` deadline and the assignment has no deadline. If a parent submission exists on the parent assignment, the `empty` clause produces this submission, and the calculation recursively uses its `deadline`. The `onTime` calculation checks the calculated `deadline` against `finished`, yielding the required result.

Students only receive grades for composite Submissions that pass. If the submission is onTime (state yes), it may pass: we check the gradeType. If the gradeType is manual, we check if the entered grade conforms to the minimum required grade. Otherwise, we check if (1) the student has child Submissions that pass (recursively using inverse references), (2) if the student submitted Submissions to all child Assignments, and (3) if the average child-submission grade conforms to the minimum required grade of the composite assignment. Only then, the student gets a finalGrade for a composite submission, which conforms to the system specification.

The ICEDUST model allows violations of the system specification, leading to incorrect or inconsistent data. For example, the ICEDUST model allows circular dependencies among Assignments. In addition, the ICEDUST model allows Students to submit multiple times to the same assignment, because ICEDUST neither supports nesting nor more complex constraints on relations. In the presence of cycles, calculations in ICEDUST can fail or produce incorrect results. The type of failure depends on the calculation strategy; ICEDUST implements several different strategies. Eager incremental evaluation neither guarantees functional correctness nor termination. The lazy incremental evaluation strategy and the on-demand evaluation strategy can run infinitely. In contrast, ALANLIGHT guarantees correct and consistent results.

Numerically bounded recursion. In some recursive calculations, atomic (typically numerical) values bound the recursion. ALANLIGHT cannot express this; by design, as we further explain below. For example, ALANLIGHT cannot *generate* all Fibonacci numbers up to a certain numerical bound, while R-SQL [2] supports this:

```
fib1(n float, f float) := SELECT fib.n, fib.f FROM fib;
fib2(n float, f float) := SELECT fib.n, fib.f FROM fib;

fib(n float, f float)  := SELECT 0,1 UNION SELECT 1,1 UNION
  SELECT fib1.n+1,fib1.f+fib2.f FROM fib1,fib2
  WHERE fib1.n=fib2.n+1 AND fib1.n<10;
```

Note that R-SQL does not guarantee termination; without the bound fib1.n<10, the query runs infinitely.

Although ALANLIGHT precludes *generating* (rows holding) Fibonacci numbers, the language does support *calculating* Fibonacci numbers. To this end, the language requires either a fixed number of copies of the expression, or alternatively requires creating entities (rows) in advance:

```
fib_numbers: collection ~ g_iter = acyclic-graph {
  has_previous: stategroup (
    yes { prev: ~ ref (this.parent.parent.fib_numbers[g_iter]) }
    no { }
  )
  fib: integer = switch (this.has_previous) as fib2 (
    | no  = one
    | yes = switch (fib2>prev.has_previous) as fib1 (
      | no  = one
      | yes = fib1>prev.fib_number + fib2>prev.fib_number ) )
}
```

Observe that this requirement is essential for guaranteeing ALANLIGHT's boundedness properties; without this requirement, boundedness in trivially countable size properties of a model would no longer hold. Instead, the running time would also depend on numerical values. What is more, the PTIME bound in the maximum size of a user provided collection would no longer hold when supporting concepts required for the R-SQL query. R-SQL generates rows (collection items) from atomic values. The number of generated rows can easily exceed the maximum size of a user provided collection.

Similar to the above problem, ALANLIGHT cannot express algorithms involving a dynamic number of iterations; that is, if this dynamic number is calculated from atomic numerical values in a dataset. An example is the PageRank [52] algorithm, which calculates ranks for web pages. To this end, the algorithm repeatedly traverses graphs of web nodes, modifying rank values until convergence – in the form of a numerical rank difference threshold – is reached. Many systems for complex data processing support such algorithms, including DEALS [64], GRAPHX [73] (Apache Spark library), GELLY [9] (Apache Flink library), HUSKY [74], and NAIAD [46] which includes differential dataflow [39] and GRAPHLINQ. The following figure shows a PageRank algorithm for a single web node expressed in GRAPHX, GELLY, and HUSKY using the Pregel API [35]:

```
def web_node_exec(v): pr = (v.get_msgs()) * 0.85 + 0.15
  v.set_value(pr)
  for nb in v.get_nbs():
    v.send_msg(pr/len(v.get_nbs()), nb.id)
```

The algorithm iterates over the web nodes, updating their PageRank (pr) value using the PageRank equation. For the next iteration, each vertex sends an equally divided part of its PageRank value to the neighbouring nodes that it references. The algorithm stops iterating web nodes when the PageRank value converges, which is after a dynamic number of iterations.

However, a dynamic number of iterations gives unpredictable running times for large datasets. Furthermore, computations may diverge, and queries may not terminate or yield incorrect values. Therefore, the number of iterations is sometimes bounded by a static number value indicating a maximum number of iterations. The bound enables expressing PageRank in ALANLIGHT as well. We can apply the presented approach for calculating a bounded amount of Fibonacci numbers.

Note that preventing a dynamically calculated number of iterations is in fact *essential* to guaranteeing ALANLIGHT's time bounds. ALANLIGHT guarantees boundedness in trivially countable *size* properties of a model and a corresponding dataset. Supporting a dynamically calculated number of iterations precludes that, yielding running times that depend on atomic values in a dataset.

Summary. In summary, ALANLIGHT requires acyclic graph traversal for recursive calculations, and thus precludes recursion on potentially cyclic graphs. In addition, ALANLIGHT precludes generating entities (rows) from atomic values. The language only creates new entities for existing entities, essentially augmenting them. Furthermore, ALANLIGHT precludes expressing algorithms where dynamically calculated atomic values determine the number of iterations of a looping construct. These limitations are essential for achieving ALANLIGHT's key properties.

6.2 Threats to validity

A threat to the validity of our approach is that ALANLIGHT's semantics lacks a formal proof of termination. Our future work may include such a proof. As part of this work, we developed a tool implementing the analysis (ALANLIGHT's static semantics). The tool produced expected results for all programs that we ran it on. These programs include an accounting system that we ported from ALAN to ALANLIGHT, programs describing parts of other ALAN systems that use recursive computations, and examples from this chapter.

As mentioned in Chapter 1, ALAN is being used as the principal programming language for engineering data-intensive software systems at M-industries. M-industries has successfully deployed several ALAN-based systems at manufacturing sites all over Europe which are actively being used as integral parts of manufacturing processes. We verified that ALANLIGHT's concepts cover all recursive computations that these systems require. From this, we conclude that our approach works for real-life use cases.

Chapter 7

Related Work

This chapter compares ALAN to related work, addressing several aspects of ALAN that we have not discussed so far. First, we discuss support for expressing the structure of data, constraints on data, and derivations from data. Subsequently, we evaluate support for recursive computations, and recapitulate approaches for providing guarantees such as termination. We conclude this chapter with a discussion on dependent typing.

Structure, constraints, and derivations. Equivalent to ALANLIGHT, XML schema languages (XML SCHEMA [36], SML [66], RELAX NG [70], SCHEMATRON [14], etc.) support specifying hierarchical data structures in hierarchical manner. To a varying extent, XML schema languages also support expressing referential integrity constraints. For example, SML [66] supports expressing graph (acyclicity) constraints, but only between different XML documents. XML schema languages are meant for validating XML documents, and do not support expressing derived values. For deriving values from XML documents, XML based technologies typically use XSLT or XQUERY. But, as XSLT and XQUERY are Turing-complete [28, 50] languages, they are unable to generally guarantee bounded running times. In contrast, ALANLIGHT enables inline declarative specification of derived values, has a type system that uses referential integrity constraints (including acyclicity constraints) ensuring correctness, and is not Turing-complete.

SQL databases store data in a tabular format, and SQL database schemas describe a flat list of tables. SQL databases have several limitations, as Kent [27] and Reiter [54] point out. For example, they allow incompleteness (`NULL` values) and lack a built-in mechanism for subtyping. SQL also has only rudimentary support for referential integrity constraints [19]. Enforcing more complex constraints requires writing custom code in a GPL, or a separate language such as OCL [13].

OCL is a widely adopted navigation-based query language for object graphs [69]. It was originally designed to overcome limitations of the UML [8]. The Eclipse Modeling Framework (EMF) offers the UML dialect ECORE for modeling data, and supports expressing constraints and derivations in ECLIPSE OCL [1]. VIATRA QUERY (formerly [2] EMF-

[1]https://wiki.eclipse.org/OCL/OCLinEcore

[2]https://viatra.net/news/2016/2/say-goodbye-to-emf-incquery-say-hello-to-viatra-query

IncQuery [69]) offers an alternative solution for defining constraints and derivations over ECORE models. However, VIATRA QUERY cannot guarantee values of a predefined indivisible type, as it provides no multiplicity guarantees. Furthermore, ECLIPSE OCL and VIATRA QUERY both allow the use of arbitrary JAVA code, making them Turing-complete.

Deductive database languages such as DATALOG [31] declaratively define programs (including the structure of data, constraints, and derivations) as a flat list of rules called logical predicates. A rule essentially defines an object, consisting of *atomic* attribute values and relations with other rules which themselves recursively consist of atomic attribute values. However, many practical applications in different fields require storing, manipulating, and computing complex nested objects [10]; for example, applications for exchanging data between machines and supporting systems. These objects typically require a strict predefined structure and conformance to complex constraints. Contrary to typical (rule based) deductive database languages, ALANLIGHT supports expressing complex nested objects, and guarantees strict schema conformance.

As DATALOG itself is too limiting for practical applications, many extensions have been proposed [10, 18]. For example, LOGIQL [21] extends DATALOG featuring advanced constraints, including graph constraints. However, contrary to ALANLIGHT, LOGIQL does not use these constraints in its type system for ensuring safe and correct recursive calculations. Furthermore, LOGIQL does not guarantee termination [18, 59]. To our knowledge, the aforementioned extended DATALOG DEAL [63], does not support constraints. In the DATALOG community, supporting recursive calculations (aggregation and negation) is an important subject of discussion. Different proposals for supporting it exist, but the community has not converged to any specific proposal [18].

As mentioned before, ICEDUST [23] is a data modeling language with integrated support for expressing constraints and derivations. Similar to ALANLIGHT, it uses expressions for inline specification of derived values at attributes (unlike DATALOG which uses rules that are independent of an object structure, and other approaches we mentioned so far, which use separate query languages). ICEDUST also features multiplicity guarantees.

ICEDUST models consist of a list of object type definitions and separately defined relations among these object types. ICEDUST objects hold atomic values, meaning that object definitions in IceDust are conceptually identical to table definitions in SQL. ICEDUST has rudimentary support for referential integrity constraints; the expressiveness is equivalent to single field foreign key constraints in SQL. That is, ICEDUST supports expressing relations among objects, but relations cannot use (complex) navigation paths expressing dependence on other relations. ICEDUST supports deriving atomic values and relations. In contrast, ALANLIGHT featuring complex nested object type definitions and complex referential integrity constraints. Furthermore, ALANLIGHT supports deriving complex nested objects.

In summary: in contrast to related work, ALANLIGHT offers *integrated* support for specifying data and data flow in a very precise and detailed manner. This enables ALANLIGHT's soundness and functional correctness guarantees, contrasting related work: satisfying the requirements for these guarantees requires developers to give a very detailed and precise specification of data and data flow.

Recursion and operational semantics. R-SQL, ECLIPSE OCL, VIATRA QUERY, LOGIQL, and DEALS all use a fixpoint semantics for ensuring terminating recursive calculations yielding potentially incorrect values. In contrast, ALANLIGHT prevents cycles for recursive calculations before performing them.

Typical XML query languages such as XSLT and XQUERY, are Turing-complete; they do not guarantee termination. Typical SQL implementations bound recursion with timeouts or user specified bounds on the number of iterations in recursion. R-SQL implements stratified fixpoint computation, ensuring terminating transitive closure computations. But, as the termination condition of the fixpoint algorithm is based on the inability to generate new rows, termination is not guaranteed in the presence of (arbitrary) arithmetic inside recursion (i.e. arithmetic can yield an infinite number of unique rows).

VIATRA QUERY features stratified recursion, and uses a fixpoint algorithm for recursive calculations. The default algorithm for evaluating recursion in VIATRA QUERY does not guarantee correct (least fixpoint) results. For guaranteed least fixpoint results, the developer can enable the DRED [20] algorithm which ensures a least fixpoint, but incurs additional overhead. VIATRA QUERY also allows the use of arbitrary JAVA code, making it Turing-complete. Another example of a language applying the DRed algorithm for incremental recursive computation is I3QL [43]. I3QL is a Turing-complete language; a general purpose programming language designed for specifying incremental computations.

LOGIQL features stratified recursion (behind a compiler flag [23]), and the engine uses a stratified partial fixpoint semantics for evaluating recursive programs [3]. However, this semantics does not guarantee fixpoint convergence in the presence of arithmetic. According to Zinn 3, LOGIQL detects cases where an earlier state is reproduced, and consequently aborts the transaction with an error. This means that LOGIQL (1) requires tracking earlier states in a computation and requires checking earlier states against subsequent states, (2) does not support on-demand evaluation of derived values (while guaranteeing correct output), and (3) does not guarantee (correct) output for recursive calculations before evaluating them. Furthermore, as stated before, LOGIQL does not guarantee termination [18, 59].

DATALOG^{FS} [38] features more advanced support for recursion, while also guaranteeing termination. The language supports a limited form of aggregation inside recursion: monotonic aggregation. DEAL supports this as well; monotonic aggregation in DEAL is based on DATALOG^{FS} work [38, 37]. In addition, DEAL also supports user defined aggregates. Furthermore, DEAL features XY-stratified recursion, which further increases the expressiveness of the language. XY-stratification is a form of local stratification that incorporates an iteration counter (a temporal argument) in rules. In this approach, subsequent iterations of a fixpoint algorithm check newly derived facts against earlier derived facts from previous iterations to prevent reevaluation.

Other DATALOG extensions with concrete implementations include BIGDATALOG and SOCIALITE. BIGDATALOG is a DATALOG implementation on Apache Spark, incorporating extensions for monotonic aggregation found in DEAL (it is developed as part of the DEALS project [62]). SOCIALITE [59, 60] has very limited support for recursive aggregation. For example, unlike DEAL it does not support recursive summation.

[3]https://developer.logicblox.com/2013/08/logiql-4-x-fixpoint-semantics/

ICEDUST utilizes a value-oriented (instead of tuple/row/fact-oriented) fixpoint algorithm for recursive aggregation of atomic values. The algorithm terminates when the (atomic) calculated value no longer changes between successive iterations of the algorithm. However, ICEDUST neither guarantees correct output (a unique least fixpoint) nor termination; computations may diverge.

Dependent types. SCALA [48] features a limited form of dependent types: path-dependent types. Path-dependent types in SCALA enable constructing object instances of a (nested) descendant type using a parent object. Because ALANLIGHT models define type hierarchies, object construction does not use path-dependent typing.

ALANLIGHT supports a similar restricted form of dependent typing: *path-dependent relation typing*. Reference attributes and keys representing references have navigation path that are interpretated relative to their context node on the instance level. They can depend on other reference attributes or collection keys:

```
products: collection {
  parts: collection ~ ref(this.parent.products) { .. }
}
my_product: ~ ref(this.products)
broken_part: ~ ref(this>my_product.parts)
```

VIATRA QUERY supports expressing path-dependent relations with constraints on top of references [24]. However, the language requires expressing references themselves in a separate language such as ECore, while ALANLIGHT supports expressing a path-dependent relation as an integral part of a reference definition.

Chapter 8

Conclusions and Future Work

In the introduction we already summarized our main contributions; for convenience we repeat them here. The contribution of this thesis is a declarative data modeling language, for expressing complex recursive calculations, while guaranteeing *soundness* (where values are always of a *predefined indivisible type*), *functional correctness* (which includes deterministic output), *polynomial time complexity* in the *size* of user data, and a *combined complexity* in the *size* of (trivially countable properties of) the specification.

In Chapter 2 we have provided an extensive overview of problems with existing languages for data transformation. In Chapter 3 we have demonstrated the need for recursive calculations. Furthermore, we have discussed the requirements that ALANLIGHT imposes on recursive calculations, for ensuring soundness, functionally correctness, and bounded, predictable running times. For statically checking requirements for safe, functionally correct, and bounded ALANLIGHT programs, we have presented a formal static semantics (Chapter 4), focusing mainly on non-trivial rules.

For understanding the meaning of concepts in ALANLIGHT, and also for guaranteeing the aforementioned boundedness properties, we provided an operational semantics for constraint checking and for on-demand minimal effort calculation of derived values (Chapter 5). ALANLIGHT's operational semantics shows that ALANLIGHT performs exactly those calculations that lead to a final demanded result. Furthermore, the semantics shows that ALANLIGHT performs calculations *at most once* during expression evaluation. We designed ALANLIGHT's semantics such that evaluation rules trivially map to a concrete implementation.

From ALANLIGHT's operational semantics, we derived upper bounds on the dynamic number of calculations and the running time of programs in the language. These bounds show that ALANLIGHT indeed guarantees polynomial time complexity in the size of user data, and a combined complexity in the size of trivially countable properties of ALANLIGHT models. ALANLIGHT's boundedness guarantees enable accurate worst-case running time predictions for changing data.

The question we asked in our problem statement was: *to what extent can we support complex recursive calculations, while guaranteeing soundness, functional correctness, and bounded, predictable running times?* To answer this question, we addressed different kinds of recursive (and iterative) computations. We showed that ALANLIGHT's concepts enable many complex recursive calculations found in related work, while providing aforementioned guarantees. ALANLIGHT exceeds the capabilities of existing approaches with regard to support for recursive calculations under aforementioned guarantees.

Two specific classes of problems are not expressible in ALANLIGHT: problems where atomic (numerical) values bound computations, and problems involving recursion on cyclic graphs. Precluding the first class of problems is essential to guaranteeing aforementioned running time bounds. We can typically reformulate problems from this class, such that ALANLIGHT can express them. For the second class of problems, we can use a fixpoint semantics. However, ensuring correct output requires very specific restrictions on operators; for example, restricting to operators `min` and `max` in a recursive expression. As future work, we consider investigating the possibility of mixing ALANLIGHT's semantics with such a semantics.

8.1 Future work

Expressiveness and conciseness. We would like to formalize several features already present in ALAN for more concise models and more expressiveness. First, we would like to formalize type reuse. For type reuse, ALAN supports component types; named types defined alongside the root type, similar to classes in JAVA. Second, we would like to formalize support for mutual recursion among earlier and later defined attributes. Note that we can integrate type reuse and mutual recursion into ALANLIGHT's semantics by simply inlining component types and expressions, respectively.

Apart from features already present in ALAN, we would like to investigate the possibility of *graph inheritance* in ALANLIGHT's static semantics. To demonstrate what we mean by this, consider the following model:

```
Products: collection ~ parts = acyclic-graph { .. }
ConfProducts: collection ~ ref(this.Products) { .. }
```

`ConfProducts` are `Products` that a manufacturer has configured. Because `Products` form an acyclic graph, `ConfProducts` also form an acyclic graph, implicitly. That is, `ConfProducts` inherit the acyclic `parts` property of `Products`. From references among `Products`, we can derive references among `ConfProducts`. These references implicitly partake in the inherited acyclic `parts` graph. As such, we can safely support calculations that recursively traverse them. That is, without the need to explicitly provide a graph and corresponding elementary references among `ConfProducts`. We are currently extending ALANLIGHT with features such as these, and gradually replacing ALAN with the extended version of ALANLIGHT.

We would also like to investigate the possibility of combining ALAN's semantics with a fixpoint semantics for enabling recursive calculations on cyclic graphs where correct output can be guaranteed. For example, for for a shortest path calculation on a cyclic graph.

Formal definition of expressiveness. Our evaluation gives insight into the types of problems that ALANLIGHT can express. However, we would like a formal definition of the class of problems that ALANLIGHT covers. That is, we would like an answer to the question: which class of problems can ALANLIGHT express, and which class of problems is inexpressible in ALANLIGHT?

Formal proof of termination. A thread to the validity of our approach is the lack of a formal proof of termination. We would like to construct such a proof with a formal proof management system such as COQ [4].

Calculation strategies. On a different note, we would like to research the impact of ALANLIGHT restrictiveness on incremental computation of derived values. In addition, we would like to compare the real-world performance of ALANLIGHT to existing approaches, and investigate distributed and parallel computation of derived values.

Furthermore, we would like to compare the real-world performance of approaches applying a fixed-point semantics to the real-world performance of ALANLIGHT's (extended) semantics. This is interesting because a fixed-point semantics induces overhead during (on-demand) derived value computation, while ALANLIGHT induces overhead in the form of graph maintenance during transaction processing.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. Foundations of databases: the logical level. Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] Gabriel Aranda, Susana Nieva, Fernando Sáenz-Pérez, and Jaime Sánchez-Hernández. Formalizing a broader recursion coverage in sql. In International Symposium on Practical Aspects of Declarative Languages, pages 93–108. Springer, 2013.

[3] Faiz Arni, KayLiang Ong, Shalom Tsur, Haixun Wang, and Carlo Zaniolo. The deductive database system ldl++. Theory and Practice of Logic Programming, 3(1):61–94, 2003.

[4] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant Reference Manual : Version 6.1. Research Report RT-0203, INRIA, May 1997. Projet COQ.

[5] Gábor Bergmann, Dénes Harmath, and Tamas Szabo. VIATRA Query user documentation: Advanced Patterns. 2017.

[6] Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 220–233. ACM, 1980.

[7] Rick Byham, Bruce Hamilton, Theano Petersen, and Craig Guyer. Top (transact-sql).

[8] Jordi Cabot and Martin Gogolla. Object constraint language (ocl): a definitive guide. In Formal methods for model-driven engineering, pages 58–90. Springer, 2012.

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 36(4), 2015.

[10] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering, 1(1):146–166, 1989.

[11] Mariano P Consens and Alberto O Mendelzon. Low complexity aggregation in Graphlog and Datalog. In International Conference on Database Theory, pages 379–394. Springer, 1990.

[12] Douglas Crockford. JavaScript: The Good Parts: The Good Parts. " O'Reilly Media, Inc.", 2008.

[13] Birgit Demuth and Heinrich Hussmann. Using UML/OCL constraints for relational database design. In «UML»'99—The Unified Modeling Language, pages 598–751. Springer, 1999.

[14] Leigh Dodds. Schematron: validating xml using xslt. In XSLT UK Conference, Keble College, Oxford, England, 2001.

[15] Andrew Eisenberg and Jim Melton. Sql: 1999, formerly known as sql3. ACM Sigmod record, 28(1):131–138, 1999.

[16] Melvin Fitting and Marion Ben-Jacob. Stratified and three-valued logic programming semantics. In ICLP/SLP, pages 1054–1069, 1988.

[17] James Gosling. The Java language specification. Addison-Wesley Professional, 2000.

[18] Todd J Green, Shan Shan Huang, Boon Thau Loo, Wenchao Zhou, et al. Datalog and recursive query processing. Foundations and Trends® in Databases, 5(2):105–195, 2013.

[19] James R Groff, Paul N Weinberg, et al. SQL: the complete reference, volume 2.

[20] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. Maintaining views incrementally. ACM SIGMOD Record, 22(2):157–166, 1993.

[21] Terry Halpin and Spencer Rugaber. LogiQL: A Query Language for Smart Databases. CRC Press, 2014.

[22] Daco C Harkes, Danny M Groenewegen, and Eelco Visser. Icedust: Incremental and eventual computation of derived values in persistent object graphs. In LIPIcs-Leibniz International Proceedings in Informatics, volume 56. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[23] Daco C Harkes and Eelco Visser. Icedust 2: Derived bidirectional relations and calculation strategy composition. In LIPIcs-Leibniz International Proceedings in Informatics, volume 74. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[24] Abel Hegedus, Gábor Bergmann, Dénes Harmath, and Tamas Szabo. Viatra query user documentation: Query language. 2017.

[25] Adam Jorgensen, Jorge Segarra, Patrick LeBlanc, Jose Chinchilla, and Aaron Nelson. Microsoft SQL Server 2012 Bible, volume 773. John Wiley & Sons, 2012.

[26] Owen Kaser, CR Ramakrishnan, and Shaunak Pawagi. On the conversion of indirect to direct recursion. ACM Letters on Programming Languages and Systems (LOPLAS), 2(1-4):151–164, 1993.

[27] William Kent. Limitations of record-based information models. ACM Transactions on Database Systems (TODS), 4(1):107–131, 1979.

[28] Stephan Kepser. A simple proof for the turing-completeness of xslt and xquery. In Extreme Markup Languages®, 2004.

[29] Brian W Kernighan and Dennis M Ritchie. The C programming language. 2006.

[30] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pages 207–220. ACM, 2009.

[31] Phokion G Kolaitis and Moshe Y Vardi. On the expressive power of datalog: tools and a case study. Journal of Computer and System Sciences, 51(1):110–134, 1995.

[32] Robert P. Kurshan. Computer-aided verification. IEEE Spectrum, 33:61–67, 1996.

[33] Daniel Le Métayer. Ace: An automatic complexity evaluator. ACM Transactions on Programming Languages and Systems (TOPLAS), 10(2):248–266, 1988.

[34] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In International Conference on Logic for Programming Artificial Intelligence and Reasoning, pages 348–370. Springer, 2010.

[35] Jinfeng Li, James Cheng, Yunjian Zhao, Fan Yang, Yuzhen Huang, Haipeng Chen, and Ruihao Zhao. A comparison of general-purpose distributed systems for data processing. In Big Data (Big Data), 2016 IEEE International Conference on, pages 378–383. IEEE, 2016.

[36] Murray Maloney, Noah Mendelsohn, David Beech, and Henry Thompson. XML schema part 1: Structures second edition. W3C recommendation, W3C, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

[37] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. An extension of datalog for graph queries. In Twentieth Italian Symposium on Advanced Database Systems, SEBD 2012, Venice, Italy, June 24-27, 2012, Proceedings, pages 177–184, 2012.

[38] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. Extending the power of datalog recursion. The VLDB Journal, 22(4):471–493, 2013.

[39] F.D. McSherry, R. Isaacs, M.A. Isard, and D.G. Murray. Differential dataflow, October 20 2015. US Patent 9,165,035.

[40] Jim Melton and Alan R Simon. Understanding the new SQL: a complete guide. Morgan Kaufmann, 1993.

[41] Robin Milner. A theory of type polymorphism in programming. Journal of computer and system sciences, 17(3):348–375, 1978.

[42] Robin Milner. The definition of standard ML: revised. MIT press, 1997.

[43] Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. i3ql: Language-integrated live data views. ACM SIGPLAN Notices, 49(10):417–432, 2014.

[44] Bruce Momjian. PostgreSQL: introduction and concepts, volume 192. Addison-Wesley New York, 2001.

[45] Karen Morton, Kerry Osborne, Robyn Sands, and Riyaj Shamsudeen. Pro Oracle SQL. Springer.

[46] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 439–455. ACM, 2013.

[47] AB MySQL. Mysql, 2001.

[48] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, 2004.

[49] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.

[50] Ruhsan Onder and Zeki Bayram. Xslt version 2.0 is turing-complete: A purely transformation based proof. In International Conference on Implementation and Application of Automata, pages 275–276. Springer, 2006.

[51] Mike Owens and Grant Allen. SQLite. Springer, 2010.

[52] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[53] Piotr Przymus, Aleksandra Boniewicz, Marta Burzańska, and Krzysztof Stencel. Recursive query facilities in relational databases: a survey. Database Theory and Application, Bio-Science and Bio-Technology, pages 89–99, 2010.

[54] Raymond Reiter. Towards a logical reconstruction of relational database theory. In On conceptual modelling, pages 191–238. Springer, 1984.

[55] Tiark Rompf and Nada Amin. Type soundness for dependent object types (dot). In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 624–641. ACM, 2016.

[56] Mads Rosendahl. Automatic complexity analysis. In Proceedings of the fourth international conference on Functional programming languages and computer architecture, pages 144–156. ACM, 1989.

[57] Kenneth A Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, pages 114–126. ACM, 1992.

[58] Fernando Sáenz-Pérez, Susana Nieva, Jaime Sanchez-Hernandez, and Gabriel Aranda. R-sql: An sql database system with extended recursion. Electronic Communications of the EASST, 64, 2014.

[59] Jiwon Seo, Stephen Guo, and Monica S Lam. Socialite: Datalog extensions for efficient social network analysis. In Data Engineering (ICDE), 2013 IEEE 29th International Conference on, pages 278–289. IEEE, 2013.

[60] Jiwon Seo, Stephen Guo, and Monica S Lam. Socialite: An efficient graph query language based on datalog. IEEE Transactions on Knowledge and Data Engineering, 27(7):1824–1837, 2015.

[61] SQL Server. T-sql for sql server 2000 programmers.

[62] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In Proceedings of the 2016 International Conference on Management of Data, pages 1135–1149. ACM, 2016.

[63] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. Optimizing recursive queries with monotonic aggregates in deals. In Data Engineering (ICDE), 2015 IEEE 31st International Conference on, pages 867–878. IEEE, 2015.

[64] Alexander Shkapsky, Kai Zeng, and Carlo Zaniolo. Graph queries in a next-generation datalog system. Proceedings of the VLDB Endowment, 6(12):1258–1261, 2013.

[65] Michael Sipser. Introduction to the Theory of Computation, volume 2. Thomson Course Technology Boston, 2006.

[66] Virginia Smith and Valentina Popescu. Service modeling language, version 1.1. W3C recommendation, W3C, May 2009. http://www.w3.org/TR/2009/REC-sml-20090512/.

[67] Bjarne Stroustrup. The C++ programming language. Pearson Education India, 1986.

[68] Simon Thompson. Haskell: the craft of functional programming, volume 3. Addison Wesley Reading, 1999.

[69] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. Emf-incquery: An integrated development environment for live model queries. Science of Computer Programming, 98:80–99, 2015.

[70] Eric Van der Vlist. Relax NG: A Simpler Schema Language for XML. " O'Reilly Media, Inc.", 2003.

[71] K. Wicovsky, Peter Thanisch, and M. Howard Williams. Improved recursion handling through integrity constraints. Comput. J., 34(3):282–285, 1991.

[72] Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. Information and computation, 115(1):38–94, 1994.

[73] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In First International Workshop on Graph Data Management Experiences and Systems, page 2. ACM, 2013.

[74] Fan Yang, Jinfeng Li, and James Cheng. Husky: Towards a more efficient and expressive distributed computing framework. Proceedings of the VLDB Endowment, 9(5):420–431, 2016.

[75] Carlo Zaniolo, Mohan Yang, Ariyam Das, and Matteo Interlandi. The magic of pushing extrema into recursion: Simple, powerful datalog programs. In AMW, 2016.