ETH Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Department of Computer Science
Chair of Programming Methodology

Master's Thesis

Must Analysis of Collection Elements

September 1, 2013

Author:
Yves Bonjour

Supervisors:
Lucas Brutschy
Prof. Dr. Peter Müller

Acknowledgments

I would like to thank Prof. Dr. Peter Müller for giving me the opportunity to write my master's thesis at the Chair of Programming Methodology.

I am very grateful to my supervisor Lucas Brutschy for his support and valuable feedback. Furthermore, I would like to thank all the members of the Chair of Programming Methodology. In particular Dr. Pietro Ferrara for helping me to better understand the concepts of abstract interpretation.

Abstract

TouchDevelop is a novel programming language and environment developed by Microsoft Research that aims at writing scripts on mobile devices. An integral part of the TouchDevelop standard library are collections like lists or maps. The faulty use of collections (e.g. accessing a map at an invalid key) can give rise to errors. We want to help developers avoid such errors by presenting them accurate warnings about potential bugs in their scripts. To achieve this, we introduce an approach to statically track the elements of collections in TouchDevelop scripts.

Because of the unbounded nature of collections, it is not feasible to track individual elements of a collection. We need to abstract the properties of collection elements that we are interested in. We present two complementary abstractions: The first abstraction, the *May Analysis*, captures whether an element might be in a collection and the second abstraction, the *Must Analysis*, captures whether an element must be in a collection. To solve the problem of unboundedness, we summarize all elements that were added to a collection at the same program point.

By specifying a high-level representation for collections and defining the semantics of collection operations based on this representation, we are able to handle a wide range of different collection types. Particularly we can handle all collection types defined by the TouchDevelop standard library. Our analysis distinguishes between the shape of the collection and the values of the collection elements. To abstract the values of collection elements we use an exchangeable value domain that can handle object references as well as primitive values.

We show that the implemented *Must Analysis* improves the precision of hand-constructed as well as real world programs and that for 95% of the 4635 examined scripts we needed less than a minute to analyze each.

Contents

Chapter 1

Introduction

TouchDevelop [19], [11] is a novel programming language developed by Microsoft Research. It is geared toward development on mobile devices and targets lay programmers with only little experience in programming. Under these circumstances it is important that the programming environment supports the user during the development process (e.g. by pointing out possible errors in the source code).

The TouchBoost project aims to improve the programming experience of TouchDevelop by statically analyzing the performance and correctness of scripts. This thesis has been written in the context of the TouchBoost project and its goal is to design, formalize and implement a technique to statically track the contents of collections in TouchDevelop scripts. The technique is implemented as an extension to the static analyzer Sample [6], [7].

1.1 Motivation

Collections such as lists or maps are an integral part of the TouchDevelop standard library. The songs stored on a mobile device for example are represented as a collection of Song objects, the fields of a JSON object are represented as a map and games use sets to store obstacles. Almost every non-trivial script uses collections. When handling collections, the developer has to be cautious. If he for example accesses a map at an invalid key, or a list at an invalid index, it can cause his script to crash. We want to develop a static analysis that accurately warns the developer of potential run time errors caused by such invalid collection accesses or other potential errors that occur in scripts using collections. Our analysis determines all possible collections that can exist at each program point. Since a collection can potentially have an unbounded number of elements, we need to abstract away from individual elements. One simple approach to do this, is to abstract all elements of a collection by a single summary element. For arrays this approach is known as 'array smashing' [1]. In TouchDevelop, this often does not provide enough precision to prove the desired property and therefore leads to false warnings. To illustrate this, we are going to look at an example.

The TouchDevelop script in listing 1.1 shows a procedure that takes a String Map, which maps the name of a country to the name of its current president, as a parameter. In this example we want to print the name of the current president of the *USA*. But since the collection is passed as an argument, we don't know anything about its content. To ensure that no runtime error can occur, we first have to check that the key *USA* is contained in the map before we access it.

Our analysis should be able to prove that the collection access presented in this example is

Listing 1.1: Motivation example

```
1 action print_presidents(presidents: String_Map) {
2         if ($presidents->keys->contains("USA")) then {
3                 $presidents->at("USA")->post_to_wall();
4         } else {
5                 "unknwon"->post_to_wal();
6         }
7 }
```

safe. The collection smashing approach over-approximates the elements in a collection. In other words, it captures whether an element can be in a collection or not. If a collection at some program point might either contain the elements a and b or the elements b and c, then an approximation that uses smashing captures only that a, b or c can be in the collection. But it does not capture that b must be in the collection. For the presented example this means that the analysis is never able to say that the key *USA* must be in the map. Hence, it is also not able to prove that a collection access at that key is safe.

To be able to prove this property, we need an abstraction that under-approximates the elements contained in a collection. This means that the abstraction captures whether an element must be contained in a collection or not. If a collection at some program point can either contain the elements a and b or the elements b and c, the analysis needs to track that b certainly must be contained in the collection. We call such an analysis *Must Analysis*. In the presented example a *Must Analysis* would work as follow: At the beginning of the procedure we don't know anything about the content of the map. This means that we do not know about any key that it must be in the map. Hence, the abstraction of the map in the *Must Analysis* is empty. When the *then* branch is entered, we can assume that the key *USA* must be present in the list and we can add it to the abstraction. Since the collection access happens inside the *then* branch, the *Must Analysis* is able to prove that the key *USA* must be in the map and therefore that the collection access is safe.

The smashing approach also has other disadvantages. Since it is a very coarse abstraction, it is not able to distinguish individual collection elements. A script that accesses an individual element suffers from that imprecision. For our analysis we therefore use a more fine grained abstraction. Namely we summarize all collection elements that are added to a collection at the same allocation site and represent them with one abstract element. This abstraction has already been used in other analyses to abstract heap allocated structures [2].

Beside the *Must Analysis* we still want to be able to track which elements might be contained in a collection. This can for example be useful if we want to determine whether it is impossible that an element is contained in a collection. We therefore also present a second analysis called *May Analysis* which captures whether an element may be in a collection and summarizes collection elements based on the allocation site.

The TouchDevelop standard library offers a variety of different collections such as lists, maps or sets. Our technique shall be able to handle all these types of collections. We therefore use a high-level representation of collections for which we define basic semantic operations such as add or remove. The semantics of the different collection types (e.g. replacing the value of a key in a map) are then represented as a composition of these basic operations.

1.2 Outline

The rest of this thesis is structured as follow: In Chapter 2 we give an overview of TouchDevelop, Sample and the abstract interpretation framework. Chapter 3 introduces collections as they are provided by the TouchDevelop standard library, describes a high-level representation for these collections and formally defines the concrete domain of our analysis. In Chapter 4 we describe two abstractions for collection elements: The *May Analysis* that is able to tell which elements may be in a collection and the *Must Analysis* that can tell which elements must be contained in a collection. We define the concrete and the abstract semantics of collection operations and prove their soundness in Chapter 5. Chapter 6 gives an overview of how the analyses were implemented in Sample. To show how the analyses work we evaluated it on a set of hand constructed and real world TouchDevelop scripts. We also examine the precision and performance of the analysis when it is applied to a large number of real world scripts. This is described in Chapter 7. Finally, we provide our conclusion, present related work and describe how the analysis can be further improved in Chapter 8. Appendix A gives an overview of the mathematical expressions and notations used in this thesis.

Chapter 2

Preliminary Knowledge

2.1 TouchDevelop

TouchDevelop [11] is an application creation environment developed by Microsoft Research. It is centered around the idea to create mobile applications directly on the mobile device where it shall be used. Typically TouchDevelop scripts are written by students or hobbyist programmers who want to script their mobile devices but fear the effort of installing a complete programming environment. TouchDevelop scripts are usually rather short and often have less than 100 lines of code.

TouchDevelop comes with its own typed, structured programming language and a standard library. The standard library is designed to easily execute tasks typical to mobile applications. If for example offers interfaces to easily access the music library stored on the device or painlessly communicate with web services.

Developers can share their scripts using the TouchDevelop cloud. A developer can publish the source code of a script. The initially published version of a script is called a root script. Since a user can download a root script, modify it and re-publish it, there exist a lot of variations of the same root script. We use the TouchDevelop cloud to gain access to real world scripts. We can use those script to test our analysis. To ensure that we are not analyzing a lot of similar scripts, we only use root scripts.

2.1.1 Structure of a TouchDevelop Script

The logic of a TouchDevelop script is split among in actions and events. Actions are ordinary methods that can either be public or private. Events are event handlers that are called by the TouchDevelop runtime environment when for example a button has been pressed. Every public action can be the entry point of a script. This means that it is possible to directly run every public method of a script.

To persist data, the developer can use global variables. When a script terminates the values stored in global variables are persisted and restored the next time the script is started. If a script is executed for the first time, all global variables are initialized with invalid. The invalid value is a particularity of the TouchDevelop programming language. An object or a primitive value can be valid or invalid. Invalid objects and values can not be passed to library methods or stored in collections. Furthermore, it is not possible to call a method or to access a field of an invalid object. If a developer erroneously tries to do this, the script crashes.

2.2 Abstract Interpretation

A program can have infinitely many possible executions. If we want to make statements that hold for all possible executions (e.g. an element must be in a collection), we need to abstract away from all these executions and focus only on the properties that we are interested in (e.g. which elements must be in a collection). Abstract interpretation provides a generic framework to build sound abstractions. For this we need to define a concrete and an abstract domain, which both need to be complete lattices. We can abstract an element from the concrete domain with an element from the abstract domain. An abstract element can be concretized to an element in the concrete domain. In our case an element in the concrete domain is a set of concrete states and an element in the abstract domain is an abstract state.

For each statement in a programming language, the semantics for both the concrete and the abstract domain have to be defined. The concrete semantics transfers a set of concrete states to another set of concrete states. This is not computable, since in a program execution there can be infinitely many possible concrete states. The abstract semantics transfers an abstract state into another abstract state. The abstraction must be defined such that the abstract semantics is computable. The coarser the abstraction is the better an analysis performs since the abstract semantics is easier to compute but also the less properties can be shown, since more information is lost through the abstraction.

Multiple abstract domains can be combined with the cartesian product, where all operators are applied to both domains individually. The concretization of such a combined domain is the intersection of the concretizations of the two domains. We assume that the reader has a basic understanding of abstract interpretation. More information about abstract information can be found in [4], [5].

2.3 Sample

Sample [6], [7] is a generic static analyzer that has been developed at the Chair of Programming Methodology. It is based on the abstract interpretation framework [4], [5].

The analyzer is designed in a generic fashion which allows extensions for new domains and programming languages. Multiple analyses have already been implemented with Sample such as string [3], access permissions [8] or information leaking [16] analysis.

To get an overview of how Sample works we are going to show how the source code of a method is analyzed with Sample. The analysis process is depicted in figure 2.1.

Since Sample supports multiple programming languages, the source code of the method first has to be transformed into an intermediate format. This intermediate format is called Simple and represents the Control Flow Graph of the program. The transformation logic has to be implemented for each programming language individually. Currently translations for Scala and TouchDevelop exist. Based on the Control Flow Graph and the entry state of the method a least fixpoint computation is executed. The result of this computation is an annotated Control Flow Graph that contains for each statement the entry and the exit state at which the fixpoint was reached. Based on this the properties that shall be shown can be checked and warnings are created if a property could not be proven.

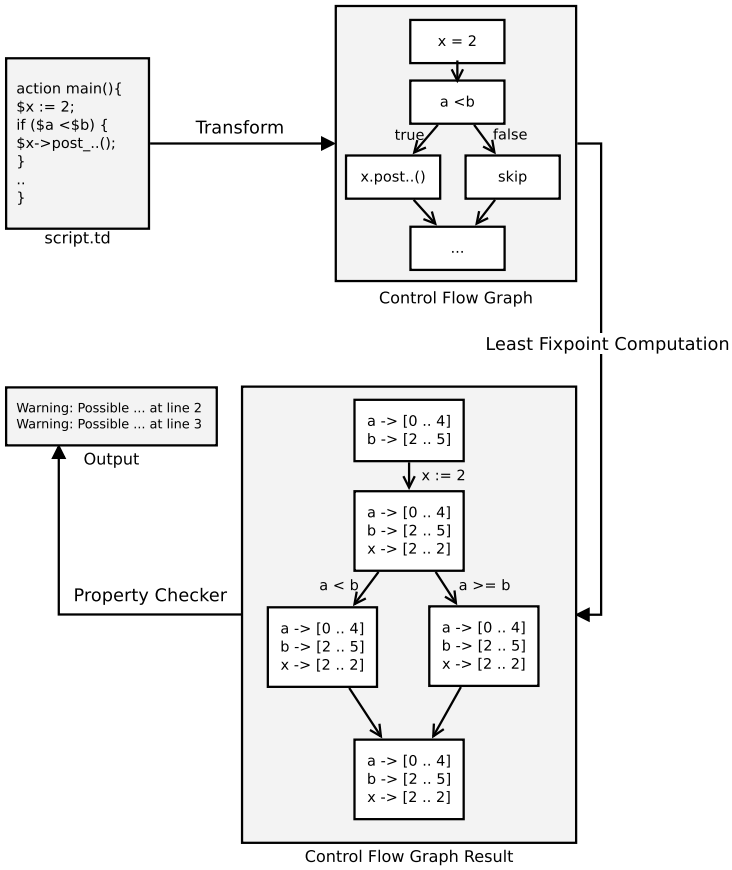

Figure 2.1: Overview of analysis process in Sample

2.3.1 Domains in Sample

An analysis is parameterized with a heap domain and a value domain. The heap domain abstracts the runtime heap structures of the analyzed script. The value domain abstracts the values in the program and is used to infer the desired property.

For the analysis of TouchDevelop scripts we use a value domain that is further decomposed into multiple domains to capture different value types.

- *Invalid Domain:* Captures whether an identifier is valid or invalid.
- *String Domain:* Captures the possible values of identifiers that are of type *String*. The currently used abstraction for strings is a k-set. This means that we track up to k different values for an identifier. If an identifier has more than k possible values, the value of that identifier is top (meaning it could have any possible string).
- *Numerical Domain:* Captures the possible values of identifiers that are of type *Number* or *Boolean*. Sample makes use of the APRON library [17] [12] to support relational numeric domains such as Octagons [15].

Figure 2.2 gives an overview of how the used domains are composed.

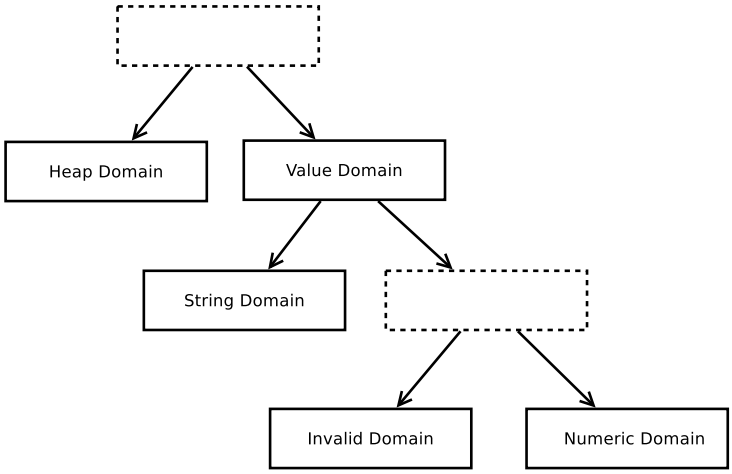

Figure 2.2: Domains used for TouchDevelop Analysis

2.3.2 Analysis of TouchDevelop Scripts

We are now going to look at how a TouchDevelop script is analyzed using Sample's least fix point computation engine.

Remember that TouchDevelop initializes all global variables with invalid, the first time a script is executed. Therefore we use an entry state for the analysis where all global variables are equal to invalid. Since all public actions can be an entry point of a script they are analyzed in parallel (meaning that all actions are analyzed with the same entry state). Since we don't know which action actually gets executed when a script is started, we have to take the least upper bound of all the exit states calculated for the individual actions. After an action has been executed, any of the events in a TouchDevelop script could be triggered. Therefore we have to analyze all the events in parallel as well and take the least upper bound of all their exit states. Because global variables are persisted and thus can have a different value every time a script is started, we need to repeat the analysis and use the retrieved state as the new entry state until until the global variable have reached a fix point. Once this fix point is reached the analysis is completed. Figure 2.3 gives an overview of the analysis process.

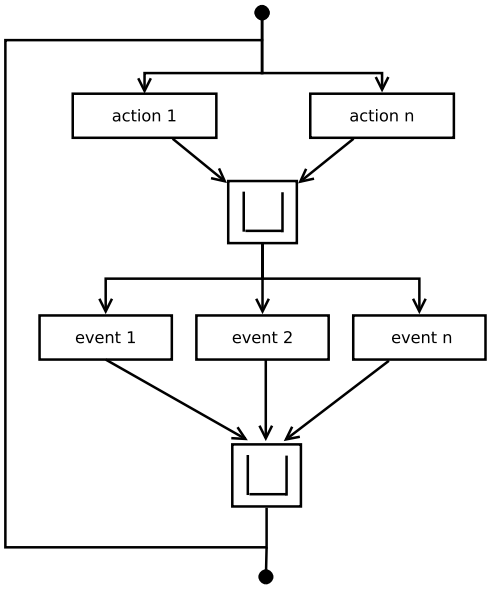

Figure 2.3: Overview of the analysis process for a TouchDevelop script

Chapter 3

Concrete Domain

This Chapter describes how the state of a TouchDevelop script is represented in our analysis. We will first give an overview of the available collection types in TouchDevelop such that we can then define a high-level representation for all these collection types. Finally we will formally define the concrete domain of our analysis based on this high-level representation.

3.1 Collections in TouchDevelop

This Section gives an overview of the collections available in TouchDevelop according to the API version 2.11 [20]. Furthermore we are going to show a few particularities about TouchDevelop that are important for our analysis. Notice that for simplicity we do not show all collection operations defined by the TouchDevelop but rather focus on the most important ones.

In TouchDevelop each collection has a specific type. A list of Songs for example is a SongCollection and a list of Messages is a MessageCollection. TouchDevelop does not support generic collections. All collections in TouchDevelop can be accessed by a key. Collections that don't naturally have a key (e.g. lists) can be accessed with an index, where the index represents the position of the element in the collection. For our analysis it is particularly important that an access to an invalid key or an out of bounds index does not lead to a runtime error. Instead an invalid object is returned. A runtime error only occurs if this invalid object is used wrongly (e.g. passed as an argument to another method).

Since we later want to define a high-level representation for all collections, we do not consider how exactly the collection operations are implemented. We rather focus on the contracts that a collection operation has to fulfill. This section serves as an overview, we will formally define the concrete semantics of the collection operations in chapter 5.

3.1.1 Categories of Collections

If we categorize collections based on the operations they offer and the contracts that these operations need to fulfill, we can distinguish three different categories of collections:

- *Lists :* Ordered sequences of elements. The same element can occur multiple times. A list can can either be mutable or read-only. Example: String collection
- *Sets :* Ordered collection of elements. The same element can occur only once. Sets in TouchDevelop are always mutable. Example: Sprite Set

- *Maps :* A map from keys to values. Maps in TouchDevelop are mutable. Example: String Map

Additionally there exist some objects like the JSON Object that are not collections in the classical sense but support collection like operations.

We will now analyze each of these categories more thoroughly.

Read-only Lists

The collections with the narrowest interface are read-only lists. Usually these collections are used to represent entities of the system (e.g. Song Collection, Contact Collection). Collections of this category can be accessed with a numeric linear index that starts at 0 and represents the position of an element in the list.

Notice that an access to an index that is out of bounds does not trigger a runtime error but rather returns an invalid object. Listing 3.1 shows a collection access which may return an invalid object.

Listing 3.1: List accesses that might return an invalid object

```
1 media->songs->at(0);
```

If the returned object is handled correctly as shown in listing 3.2, the program never triggers a run time error.

Listing 3.2: Invalid list access with check

```
1 $song := media->songs->at(0);
2 if (not $song->is_invalid()) then {
3         $song->play();
4 }
```

However, if the returned object is accessed unchecked as shown in listing 3.3 a runtime error occurs, if the collection access has returned an invalid object.

Listing 3.3: Potential runtime error

```
1 media->songs->at(0)->play();
```

Simple read-only lists can not be created by the developer but are obtained through library calls.

We can summarize the methods that are offered by this category of collections as follows:

- *at(index:Number):V* - Returns the element at the given index or invalid, if the index is out of bounds.
- *count:Number* - Returns the number of elements in the collection.

V denotes the type of the collection elements (e.g. for a SongCollection: V = Song)

Mutable Lists

As read-only lists, mutable list can also be accessed by a numeric linear index but additionally offer methods to alter the list's content.

The developer can instantiate mutable lists with the aid of the collections module. Listing 3.4 shows how a new empty *Link Collection* can be instantiated.

Listing 3.4: Create a Link collection

```
1 collections->create_link_collection()
```

Some mutable lists such as the *String* and the *Number Collection* also offer the possibility to check whether a value is contained in the list or not.

- *createCollection():T* - Creates a new list.
- *at(index:Number):V* - Returns the element at the given index or invalid, if the index is out of bounds.
- *count:Number* - Returns the number of elements in the collection.
- *add(item:V):Nothing* - Appends an item to the end of the list.
- *clear:Nothing* - Removes all elements from the list.
- *remove at(index:Number):Nothing* - Removes the element at the given index.
- *contains(item:V):Boolean* - Checks whether the given item is contained in the list or not.

V denotes the type of the list elements and T the type of the list itself (e.g. for a LinkCollection: V = Link, T = LinkCollection)

Sets

Sets are very similar to mutable lists. Their elements are ordered by the time of insertion and they offer the same operations as mutable lists. But in contrast to mutable lists, each element can only occur once. The elements can be accessed by a numerical linear index that starts at 0 and represents the position of an element in the set.

We can summarize the operations offered by sets as follows:

- *createCollection():T* - Creates a new set.
- *at(index:Number):V* - Returns the element at the given index or invalid, if the index is out of bounds.
- *count:Number* - Returns the number of elements in the collection.
- *add(item:V):Boolean* - Adds an element to the end of the set.
- *remove first:V* - Removes the first element from the set.
- *contains(item:V):Boolean* - Checks whether an element is contained in the list.

V denotes the type of the collection elements and T the type of the list itself (e.g. for a SpriteSet: V = Sprite, T = SpriteSet)

Maps

Maps are collections of key-value pairs where each key can only occur once. Since all maps are mutable they offer operations to add and remove key-value pairs as well as operations to access the value at a given key. Similar to mutable lists, new maps can be created by the developer using TouchDevelop's collections module.

As for lists, the access to a non-existing key does not raise a run-time error but rather returns an invalid object. The *Number Map*, however, is an exception to that. Instead of an invalid object it returns 0. For the formalization of the semantics we will ignore this special case, since it can be handled analogously to the standard access operation for maps.

We can summarize the operations provided by maps as follows:

- *createCollection():T* - Creates a new map.
- *at(key:K):V* - Returns the value stored at the given key or invalid if the key is not in the map.
- *count:Number* - Returns the number of elements in the map.
- *set at(key:K, value:V):Nothing* - Adds the value at the given key. If the provided key already exists, the existing value is replaced with the provided value.
- *remove(key:K):Nothing* - Removes the element at the given key. If the provided key is not in the map, the map is not altered.

K and V denote the types of the key and the value of the map and T denotes the type of the map itself (e.g. for a StringMap: K = String, V = String, T= StringMap)

Special Collections

Special collections are objects which offer collection-like operations but are not collections in the typical sense. An example of such an object is the *JSON Object*. It represents a JSON (JavaScript Object Notation) data structure. The fields of a *JSON Object* can be accessed using the *at* operation and the name of the field as a key. The result of this operation is another *JSON Object* (or invalid if no field with the given name exists). A *JSON Object* thus can be represented as a map from Strings to *JSON Objects*.

3.1.2 Foreach Loops

In TouchDevelop one can iterate over all elements in a collection using a *foreach* loop as shown in figure 3.5.

Listing 3.5: Foreach-Loop

```
1 var links := ...
2 links->add(web->link url("Link1", "http://www.link.ch"))
3 links->add(web->link url("Link2", "http://www.link.org"))
4
5 foreach e in links
6         where true
7 do
8         e->post to wall
```

The semantics of the *foreach* loop are as follows: The collection that shall be iterated over is cloned and then the cloned collection is traversed in the order specified by the collection. For maps the list of keys, ordered by the time of insertion, is traversed.

Where Clause

Additionally TouchDevelop offers the possibility to specify a where clause for a foreach loop as shown in Listing 3.6.

This is semantically the same as the code shown in listing 3.7.

Listing 3.6: Where clause

```
1 var links := ...
2 links->add(web->link url("Link1", "http://www.link.ch"))
3 links->add(web->link url("Link2", "http://www.link.org"))
4
5 foreach e in links
6         where e->name->equals("Link1")
7 do
8         e->post to wall
```

Listing 3.7: Foreach-loop with if statement

```
1 var links := ...
2 links->add(web->link url("Link1", "http://www.link.ch"))
3 links->add(web->link url("Link2", "http://www.link.org"))
4
5 foreach e in links
6         where true
7 do
8         if e->name->equals("Link1") then
9                 e->post to wall
10        else do nothing
```

3.1.3 Invalid Object

In TouchDevelop objects, Strings and Numbers can be invalid. If a collection is accessed with a key that is not in the domain the program does not crash but rather an invalid object is returned.

Invalid Objects as Arguments

If an invalid object is passed to a library method or the field or method of an invalid object is accessed the program crashes. Hence the code in Listing 3.8 results in a runtime error.

Listing 3.8: Invalid object as argument

```
1 var s := invalid->string
2 var strings := collections->create string collection
3 strings->add(s) //ERROR
```

This is important in the context of collections, as it means that a collection can never have an invalid object as a value.

3.2 High-level Representation of Collections

As we have seen, TouchDevelop offers different types of collections. Our analysis shall be able to handle all these collection types. Instead of defining an analysis for each collection type individually, we are going to introduce a high-level representation, that is able to represent all collection types presented in Section 3.1. This will allow us to define basic semantic operations for this representation (e.g. adding a new element), that we can use to build the semantics for the individual collection types.

The high-level representation describes how a collection is represented in memory. To introduce this representation we are going to use an example collection. We consider a *String Map s* that has two mappings: The key *Zurich* points to *ETHZ* and the key *Bern* points to *BFH*.

We look at a collection as a set of memory locations. At each memory location a pointer to a key and a pointer to a value is stored. We call these memory locations *Elements*. To tell which *Elements* belong to which collection, we keep a set of *Elements* for each collection. We call this set the *Element Set* of a collection.

Furthermore, we need to know what the content of a collection element is. More precisely we need to know what the content of its key and the content of its value is. We therefore keep a map which maps an *Element* and one of the keywords *key* or *value* to a value. A value is either a reference to an object or a primitive value (String, Number or Boolean). We call this map the *Value Map* of a collection.

For the running example this means that we represent the collection as a set with two *Elements* A and B and a *Value Map* $[(A, key) \rightarrow Zurich, (A, value) \rightarrow ETHZ, (B, key) \rightarrow Bern, (B, value) \rightarrow BFH]$. We can also draw this as a graph like in figure 3.1.

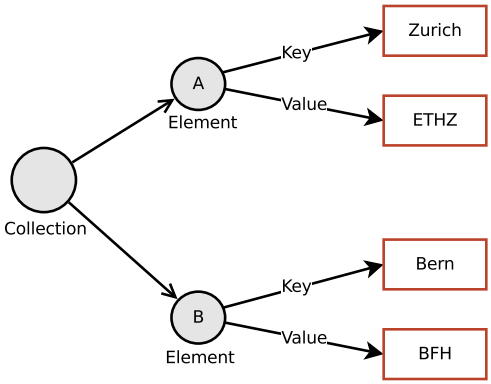

Figure 3.1: Representation of the String Map $[Zurich \rightarrow ETHZ, Bern \rightarrow BFH]$

A TouchDevelop script can have multiple collections. Each collection is identified by a single memory location that we call *Reference*. If the field of an object or a variable in the program is a collection, then it points to the *Reference* of the collection. This allows us to integrate the collection into the heap structure. Figure 3.2 shows an example where an object field and a variable both point to the same collection. Notice that we do not show the keys and values of the collection elements in this illustration. For simplicity reasons we will not include the representation of variables and the heap structure in the formal description of the concrete domain.

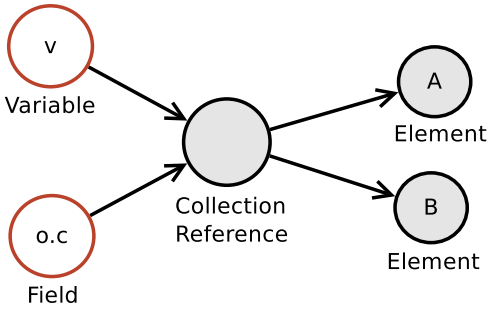

Figure 3.2: A collection that is pointed to by a variable and an object field

To track which *Element Set* belongs to which collection *Reference*, we use a map that maps collection *References* to *Element Sets*. We call this map the *Collection Environment*.

We have learned, that an *Element* is the memory location where the pointer to the key and the value of the collection element is stored. An *Element* therefore always belongs to exactly one collection. For this reason we can join the *Value Maps* of all the collections in a TouchDevelop script to one map, without clashes. We call this joined map the *Value Environment*.

In the running example we therefore would end up with the *Collection Environment* $[c \rightarrow \{A, B\}]$ and the *Value Environment* $[(A, key) \rightarrow Zurich, (A, value) \rightarrow ETHZ, (B, key) \rightarrow Bern, (B, value) \rightarrow BFH]$.

The high-level representation that we have described here, requires each collection type to define keys and values. In the previous section we have learned, that TouchDevelop distinguishes three categories of collections: Maps, Lists and Sets. A map naturally defines keys and values. For lists the key is a number and corresponds to the position of the element in the list and the value is the element itself. For sets the key is a number as well and corresponds to the position of the element in the insertion order. The value is the element itself.

3.3 Formal Definition of Concrete Domain

After we have described the intuition behind the high-level representation for collections, we can now formally define the concrete domain of our analysis based on this representation.

First we define Ref to be the set of all memory-locations in a TouchDevelop script. The set of all memory-locations used for collection *Elements* we name $Elem$.

$$Elem \subseteq Ref$$

We can then formally define an *Element Set* e as a set of *Elements*:

$$e \in \mathcal{P}(Elem)$$

where $\mathcal{P}(S)$ denotes the power set of S.

The *Collection Environment* that maps the *Reference* of a collection to it's *Element Set* is defined as a function from Ref to *Element Sets*:

$$Env_C : Ref \rightarrow \mathcal{P}(Elem)$$

The *Value Environment* maps an *Element* and one of the keywords *key* or *value* to a value. We define the set of possible values that can be assigned to the *key* or the *value* field of a collection

element as V. The *Value Environment* is then defined as:

$$Env_V : (Elem \times \{"key", "value"\}) \to V$$

The *Environment*, which in our analysis represents the concrete state of a program, is the cartesian product of the *Collection Environment* and the *Value Environment*:

$$Env : Env_C \times Env_V$$

When we statically analyze a program we have to consider all possible traces of the program. Considering all this possible traces, we use collecting semantics to collect for each program point all possibly reachable states. Our concrete state is defined as an *Environment*. Therefore, the lattice structure on which defines our concrete domain, is defined as follows:

$$< \mathcal{P}(Env), \subseteq, \emptyset, Env, \cup, \cap >$$

Chapter 4

Abstract Domains

Section 3 introduced, how collection elements are represented in the concrete domain. In this chapter we will now show two complementary abstractions for those collection elements.

The first abstraction captures if an element *might* be contained in a collection. We call this abstraction *May Analysis*. If we track for example the elements of a set and at a certain point in the program the set either contains the elements a, b or the elements b, c. Then the *May Analysis* needs to track that the elements a, b, and c might be in the collection. This analysis over-approximates the collection elements and can for example be used to prove that an element certainly is not contained in a collection.

The second abstraction tracks which elements certainly *must* be contained in a collection. We call this abstraction *Must Analysis*. If we use the same example as before where a set either contains the elements a, b or the elements b, c then the *Must Analysis* tracks that the element b certainly must be in the collection. Since this analysis under-approximates the elements in a collection and can for example be used to prove that a collection access at a given key always returns a valid object. For the *Must Analysis* we also define two different least upper bound operators. A standard least upper bound operator that, when joining two collections, only keeps the elements that are contained in both collections and a an extended least upper bound operator that uses a more involved operation to gain precision.

We define those two abstractions separately and uses the cartesian product to combine the the two analyses.

One problem that both abstractions need to solve is that collections can potentially have an unbounded number of elements. If a program for example consists of an endless loop, where an element is added to a collection inside the loop body, we end up with a collection that has infinitely many elements. We therefore need to abstract away from individual elements. ow heap structures are abstracted is known under the problem of shape analysis [13], [2]. We summarize all the collection elements that were added at the same allocation site and represent them with one single abstract collection element [2]. Since there exist different possibilities how one can abstract the individual collection elements, we designed the technique in such a way that this abstraction is exchangeable.

But not only the number of elements can be unbounded but also the number of collections itself. We therefore also abstract the *References* that represent the collections by the allocation site where they were created. This means that an abstract collection can represent multiple concrete collections. We call such collections summary collections. Summary collections produce new challenges which we will discuss in section 4.5.

Since we summarize multiple - potentially infinitely many - collection elements, we also need to be able to abstract the values of these collection elements. To do this we need a value domain that is able to handle summary identifiers. We do not define a specific value domain but rather describe the properties and semantic operators that it needs to have, in order to be used with our analysis. The specific value domain that we used is described as part of the implementation in Chapter 6. In general our technique becomes more precise if the value domain is more precise.

Before we formally define the two abstract domains for the *May* and the *Must Analysis*, we introduce the concepts described above more precisely.

4.1 Abstraction of References and Elements

In the concrete domain we have defined *References* as memory locations. The number of memory locations that are allocated in a TouchDevelop script can potentially be unbounded and we therefore need to abstract away from them. Similar to [7] we abstract multiple *References* with one *Heap Identifier*. We call the set of all *Heap Identifiers* in the abstract domain HId. In contrast to the set of *References*, the number of elements in the set of *Heap Identifiers* needs to be bounded to ensure that the analysis terminates. There are different approaches how *References* can be abstracted to *Heap Identifiers*. For this reason we have designed the technique in a way that this abstraction is exchangeable. A *Reference* abstraction must provide an abstraction function α_{HId} to convert a set of *Reference* to a set of *Heap Identifiers* and a concretization function γ_{HId} to convert a *Heap Identifier* to a set of *References*.

$$\alpha_{HId} : \mathcal{P}(Ref) \to \mathcal{P}(HId) \quad \gamma_{HId} : HId \to \mathcal{P}(Ref)$$

The abstraction we use abstracts *References* by allocation site. This is a well studied technique [2] and is already implemented in Sample as part of the heap abstraction. All *Elements* that were created at the same program point are abstracted by the same *Heap Identifier*.

Remember that an *Element* is a memory location as well. In fact we have defined the set of all *Elements* as a subset of the set of all *References*. We can therefore use the same abstraction for *Elements* as we use for *References*. This means that we summarize all *Elements* that were added to the collection at the same allocation site with one *Heap Identifier*.

4.2 Tuple Identifiers

We will see that the extended least upper bound operator, presented in Section 4.8.5, requires the ability to represent the fact that there must be an element in a collection that is abstracted by any of multiple possible *Heap Identifiers*. To describe such a collection element, we introduce a new type of identifier called *Tuple Identifier*. A *Tuple Identifier* is a set of *Heap Identifiers*. It abstracts all elements that are abstracted by any of the *Heap Identifiers* in that set. We call the set of all *Tuple Identifiers* TId.

$$TId : \mathcal{P}(HId)$$

4.2.1 Abstraction and Concretization of Tuple Identifiers

A *Tuple Identifier* abstracts multiple collection *Elements*. More precisely, a *Tuple Identifier* t abstracts all the *Elements* that are abstracted by all *Heap Identifiers* $h \in t$.

$$\gamma_{TId} : TId \to \mathcal{P}(Elem)$$
$$\gamma_{TId}(t) = \bigcup_{h \in t} \gamma_{HId}(h)$$

A set of collection *Elements* E is abstracted by the *Tuple Identifier* that represents the set of *Heap Identifiers* retrieved from $\alpha_{HId}(E)$.

$$\alpha_{TId} : \mathcal{P}(Elem) \to TId$$
$$\alpha_{TId}(E) = \alpha_{HId}(E)$$

4.3 Representation of Abstract Environment

After we have seen how single collection elements are abstracted, we can now look at how they are embedded in the abstract state.

An abstract collection element is represented as a *Tuple Identifier*. To track which *Tuple Identifiers* are in an abstract collection we keep a set of *Tuple Identifiers* $c^{(A)}$ which we call *Abstract Element Set*.

$$c^{(A)} \in \mathcal{P}(TId)$$

The *Abstract Element Set* is the counterpart of the concrete *Element Set* in the abstract domain.

Since there can be many collections in a TouchDevelop script, there can also exist multiple abstract collections. Analogous to the concrete domain we define an *Abstract Collection Environment* that maps *Heap Identifiers* to *Abstract Element Sets*.

$$Env_C^{(A)} : HId \to \mathcal{P}(TId)$$

To abstract the values of collection elements we use an *Abstract Value Environment* $env_V^{(A)} \in Env_V^{(A)}$. How the *Abstract Value Environment* looks like is defined by the value domain, which is a parameter of our analysis. The properties of the value domain are described in Section 4.4.

Finally the complete *Abstract Environment* is defined as the cartesian product of the *Abstract Collection Environment* and the *Abstract Value Environment*.

$$Env^{(A)} : Env_C^{(A)} \times Env_V^{(A)}$$

The *Abstract Environments* will be used as the elements of the lattices that define the abstract domains of the *May* and *Must Analysis*. We will however see that the lattice operators are defined differently for the two domains.

4.4 Value Domain

The abstraction of the values of collection elements influences the precision of the analysis. Since primitive values and object references can be abstracted in many different ways, we designed the analysis such that the abstraction of the *Value Environment* is easily exchangeable. We however require certain properties that the value domain needs to fulfill. Particularly it needs to define lattice operators, abstraction and concretization functions and a set of semantic operations.

4.4.1 Abstraction and Concretization functions

A value abstraction needs to define a concretization function γ_V that concretizes an *Abstract Value Environment* to a set of *Value Environments* and an abstraction function α_V that does the opposite.

$$\alpha_V : \mathcal{P}(Env_V) \to Env_V^{(A)}$$
$$\gamma_V : Env_V^{(A)} \to \mathcal{P}(Env_V)$$

4.4.2 Lattice operators

The value domain needs to implement the lattice operators for the *Abstract Value Environments*. Particularly it needs to provide a least upper bound and a greatest lower bound operator.

$$\sqcup_V : \left(Env_V^{(A)} \times Env_V^{(A)}\right) \to Env_V^{(A)}$$
$$\sqcap_V : \left(Env_V^{(A)} \times Env_V^{(A)}\right) \to Env_V^{(A)}$$

4.4.3 Semantic Operators

A value domain needs to offer a few semantic operators for the concrete and the abstract domains. In this section we will describe the interfaces for these operators. We assume that all these operations are sound, meaning that the concrete semantics is over-approximated by the abstract semantics.

Assign

Assigns an expression to an identifier. An expression is a construct that needs to be provided by the value domain. It is either a value, an identifier, an arithmetic operator or a comparison operation. We need to assign expressions instead of values to an identifier, to be able to make use of relational value domains. Expressions allow us to represent relations between identifiers. We can for example say, that the key of a collection element is equal to its value. How expressions are evaluated depends on the used value domain.

$$assign_V : ((Elem \times \{"key", "value"\}) \times V \times Env_V) \to Env_V$$
$$assign_V^{(A)} : \left((TId \times \{"key", "value"\}) \times Expression \times Env_V^{(A)}\right) \to Env_V^{(A)}$$

Assume

Assumes that a given expression holds in a the *Value Environment*. If a provided assumption can never be true in the provided *Value Environment*, the returned *Value Environment* is bottom.

$$assume_V : (Expression \times Env_V) \to Env_V$$
$$assume_V^{(A)} : \left(Expression \times Env_V^{(A)}\right) \to Env_V^{(A)}$$

Equals

With this function we can check whether an identifier is equal to a given expressions in a *Value Environment*.

$$equals_V : ((Elem \times \{"key","value"\}) \times V \times Env_V) \to \{true, false\}$$

Since an *Abstract Value Environment* represents multiple concrete *Value Environments*, the abstract semantics of $equals_V$ returns a set of boolean values instead of a single boolean value. They have the following meaning:

- If the identifier must be equal to the expression in all *Value Environments* abstracted by the provided *Abstract Value Environment*, $\{true\}$ is returned.
- If the identifier can not be equal to the expression in an *Value Environments* abstracted by the provided *Abstract Value Environment*, $\{false\}$ is returned.
- If the identifier is equal to the expression in some bot not in all *Value Environments* abstracted by the provided *Abstract Value Environment*, $\{true, false\}$ is returned.

$$equals_V^{(A)} : \left((TId \times \{"key","value"\}) \times Expression \times Env_V^{(A)}\right) \to \mathcal{P}(\{true, false\})$$

Replace

This operation is used to assign to an identifier the least upper bound of the values of a set of other identifiers. More details about this operation and the used Replacement structure will be explained in Section 4.6.

$$replace_V : \left(Env_V^{(A)} \times Replacement\right) \to Env_V^{(A)}$$

Bottom Check

We need a function for the abstract value domain that checks whether a given identifier is bottom or not.

$$isBottom_V^{(A)} : \left((TId \times \{"key","value"\}) \times Env_V^{(A)}\right) \to \{true, false\}$$

4.5 Summary Collections

The *Abstract Collection Environment* maps *Heap Identifiers* to *Abstract Element Sets*. Remember that a *Heap Identifier* can abstract multiple *References*. If a *Heap Identifier*, that represents a collection, in the domain of the *Abstract Collection Environment* abstracts multiple *References*, the abstract collection does no longer represent one collection in the script but rather multiple ones. Such an abstract collection is called a summary collection.

Formally, we say an abstract collection, that is represented by the *Heap Identifier* $c^{(A)}$, is a summary collection, if $|\gamma_{HId}\left(c^{(A)}\right)| > 1$.

Listing 4.1 shows an example which leads to a summary collection. In this example the two different collections are created at the same allocation site. The two *References* that represent the concrete collections are abstracted by one *Heap Identifier* $c^{(A)}$. This means that the collection, represented by $c^{(A)}$ is a summary collection.

Listing 4.1: Create collections with a factory method

```
1 $c := code->create_collection();
2 $d := code->create collection();
3
4 action create_collection(){
5         return collections->create_link_collection();
6 }
```

Semantic operations that involve a summary collection have to consider that a summary collection represents multiple collections. To illustrate this let's look at an example: Assume that we add an element to collection c in the script in listing 4.1. In the abstract domain the collections c and d are represented as one summary collection. The element however is added only to c. In the *Must analysis* the collection only contains the elements that certainly are in a collection. It would therefore not be sound for the abstract semantics to add the element to the summary collection, as it is not contained in d. More details on how summary collections must be treated will be shown when the abstract semantics are defined.

To simplify the presentation here, we do not consider summary collections when defining the abstract domain.

4.5.1 Detection of summary collections

Whether an identifier $c^{(A)}$ represents a summary collection or not can be determined by checking if $\gamma_{HId}\left(c^{(A)}\right)$ contains more than one element. But since a summary collection can abstract potentially infinitely many collections, $\gamma_{HId}\left(c^{(A)}\right)$ can also have infinitely many elements, which makes it uncomputable. Hence, we need an alternative way to determine whether a collection is a summary collection or not. Sample uses a technique to identify summary *Heap Identifiers* by checking if a *Heap Identifier* is created more than once in a script. Since a collection is represented in the *Abstract Collection Environment* as a *Heap Identifier*, we can directly apply this technique to determine whether a collection is a summary collection or not.

For the formalization we therefore define an operation $isSummary$ that can determine whether an identifier is a summary identifier or not.

$$isSummary : (HId \cup TId) \rightarrow \{true, false\}$$

4.6 Replacements

Some operations in the *Abstract Collection Environment* remove or replace *Heap Identifiers*. This information also needs to be reflected in the state of of the *Abstract Value Environment*. To pass the information which *Heap Identifiers* need to be replaced from the *Abstract Collection Environment* to the *Abstract Value Environment* we are using *Replacements*. *Replacements* were first introduced by Ferrara et. al. [7].

A *Replacement* is a map from sets of *Heap Identifiers* to sets of *Heap Identifiers*:

$$Replacement : \mathcal{P}(HId) \rightarrow \mathcal{P}(HId)$$

An entry in this map might be $\{hId_1, hId_2\} \to \{hId_3, hId_4\}$. This means hat hId_1 and hId_2 are replaced by hId_3 and hId_4. We call the left set of the entry ($\{hId_1, hId_2\}$) the *from* set and the right set ($\{hId_3, hId_4\}$) the *to* set.

A value domain needs to define a function $replace_V$ that performs the replacement by assigning to all *Heap Identifiers* in the *to* set the least upper bound of the values of *Heap Identifiers* in the *from* set for all entries in the *Replacement* map.

$$replace_V : \left(Env_V^{(A)} \times Replacement\right) \to Env_V^{(A)}$$

For our example this would mean that the least upper bound of the values of $\{hId_1, hId_2\}$ is assigned to hId_3 as well as to hId_4. The formal description of *Replacements* and the replace function can be found in [7].

For a functional domain, that maps *Heap Identifiers* to values, the $replace_V$ function can be implemented as follows: For each entry in the *Replacement* with a *from* set F and a *to* set T, we apply the function r, that replaces the values of all *Heap Identifiers* in the *to* set with the least upper bound of the values of all *Heap Identifiers* in the *from* set.

$$r : (Env_V \times \mathcal{P}(HId) \times \mathcal{P}(HId)) \to Env_V$$

$$r(env, F, T) = \left[\begin{array}{ll} h \to v : & h \in (dom(env) - F) \cup T \wedge \\ & v = \begin{cases} \bigsqcup_{h' \in F} env(h') & \text{if } h \in T \\ env(h) & \text{otherwise} \end{cases} \end{array}\right]$$

For relational value domains the implementation of the $replace_V$ operation is more involved.

4.6.1 Joining Replacements

We will need to be able to join multiple *Replacements* to one *Replacement* to easier pass it to other domains. For this operation it is important to know that all the entries in a Replacement map are executed in parallel. This means that they are all executed on the same state. When we join two *Replacements* we still want to execute them in parallel. Hence, the join operation $\oplus$ on *Replacements* simply joins the two maps of the *Replacements*.

$$\oplus : Replacement \times Replacement \to Replacment$$

$$rep_1 \oplus rep_2 = \left[\begin{array}{ll} t \to f : & (t \in dom(rep_1) \wedge f = rep_1(r)) \vee \\ & (t \in dom(rep_2) \wedge f = rep_2(t)) \end{array}\right]$$

In an earlier version of the formalization we used an operation to concatenate Replacements. This operation is no longer used, but because it might be of general interest its formalization can be found in Appendix C.

4.7 May Analysis

We now define the abstract domain for the *May Analysis*. The *May Analysis* tracks which elements might be in a collection.This means that if we abstract a set of concrete collections the resulting abstract collections should contain all elements that are in any of the concrete collections.

On the other hand, each concrete collection that is abstracted by an abstract collection, can have no other elements than the elements contained in the abstract collection. It however does

not need to have all the elements of the collection. We can therefore say that the concretization of an abstract collection consists of all concrete collections that contain a subset of the elements in the abstract collection.

When joining two abstract collections, the resulting abstract collection should abstract all concrete collections that were abstracted by either of the two joined abstract collection.

In this section we will formally describe how this intuition can be applied for our technique and how we can use it to describe the abstract domain for the *May-Analysis*.

4.7.1 Abstract Domain

The abstract domain is defined as a lattice structure. The elements of this lattice are *Abstract Environments* ($Env^{(A)}$) that were defined earlier.

$$< Env^{(A)}, \sqsubseteq_{May}, \bot_{May}, \top_{May}, \sqcup_{May}, \sqcap_{May} >$$

Intuitively we can say that an abstract collection in the *May Analysis* abstracts less concrete collections, if it contains less elements. Formally we have do define the partial ordering operator of the lattice for the complete *Abstract Collection Environment*.

Partial Ordering

Since an *Abstract Environment* is the cartesian product of the *Collection Environment* and the *Value Environment*, the partial ordering among *Abstract Environments* is defined as the typical partial ordering relation for cartesian product domains.

$$\begin{gathered}\left(envL_C^{(A)}, envL_V^{(A)}\right) \sqsubseteq_{May} \left(envR_C^{(A)}, envR_V^{(A)}\right) \\ \Leftrightarrow \\ \left(envL_C^{(A)} \sqsubseteq_C^{May} envR_C^{(A)}\right) \wedge \left(envL_V^{(A)} \sqsubseteq_V envR_V^{(A)}\right)\end{gathered}$$

The partial ordering relation for the *Abstract Value Environment* $\sqsubseteq_V$ is defined by the value domain but the partial ordering relation for the *Abstract Collection Environment* $\sqsubseteq_C^{May}$ needs to be defined by our analysis. Since the *Abstract Collection Environment* is a functional domain its partial ordering operator is defined as the standard partial ordering relation for functional domains.

$$envL_C^{(A)} \sqsubseteq_C^{May} envR_C^{(A)} \Leftrightarrow \forall h \in dom\left(envL_C^{(A)}\right) : envL_C^{(A)}(h) \sqsubseteq_E^{May} envR_C^{(A)}(h)$$

The partial ordering among *Abstract Element Sets* $\sqsubseteq_E^{May}$ is defined as follows:

$$eL^{(A)} \sqsubseteq_E^{May} eR^{(A)} \Leftrightarrow eL^{(A)} \subseteq eR^{(A)}$$

Top and Bottom

Similar to the partial ordering operator, the top and bottom elements of the abstract domain are the standard operators for the respective domains. We therefore only show the definition of the top and bottom elements for the *Element Sets*. The top element is the set containing all *Tuple Identifiers* whereas the bottom element is the empty set.

$$\top_E^{May} = TId \quad \bot_E^{May} = \emptyset$$

4.7.2 Concretization function

The concretization function converts an *Abstract Environment* into a set of *Environments*.

Since the *Abstract Environment* is the cartesian product of the *Abstract Collection Environment* and the *Abstract Value Environment* we decompose the concretization function into the concretization functions of those two environments.

$$\gamma_{May} : Env^{(A)} \to \mathcal{P}(Env)$$

$$\gamma_{May}\left(env^{(A)}\right) = \gamma_{May}\left(env_C^{(A)}, env_V^{(A)}\right) = \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right) \times \gamma_V\left(env_V^{(A)}\right)$$

We have already seen that the value domain is exchangeable and that each value domain therefore has to provide a concretization function γ_V. It takes an *Abstract Value Environment* and concretizes it to a set of *Value Environments*.

The concretization function for the collection environment γ_C^{May} however is defined by our analysis. Remember that the *Abstract Collection Environment* is a map from *Heap Identifiers* to sets of *Tuple Identifiers*. Each entry in that map corresponds to one abstract collection.

Let's look at one such entry $h \to c^{(A)}$. The *Heap Identifier* h is an abstraction of one or more collection *References* as described in section 4.5. The set of *References* that are abstracted by h is obtained by $\gamma_{HId}(h)$. The *Abstract Element Set* $c^{(A)}$, which is a set of *Tuple Identifiers*, abstracts one or multiple *Element Sets*. For each *Tuple Identifier* t in $c^{(A)}$, the *value* and the *key* fields are tracked in the *Abstract Value Environment*. It is possible that the values of those fields in the *Abstract Value Environment* are bottom. We will later see how this can occur as a result of the abstract semantics of the remove operation. If the key or the value field of t is bottom in the *Abstract Value Environment*, then the abstract collection element, represented by t, has either no key or no value. This is the same as if t would not be in the *Abstract Element Set*. We therefore need to filter out all the *Tuple Identifiers* from the *Abstract Element Set* that have either no key or no value. We do this by checking for each *Tuple Identifier* t in the *Abstract Element Set*, that neither the *key* nor the *value* field is equal to bottom in the *Abstract Value Environment*. We name the set of filtered *Tuple Identifiers* of an *Abstract Element Set* $c^{(a)}$: $E^{c^{(A)}}$. The set of *Element Sets* that are abstracted by $c^{(A)}$ is therefore obtained by $\gamma_E^{May}\left(E^{(c^{(A)})}\right)$. Each of the collection *References* might point to any of the *Element Sets*. We therefore consider all the combinations of *References* in $\gamma_{HId}(h)$ with *Element Sets* in $\gamma_E^{May}\left(E^{(c^{(A)})}\right)$.

Since this needs to be done for all entries in the *Abstract Collection Environment* we take the union over all the sets created for each entry. The result of this is a set with all possible mappings from collection *References* to *Element Sets*. The *Collection Environments* represented by the given *Abstract Collection Environment* are then all functions that are built from all possible subsets of this set.

$$\gamma_C^{May} : Env_C^{(A)} \rightarrow \mathcal{P}(Env_C)$$

$$\gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right) = \left\{ [r \rightarrow c : (r,c) \in x] : \quad x \subseteq \bigcup_{h \in dom\left(env_C^{(A)}\right)} \left\{ (r',c') : \begin{array}{l} r' \in \gamma_{HId}(h) \wedge \\ c' \in \gamma_E^{May}(E) \wedge \\ E = \left\{ \begin{array}{l} t : \\ t \in env_C^{(A)}(h) \wedge \\ notBottom\left(t, env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right\}$$

The function *notBottom* checks that neither the key nor the value field of a tuple t is equal to bottom in the *Value Environment*.

$$notBottom : \left(TId \times Env_V^{(A)}\right) \rightarrow \{true, false\}$$

$$notBottom\left(t, env_V^{(A)}\right) = \neg isBottom_V^{(A)}\left((t, \text{"}key\text{"}), env_V^{(A)}\right) \wedge \neg isBottom_V^{(A)}\left((t, \text{"}value\text{"}), env_V^{(A)}\right)$$

What is now left to define is how an *Abstract Element Set* is concretized to a set of *Element Sets*. This concretization is described by the function γ_E^{May}. Remember that in the *May Analysis* an abstract collection abstracts the information which elements *may* be in a collection. Also remember that with the *Element Set* we represent the information which elements are contained in the collection. The *Element Sets* represented by an *Abstract Element Set* are therefore all the subsets of the *Abstract Element Set*.

$$\gamma_E^{May} : \mathcal{P}(TId) \rightarrow \mathcal{P}(\mathcal{P}(Elem))$$

$$\gamma_E^{May}\left(c^{(A)}\right) = \left\{ x : x \subseteq \bigcup_{t \in c^{(A)}} \gamma_{HId}(t) \right\}$$

4.7.3 Abstraction function

The abstraction function abstracts a set of *Environments* to an *Abstract Environment*.

Since an abstract in the *May Analysis* contains all elements that possibly might be in a collection, it can be built by taking the union of all elements contained in the concrete collections that shall be abstracted.

Formally, we need to define the abstraction function for the complete environment.

Since an *Environment* is the cartesian product of a *Collection Environment* and a *Value Environment* we decompose the abstraction function into the abstraction functions of those two environments.

$$\alpha_{May} : \mathcal{P}(Env) \rightarrow Env^{(A)}$$

$$\alpha_{May}(env) = \alpha_{May}(env_C, env_V) = \left(\alpha_C^{May}(env_C), \alpha_V(env_V)\right)$$

The abstraction function for the value environment α_V is defined by the value domain. It abstracts a set of *Value Environments* to an *Abstract Value Environment*. The abstraction function for the collection environment α_C^{May} however is defined by our analysis. It abstracts a set of *Collection Environments* to an *Abstract Collection Environment*.

The *Abstract Collection Environment* is a function from *Heap Identifiers* to *Abstract Element Sets*. The domain of the *Abstract Collection Environment* function is the set of all the *Heap Identifiers* that abstract the collection *References* in the domains of the *Collection Environments*. The *Abstract Element Set* to which such a *Heap Identifier* points to abstracts all the *Element Sets* to which the collection *References*, that where abstracted by that *Heap Identifier*, point to in any *Collection Environment*.

$$\alpha_C^{May} : \mathcal{P}(Env_C) \to Env_C^{(A)}$$

$$\alpha_C^{May}(envs_C) = \left[\begin{array}{lll} h \to c^{(A)} : & h \in \alpha_{HId}(\{r : r \in dom(env_C) \wedge env_C \in envs_C\}) \wedge \\ & c^{(A)} = \alpha_E^{May}\left(\left\{ \begin{array}{ll} env_C(r) : & r \in dom(env_C) \wedge \\ & \alpha_{HId}(\{r\}) = h \wedge \\ & env_C \in envs_C \end{array} \right\}\right) \end{array} \right]$$

A set of *Element Sets* is abstracted by a single *Abstract Element Set*. Remember that the *May Analysis* abstracts multiple collections by telling which elements may be in the collection. Every *Element* that is in one of the provided *Element Sets* might be in the collection. Therefore the *Abstract Element Set* contains all *Tuple Identifiers* that abstract a collection *Element* which is in any of the provided *Element Sets*.

$$\alpha_E^{May} : \mathcal{P}(\mathcal{P}(Elem)) \to \mathcal{P}(TId)$$

$$\alpha_E^{May}(C) = \bigcup_{R \in C} \alpha_{HId}(R)$$

4.7.4 Least upper bound

Intuitively the least upper bound of two abstract collections for the *May Analysis* is an abstract collection that represents all the elements that may be in any of the concrete collections that are abstracted by one of the two abstract collections.

Formally we need to define the least upper bound for two *Abstract Environments*. For this we again decompose the least upper bound operator into the least upper bound operator of the *Abstract Value Environment* $\sqcup_V$ and the least upper bound operator of the *Abstract Collection Environment*.

$$\sqcup_{May} : \left(Env^{(A)} \times Env^{(A)}\right) \to Env^{(A)}$$

$$\begin{aligned} envL \sqcup_{May} envR &= (envL_C, envL_V) \sqcup_{May} (envR_C, envR_V) \\ &= \left(envL_C \sqcup_C^{May} envR_C, envL_V \sqcup_V envR_V\right) \end{aligned}$$

The least upper bound operator for the *Abstract Value Environment* $\sqcup_V$ is provided by the value domain that is used.

The least upper bound operator for the *Abstract Collection Environment* however is defined by our analysis.

Since the *Abstract Collection Environment* is a function from *Heap Identifiers* to *Abstract Element Sets* we define the least upper bound operator as the standard least upper bound operator used for functional domains: Let's call the two *Abstract Collection Environments* which are the operands of the least upper bound operator *left* and *right*. The least upper bound operator builds a new *Abstract Collection Environment* whose domain is the union of the domains of the

left and the *right Abstract Collection Environment*. A *Heap Identifier* h in the domain of that new *Abstract Collection Environment* points to:

- The *Abstract Element Set* that h points to in the left *Abstract Collection Environment*, if h only occurs in the domain of the left *Abstract Collection Environment*.
- The *Abstract Element Set* that h points to in the right *Abstract Collection Environment*, if the h only occurs in the domain of the right *Abstract Collection Environment*.
- The union of the *Abstract Element Set* that h points to in the left *Abstract Collection Environment* and the one that h points to in the right *Abstract Collection Environment*, if h occurs in the domains of the left and the right *Abstract Collection Environments*.

$$\sqcup_C^{May} : \left(Env_C^{(A)} \times Env_C^{(A)}\right) \to Env_C^{(A)}$$

$$envL_C \sqcup_C^{May} envR_C = \left[\begin{array}{l} h \to c^{(A)} : \\ \quad h \in dom(envL_C) \cup dom(envR_C) \wedge \\ \quad c^{(A)} = \begin{cases} envL_C(h) & \text{if } h \in dom(envL_C) - dom(envR_C) \\ envR_C(h) & \text{if } h \in dom(envR_C) - dom(envL_C) \\ envL_C(h) \cup envR_C(h) & \text{otherwise} \end{cases} \end{array}\right]$$

Notice that if an abstract collection occurs in both *Abstract Collection Environments*, we take the union of the two *Abstract Element Sets*. This is due to the fact that the least upper bound of two abstract collections needs to represent all the elements that might be in one of the collections abstracted by both abstract collections that are joined.

4.8 Must Analysis

We can now define the second abstract domain: The *Must Analysis*. In contrast to the *May Analysis* the *Must Analysis* does not capture which elements might be contained in a collection, but rather which elements certainly *must* be in the collection.

However, we will see that the Environments for both Analyses are abstracted in the same way. Only the abstraction of the *Elements Set* of a concrete collection is defined differently than in the *May Analysis*.

For the abstraction and the concretization functions, it is therefore sufficient to replace the γ_E^{May} and the α_E^{May}. We however have to be careful how we define the lattice operators, since they are different for the *Must Analysis*.

For the abstraction and the concretization functions, it is therefore sufficient to replace the γ_E^{May} and the α_E^{May}. We however have to be careful how we define the lattice operators, since they are different for the *Must Analysis*.

Intuitively we can say that if we abstract a set of concrete collections, the resulting abstract collection should contain only the elements that are contained in all concrete collections. In other words the collections represented by an abstract collection are all the concrete collections that contain a superset of the elements in the abstract collection.

4.8.1 Abstract Domain

As in the *May Analysis* the elements of the lattice structure for the abstract domain of the *Must Analysis* are *Abstract Environments* ($Env^{(A)}$). However the lattice operators are defined differently.

$$< Env^{(A)}, \sqsubseteq_{Must}, \bot_{Must}, \top_{Must}, \sqcup_{Must}, \sqcap_{Must} >$$

Intuitively an abstract collection in the *Must Analysis* abstracts less concrete collections, the more elements it contains. Formally we again have to define the partial ordering operator of the lattice for the complete *Abstract Environment*.

Partial Ordering

Except for the partial ordering of *Abstract Element Sets* $\sqsubseteq_E^{May}$, the partial ordering operator of *Abstract Environments* for the *Must Analysis* is defined analogous to the partial ordering operator for the *May Analysis*.

The partial ordering among *Abstract Element Sets* $\sqsubseteq_E^{Must}$ is defined as follows:

$$eL^{(A)} \sqsubseteq_E^{Must} eR^{(A)} \Leftrightarrow eR^{(A)} \subseteq eL^{(A)}$$

Top and Bottom

Similar to the partial ordering operator, the top and bottom elements are defined analogously to the operators in the *May Analysis*. Only the top and bottom elements for the *Element Sets* are defined differently. The top element is the empty set, whereas the bottom element is the set containing all possible *Tuple Identifiers*.

$$\top_E^{Must} = \emptyset \quad \bot_E^{Must} = TId$$

4.8.2 Concretization function

The concretization function converts an *Abstract Environment* into a set of *Concrete Environments*.

Since the *Abstract Environment* is the cartesian product of the *Abstract Collection Environment* and the *Abstract Value Environment* we decompose the concretization function into the concretization functions of these two environments.

$$\gamma_{Must} : Env^{(A)} \rightarrow \mathcal{P}(Env)$$

$$\gamma_{Must}\left(env^{(A)}\right) = \gamma_{Must}\left(env_C^{(A)}, env_V^{(A)}\right) = \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right) \times \gamma_V\left(env_V^{(A)}\right)$$

We have already seen that the value domain is exchangeable and that each value domain therefore has to provide a concretization function γ_V. It takes an *Abstract Value Environment* and concretizes it to a set of *Value Environments*.

The concretization function for the collection environment γ_C^{Must} however is defined by our analysis.

The concretization function of the collection environment for the *Must Analysis* is the same as the one for the *May Analysis*.

$$\gamma_C^{Must} : Env_C^{(A)} \to \mathcal{P}(Env_C)$$

$$\gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right) = \left\{ \begin{array}{l} [r \to c : (r,c) \in x] : \\ \\ x \subseteq \bigcup\limits_{h \in dom\left(env_C^{(A)}\right)} \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}(h) \wedge \\ & c' \in \gamma_E^{Must}(E) \wedge \\ & E = \left\{ \begin{array}{l} t : \\ t \in env_C^{(A)}(h) \wedge \\ notBottom\left(t, env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right\}$$

However, the *Abstract Element Set* must be concretized differently in the *Must Analysis*. Remember that in the *Must Analysis* an abstract collection abstracts all the concrete collections that at least contain the elements that are in the abstract collection. Also remember that with the *Element Set* we represent the information which elements are contained in the collection. The *Element Sets* represented by an *Abstract Element Set* are therefore all the supersets of the concretized *Tuple Identifiers* in the *Abstract Element Set*.

$$\gamma_E^{Must} : \mathcal{P}(TId) \to \mathcal{P}(\mathcal{P}(Elem))$$

$$\gamma_E^{Must}\left(c^{(A)}\right) \left\{ x : x \supseteq \bigcup_{t \in c^{(A)}} \gamma_{HId}(t) \right\}$$

4.8.3 Abstraction function

The abstraction function abstracts a set of *Environments* to an *Abstract Environment*.

Since the *Environment* is the cartesian product of the *Collection Environment* and the *Value Environment* we decompose the abstraction function into the abstraction functions of those two environments.

$$\alpha_{Must} : \mathcal{P}(Env) \to Env^{(A)}$$

$$\alpha_{Must}(env) = \alpha_{Must}(env_C, env_V) = \left(\alpha_C^{Must}(env_C), \alpha_V(env_V)\right)$$

The abstraction function for the value environment α_V is defined by the value domain. It abstracts a set of *Value Environments* to an *Abstract Value Environment*.

The abstraction function for the collection environment α_C^{Must} however is defined by our analysis. It abstracts a set of *Collection Environments* to an *Abstract Collection Environment*.

The *Collection Environments* are abstracted similarly as in the *May Analysis*.

$$\alpha_C^{Must} : \mathcal{P}(Env_C) \to Env_C^{(A)}$$

$$\alpha_C^{Must}(envs_C) = \left[\begin{array}{ll} h \to c^{(A)} : & h \in \alpha_{HId}(\{r : r \in dom(env_C) \wedge env_C \in envs_C\}) \\ & c^{(A)} = \alpha_E^{Must}\left(\left\{ \begin{array}{ll} env_C(r) : & r \in dom(env_C) \wedge \\ & \alpha_{HId}(\{r\}) = h \wedge \\ & env_C \in envs_C) \end{array} \right\}\right) \end{array} \right]$$

However, the set of *Element Sets* must be abstracted differently. A set of *Element Sets* is abstracted by a single *Abstract Element Set*. Remember that the *Must Analysis* abstracts

multiple collections by telling which elements must be in the collection. If an *Element* is in all the *Element Sets* it must be in the collection and therefore needs to be represented the *Abstract Element Set*. Hence, the *Abstract Element Set* is the intersection over all the *Element Sets*.

$$\alpha_E^{Must} : \mathcal{P}(\mathcal{P}(Elem)) \to \mathcal{P}(TId)$$

$$\alpha_E^{Must}(C) = \bigcap_{R \in C} \alpha_{HId}(R)$$

4.8.4 Least upper bound

Intuitively the least upper bound of two abstract collections for the *Must Analysis* is an abstract collection that represents all the elements that must be in all the collections represented by the two abstract collections.

Formally we need to define the least upper bound operator for two *Abstract Environments*.

We will see later, that in least upper bound of two *Abstract Collection Environments* some *Tuple Identifiers* are replaced by new *Tuple Identifiers*. These changes also have to be communicated to the *Abstract Value Environment*. We do this using *Replacements* as defined in Section 4.6. This is why the least upper bound operator for the *Abstract Collection Environments lubRep* does not only return the least upper bound of the *Abstract Collection Environments* but also a *Replacement* which then needs to be applied to the *Abstract Value Environment*.

$$\sqcup_{Must} : \left(Env^{(A)} \times Env^{(A)}\right) \to Env^{(A)}$$

$$\begin{aligned} &envL \sqcup_{Must} envR = \\ &(envL_C, envL_V) \sqcup_{Must} (envR_C, envR_V) = \quad (env_C, env_V) : \\ &\qquad\qquad (env_C, rep) = lubRep(envL_C, envR_C) \wedge \\ &\qquad\qquad env_V = replace_V(envL_V \sqcup_V envR_V, rep) \end{aligned}$$

The least upper bound operator for the *Abstract Value Environments* $\sqcup_V$ is provided by the value domain but the operator *lubRep* which creates the least upper bound for the *Abstract Collection Environment* as well as the necessary *Replacements* needs to be defined by our analysis.

$$lubRep : \left(Env_C^{(A)} \times Env_C^{(A)}\right) \to \left(Env_C^{(A)} \times Replacement\right)$$

$$lubRep(envL_C, envR_C) = \left(envL_C \ \sqcup_C^{Must} \ envR_C, getReplacement(envL_C, envR_C)\right)$$

Similarly as for the *May Analysis* we are using the standard least upper bound operator for functional domains to build the least upper bound for the two *Abstract Collection Environments*.

However, if a *Heap Identifier* occurs in both domains of the *Abstract Collection Environments* (which means that the abstract collection is present in both environments) we don't take the union of the *Element-Sets* but rather the intersection.

This is due to the fact that the least upper bound of two abstract collections in the *Must-Analysis* needs to represent only the elements that are contained in all the collections abstracted by both abstract collections that are joined.

$$\sqcup_C^{Must} : \left(Env_C^{(A)} \times Env_C^{(A)}\right) \to Env_C^{(A)}$$

$$envL_C \sqcup_C^{Must} envR_C = \left[\begin{array}{l} h \to c^{(A)} : h \in dom(envL_C) \cup dom(envR_C) \wedge \\ c^{(A)} = \begin{cases} envL_C(h) & \text{if } h \in dom(envL_C) - dom(envR_C) \\ envR_C(h) & \text{if } h \in dom(envR_C) - dom(envL_C) \\ envL_C(h) \cap envR_C(h) & \text{otherwise} \end{cases} \end{array} \right]$$

Since we take the intersection of two *Abstract Element Sets* all the *Tuple Identifiers* that are in only one of the two *Abstract Element Sets* are no longer present in the new *Abstract Collection Environment*. Hence we need to return a *Replacement* that tells the value domain to remove the key and value fields of those *Tuple Identifiers*.

A *Replacement* that represents the deletion of a set of identifiers simply has one entry where this set of identifiers points to an empty set. We call the set of identifiers that need to be removed S. Than the *Replacement* contains one entry where S points to the empty set.

$$getReplacements : \left(Env_C^{(A)} \times Env_C^{(A)} \right) \to Replacement$$
$$getReplacements(envL_C, envR_C) = [S \to \emptyset]$$

Now let's see how S is constructed. First notice that we are building one set of identifiers for the whole *Abstract Collection Environment*. *Tuple Identifiers* only vanish from the *Abstract Collection Environment* if an abstract collection is present in both *Abstract Collection Environments* (because in the least upper bound operator we only perform an intersection in this case). This means we only need to consider *Heap Identifiers* that are in the domain of both *Abstract Collection Environments* ($dom(envL_C) \cap dom(envR_C)$).

Each such *Heap Identifier* h points to an *Abstract Element Set* in the left and to an *Abstract Element Set* in the right *Abstract Collection Environment*. Since in the least upper bound we only keep *Tuple Identifiers* that are in the intersection of those two *Abstract Element Sets* the set S needs to contain all the *Tuple Identifiers* that are not in the intersection ($(envL_C(h) \cup envR_C(h)) - (envL_C(h) \cap envR_C(h))$).

Because the *Abstract Value Environment* tracks only the key and the value field and not the *Tuple Identifier* itself, S does not contain the *Tuple Identifier* itself but rather the key and value identifiers of the removed *Tuple Identifiers*.

$$S = \left\{ \begin{array}{l} (t, \text{"}key\text{"}), (t, \text{"}value\text{"}) : \\ t \in (envL_C(h) \cup envR_C(h)) - (envL_C(h) \cap envR_C(h)) \wedge \\ h \in dom(envL_C) \cap dom(envR_C) \end{array} \right\}$$

4.8.5 Extended least upper bound

With the described least upper bound operation we loose precision because we use the intersection of two *Abstract Element Sets*.

To illustrate this we show an example. Listing 4.2 shows a simple TouchDevelop script with a conditional. In both branches of the conditional an element is added to a *String Map*. As one can see, the key of the added element in both cases is *Zurich*.

Listing 4.2: Adding an element in all branches at the same key

```
1  action main() {
2          $universities := collections->create_string_map();
3
4          if (web->is_connected()) then{
5                  $universities->set_at("Zurich", "ETHZ");
6          } else {
7                  $universities->set_at("Zurich", "UZH");
8          }
9
10         $universities->at("Zurich");
11 }
```

We are interested in the collection access $\$universities \rightarrow at(\text{"}Zurich\text{"})$ at line 10. Obviously this collection access always returns a valid String. We can however only prove this, if we can track that the collection *must* contain an element at the key *Zurich*. The information whether an element is contained in the collection is represented by the *Abstract Element Set*.

At the end of the then branch the *Abstract Element Set* of the collection would contain one *Tuple Identifier* t_1 that represents the collection element added at line 5. The *Abstract Element Set* at the end of the else branch would contain one *Tuple Identifier* t_2 that represents the collection element added at line 7. However t_1 and t_2 are two different *Tuple Identifiers*. If the least upper bound operation simply takes the intersection of the two *Abstract Element Sets* we would end up with an empty *Abstract Element Set* and would therefore not be able to prove that there must be an element in the collection.

To get a more precise least upper bound, we need a more involved operation to join two *Abstract Element Sets*. Instead of taking the intersection we compare all *Tuple Identifiers* in the left *Abstract Element Set* with all *Tuple Identifiers* in the right *Abstract Element Set*. If we find two *Tuple Identifiers* t_i and t_j that have the same value for either the *key* or the *value* field we have to represent this in the new *Abstract Element Set*. We combine these two *Tuple Identifiers* to a new *Tuple Identifier* t. In the *Value Environment* we assign the least upper bound of the key fields of t_i and t_j to the key field of the new *Tuple Identifier* t. Analogously we do the same for the value field of t.

In the above example this operation would result in an *Abstract Element Set* with one *Tuple Identifier* whose key is *Zurich* and whose value is the least upper bound of *ETHZ* and *UZH*. Because of this, we are able to say that there must be an element with key *Zurich* in the abstract collection.

We are now going to formally define this extended least upper bound operator.

Because we need the *Abstract Value Environment* to join two *Abstract Element Sets*, we must pass it to the least upper bound operator of the *Abstract Collection Environment lubRep*. Similar to the standard least upper bound operator, the extended least upper bound operator of the *Abstract Collection Environments* replaces *Tuple Identifiers* with other *Tuple Identifiers* and therefore returns a *Replacement* which needs to be applied to the *Abstract Value Environment*.

$$\sqcup_{Must} : \left(Env^{(A)} \times Env^{(A)}\right) \to Env^{(A)}$$

$$\begin{aligned}
&(envL) \sqcup_{Must} (envR) = \\
&(envL_C, envL_V) \sqcup_{Must} (envR_C, envR_V) = \quad \left(env_C, env_V^{(1)}\right) : \\
&\qquad\qquad env_V = envL_V \sqcup_V envR_V \wedge \\
&\qquad\qquad (env_C, rep) = lubRep(envL_C, envR_C, env_V) \wedge \\
&\qquad\qquad env_V^{(1)} = replace_V(env_V, rep)
\end{aligned}$$

We define the least upper bound operator $\sqcup_C^{Must}$ and the creation of the *Replacement* separately.

$$lubRep : \left(Env_C^{(A)} \times Env_C^{(A)} \times Env_V^{(A)}\right) \to \left(Env_C^{(A)} \times Replacement\right)$$

$$lubRep(envL_C, envR_C, env_V) = \left(\sqcup_C^{Must}(envL_C, envR_C, env_V), getReplacement(envL_C, envR_C, env_V)\right)$$

To take the least upper bound of two *Abstract Collection Environments* we use the same function as for the standard least upper bound operator. However when an abstract collection is present in both *Abstract Collection Environments* we don't take the intersection of the two *Abstract Element Sets* but rather use the extended join operation $\sqcup_E^{Must}$.

$$\sqcup_C^{Must} : \left(Env_C^{(A)} \times Env_C^{(A)} \times Env_V\right) \to \left(Env_C^{(A)} \times Replacement\right)$$

$$\sqcup_C^{Must}(envL_C, envR_C, env_V) =$$

$$\left[\begin{array}{ll}
h \to c^{(A)} : & h \in dom(envL_C) \cup dom(envR_C) \wedge \\
& c^{(A)} = \begin{cases} envL_C(h) & \text{if } h \in dom(envL_C) - dom(envR_C) \\ envR_C(h) & \text{if } h \in dom(envR_C) - dom(envL_C) \\ \sqcup_E^{Must}(envL_C(h), envR_C(h), env_V) & \text{otherwise} \end{cases}
\end{array}\right]$$

The extended least upper bound operation $\sqcup_E^{Must}$ builds a new *Abstract Element Set* from the two provided *Abstract Element Sets* in the following way.

- If a *Tuple Identifier* is in both *Abstract Element Sets* this *Tuple Identifier* is added to the new *Abstract Element Set*.
- If a *Tuple Identifier* t_1 in one *Abstract Element Set* is the proper super set of a *Tuple Identifier* t_2 in the other *Abstract Element Set*, then t_1 is added to the new *Abstract Element Set*. Remember that *Tuple Identifiers* are sets of *Heap Identifiers* which allows us to use the super set relation here.
- For all other *Tuple Identifiers* we are checking whether there exist two *Tuple Identifiers* t_1 and t_2 such that t_1 is in the left *Abstract Element Set*, t_2 is in the right *Abstract Element Set* and either the key or the value field of the two *Tuple Identifiers* must be equal in the *Abstract Value Environment*. (This is formally described by the *considerT* function). For all pairs of *Tuple Identifiers* that match this criteria, we create a new *Tuple Identifier* that is the union of t_1 and t_2 (Remember that *Tuple Identifiers* are sets of *Heap Identifiers*).
- All other *Tuple Identifiers* are not part of the new *Abstract Element Set*.

$$\sqcup_E^{Must} : \left(\mathcal{P}(TId) \times \mathcal{P}(TId) \times Env_V^{(A)}\right) \to \mathcal{P}(TId)$$

$$\sqcup_E^{Must}\left(cL^{(A)}, cR^{(A)}, env_V\right) = \left\{ \begin{array}{ll} t: & t_L \in cL^{(A)} \wedge t_R \in cR^{(A)} \wedge \\ & \left(\begin{array}{l} (t_L = t_R \wedge t = t_L) \vee \\ (t_L \subset t_R \wedge t = t_R) \vee \\ (t_R \subset t_L \wedge t = t_L) \vee \\ \left(\begin{array}{l} considerT(t_R, t_L, env_V) \wedge \\ t = t_L \cup t_R \end{array} \right) \end{array} \right) \end{array} \right\}$$

$$considerT : \left(TId \times TId \times Env_V^{(A)}\right) \to \{true, false\}$$

$$\begin{array}{rl} considerT(t_1, t_2, env_V) = & equals_V^{(A)}((t_1, \text{"}key\text{"}), (t_2, \text{"}key\text{"}), env_V) = \{true\} \vee \\ & equals_V^{(A)}((t_1, \text{"}value\text{"}), (t_2, \text{"}value\text{"}), env_V) = \{true\} \end{array}$$

Beside the least upper bound we also have to create a *Replacement* that tells the *Abstract Value Environment* which *Tuple Identifiers* are no longer present in the new *Abstract Element Set*. We build one *Replacement* for the complete *Abstract Collection Environment*.

In the *Abstract Value Environment* we track the key and the value fields of a *Tuple Identifier*. Therefore a *Replacement* for the keys and a *Replacement* for the value fields are created and then joined ($\oplus$ operator). Notice that we use the $getReplacement'$ function to create the *Replacement* for the key fields as well as for the value fields. We pass the field name (either key or value) as a parameter to the $getReplacement'$ function.

$$getReplacement : \left(Env_C^{(A)} \times Env_C^{(A)} \times Env_V^{(A)}\right) \to Replacement$$

$$\begin{array}{rl} getReplacement(envL_C, envR_C, env_V) = & getReplacement'(envL_C, envR_C, env_V, \text{"}key\text{"}) \\ & \oplus getReplacement'(envL_C, envR_C, env_V, \text{"}value\text{"}) \end{array}$$

We have to consider all the *Abstract Element Sets* of the collections that exist in both *Abstract Collection Environments*, since only for those the least upper bound operation replaces *Tuple Identifiers*.

The rules to build the *Replacement* based on two *Abstract Element Sets* are similar to the ones used to build the least upper bound:

- If a *Tuple Identifier* is in both *Abstract Element Sets*, no *Replacement* is needed.
- If one *Tuple Identifier* t_1 is a proper subset of another *Tuple Identifier* t_2, an entry is added to the *Replacement* that replaces t_1 with t_2.
- If a new *Tuple Identifier* is created out of two *Tuple Identifiers* in the least upper bound operation the old *Tuple Identifiers* must be replaced by the new one.

$$getReplacement' : \left(Env_C^{(A)} \times Env_C^{(A)} \times Env_V^{(A)} \times \{\text{"}key\text{"}, \text{"}value\text{"}\}\right) \to Replacement$$

$$getReplacement'(envL_C, envR_C, env_V, n) = \left[\begin{array}{ll} F \to T: & t_L \in c_L \wedge t_R \in c_R \wedge \\ & F = \{(t_L, n), (t_R, n)\} \wedge \\ & \left(\begin{array}{l} t_L \subset t_R \wedge T = \{(t_R, n)\}) \vee \\ (t_R \subset t_L \wedge T = \{(t_L, n)\}) \vee \\ \left(\begin{array}{l} consider(t_R, t_L, env_V) \wedge \\ T = \{(t_L \cup t_R, n)\} \end{array} \right) \end{array} \right) \end{array} \right]$$

Chapter 5

Semantics

In this Chapter we formally define the concrete and the abstract semantics of the most important collection operations in the TouchDevelop standard library. We also argue that the abstract semantics are sound by showing that they over-approximate the concrete semantics.

5.1 Concrete and Abstract Semantics

The concrete semantics describe how an operation in TouchDevelop changes the state of the concrete domain.

Since in static program analysis we consider all possible program executions, the concrete semantics operate on a set of states. In our analysis the state is an *Environment* as defined in Chapters 4 and 3. Formally, a concrete semantic operation $op^{(S)}$ takes a set of *Environments* as an input and transfers them into a new set of *Environments*.

$$op^{(S)} : \mathcal{P}(Env) \to \mathcal{P}(Env)$$

We will define each concrete semantic operation by describing how it changes a single *Environment*. The concrete semantic over a set of *Environments* is then the pointwise application of the operation on that set. Formally, if we define a semantic operation op, then the concrete semantic operation over a set of *Environments* is

$$op^{(S)}(envs) = \{op(env) : env \in envs\}$$

The abstract semantics is the counterpart of the concrete semantics in the abstract domain. The state in the abstract domain is defined by the *Abstract Environment*. An abstract semantic operation $op^{(A)}$ transfers an *Abstract Environment* into another *Abstract Environment*.

$$op^{(A)} : (Env^{(A)}) \to (Env^{(A)})$$

5.1.1 Soundness Proofs for Semantics

To prove that an abstract semantic operation is sound, we need to show that it over-approximates the corresponding concrete semantic operation.

The general idea of the soundness proofs is as follows: Each abstract state abstracts multiple concrete states. If we now apply the abstract semantics to the abstract state and the concrete semantics to each of the concrete states, the new abstract state must abstract at least all the new concrete states.

Formally, the concretization of an abstract operation $op^{(A)}$ on an *Abstract Environment* $env^{(A)}$ must be a superset of the concrete operation *op* applied to all concrete *Environments* abstracted by $env^{(A)}$.

$$\gamma\left(op^{(A)}\left(env^{(A)}\right)\right) \supseteq op^{(S)}\left(\gamma\left(env^{(A)}\right)\right)$$

Or, if we apply the definition of $op^{(S)}$:

$$\gamma\left(op^{(A)}\left(env^{(A)}\right)\right) \supseteq \left\{op(env) : env \in \gamma\left(env^{(A)}\right)\right\}$$

This form will be used for most soundness proofs.

5.2 Basic Operations

To keep the formalization modular and easier to read we will first describe a few basic operations that are then used to define the semantics for TouchDevelop collection operations.

For each basic operation we will define the concrete and the abstract semantics and prove its soundness, by showing that the abstract operation over-approximates the concrete one. The soundness has to be proven for the *May* and the *Must Analysis* separately since those are two different abstractions.

We can later use these results to prove the soundness of the abstract semantics.

5.2.1 Isolation Lemma

First we will define a lemma that will be used in a lot of proofs to isolate the changes, applied to a single concrete collection, from the rest of a collection environment.

When we prove that the abstract semantics is sound, the concrete semantics is applied to every concrete *Collection Environment* that is abstracted by the same *Abstract Collection Environment*. To retrieve all the concrete collection environments that are abstracted by an *Abstract Collection Environment* $env_C^{(A)}$ we use the concretization function $\gamma_C\left(env_C^{(A)}\right)$. On each concrete collection environment that we retrieve from $\gamma_C\left(env_C^{(A)}\right)$ we apply the concrete semantics. Usually the concrete semantics only change a single collection c in the *Collection Environment*. With this lemma we isolate the parts of $\gamma_C\left(env_C^{(A)}\right)$ that create the entries for the collection c in the concrete collection environments. We can then use this to represent the concrete semantics in terms of the *Abstract Collection Environment*.

Lemma 1. *For a any* Abstract Environment $\left(env_C^{(A)}, env_V^{(A)}\right)$ *and for any abstract collection* $c^{(A)} \in dom\left(env_C^{(A)}\right)$*, and for any concrete collection* $c \in \gamma_{HId}\left(c^{(A)}\right)$*, the following equation holds for both* γ_C^{Must} *and* γ_C^{May} $\left(\gamma_C \in \left\{\gamma_C^{May}, \gamma_C^{(Must)}\right\}\right)$*:*

$$\gamma_C\left(env_C^{(A)}, env_V^{(A)}\right) =$$

$$\left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E(E) \wedge \\ & E = \left\{ \begin{array}{l} t' : \\ t' \in env_C^{(A)}(i) \wedge \\ notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) - \{c\} \wedge \\ & c' \in \gamma_E(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \\ \cup \left\{ \begin{array}{ll} (c, c') : & c' \in \gamma_E(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

Proof. The definitions for γ_C^{May} and γ_C^{Must} are identical except that one uses the function γ_E^{May} and the other uses the function γ_E^{Must} to concretize an *Abstract Element Set*. We will therefore show the lemma for all $\gamma_E \in \left\{\gamma_E^{May}, \gamma_E^{Must}\right\}$ and hence prove, that the lemma holds for γ_C^{May} and for γ_C^{Must}.

We can directly prove this by using the definition of γ_C and the restrictions defined in the lemma that $c \in \gamma_{HId}\left(c^{(A)}\right)$ and $c^{(A)} \in dom\left(env_C^{(A)}\right)$.

$$\begin{array}{l} \gamma_C\left(env_C^{(A)}, env_V^{(A)}\right) \\ = \left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ \\ x \subseteq \bigcup_{i \in dom\left(env_C^{(A)}\right)} \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}(i) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right\} \end{array}$$

$$= \left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E(E) \wedge \\ & E = \left\{ \begin{array}{l} t' : \\ t' \in env_C^{(A)}(i) \wedge \\ notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) \wedge \\ & c' \in \gamma_E(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

$$= \left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E(E) \wedge \\ & E = \left\{ \begin{array}{l} t' : \\ t' \in env_C^{(A)}(i) \wedge \\ notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) - \{c\} \wedge \\ & c' \in \gamma_E(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \\ \cup \left\{ \begin{array}{ll} (c,c') : & c' \in \gamma_E(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

□

5.2.2 Adding an Element to a Collection

This operation defines how an element is added to a collection in a collection environment.

This means that in the concrete domain we are adding an *Element* e to the *Element Set* of a collection c in a *Collection Environment* env_C.

$\mathbf{add_C}: \quad (\mathbf{Ref} \times \mathbf{Elem}) \to (\mathbf{Env_C} \to \mathbf{Env_C})$
$add_C \quad (c,e)(env_C) = env_C[c \to env_C(c) \cup \{e\}]$

In the abstract semantics a *Tuple Identifier* t is added to the *Abstract Element Set* of an abstract collection $c^{(A)}$ in an *Abstract Collection Environment* $env_C^{(A)}$.

$\mathbf{add_C^{(A)}}: \quad (\mathbf{HId} \times \mathbf{TId}) \to \left(\mathbf{Env_C^{(A)}} \to \mathbf{Env_C^{(A)}}\right)$
$add_C^{(A)} \quad (c,t)(env_C) = env_C[c \to env_C(c) \cup \{t\}]$

For the *May Analysis* we can directly use $add_C^{(A)}$:

$\mathbf{add_C^{(May)}}: \quad (\mathbf{HId} \times \mathbf{TId}) \to \left(\mathbf{Env_C^{(A)}} \to \mathbf{Env_C^{(A)}}\right)$
$add_C^{(May)} \quad (c,t)(env_C) = add_C^{(A)}(c,t,env_C)$

For the *Must Analysis* this would not be sound if the affected abstract collection is a summary collection. To illustrate this, consider an abstract summary collection $c^{(A)}$ that abstracts two

distinct concrete collections c_1 and c_2. Now we add a new element e to the second concrete collection c_2. This operation is abstracted by the abstract add operation. Remember that in the *Must Analysis* an abstract collection abstracts all the concrete collections that contain at least the elements that are in the abstract collection. It would not be sound to add e to the abstract collection $c^{(A)}$, since c_1 would not contain e after the add operation. Similarly it would be unsound to add a *Tuple Identifier* that abstracts multiple *Elements* to an abstract collection in the *Must Analysis*.

For this reason we have to refine the abstract semantics of the add operation for the *Must Analysis*. If the provided abstract collection c is not a summary collection ($|\gamma_{HId}(c) = 1|$) and the *Tuple Identifier* t that shall be added is not a summary identifier ($|\gamma_{TId}(c) = 1|$), we can safely add the element to the collection. But otherwise we can not add the element to the collection and therefore do not change the *Abstract Collection Environment*. Notice that we use the function $isSummary$ that we defined in Section 4.5 to compute if a given identifier is a summary node.

$$\mathbf{add}_{\mathbf{C}}^{(\mathbf{Must})} : \quad (\mathbf{HId} \times \mathbf{TId}) \to \left(\mathbf{Env}_{\mathbf{C}}^{(\mathbf{A})} \to \mathbf{Env}_{\mathbf{C}}^{(\mathbf{A})}\right)$$

$$add_C^{(Must)} \qquad (c,t)(env_C) = \begin{cases} add_C^{(A)}(c,t,env_C) & \text{if } \neg isSummary(c) \wedge \neg isSummary(t) \\ env_C & \text{otherwise} \end{cases}$$

Soundness - May Analysis

We first prove the soundness of $add_C^{(May)}$. To do this, we need to show that adding an element to an abstract collection over-approximates adding an element to a concrete collection.

However this is not enough to prove the soundness of $add_C^{(May)}$. Imagine that we have two concrete collections c_1 and c_2 that are abstracted by an abstract collection $c^{(A)}$ (which therefore is a summary collection). Assume now, that an element is added to c_1. This operation is abstracted by adding an element to the abstract collection $c^{(A)}$. After the add operation collection c_2 remains unchanged but is still abstracted by $c^{(A)}$. We therefore have to show that adding an element to the abstract collection also over-approximates leaving a concrete collection unchanged.

We will show these two cases with the Lemmas 2 and 3. We can then use these Lemmas to show that $add_C^{(May)}$ is sound for complete collection environments.

Lemma 2. *Adding a* Tuple Identifier t *to the* Abstract Element Set $E^{(A)}$ *of an abstract collection over-approximates adding an* Element $e \in \gamma_{TId}(t)$ *to the* Element Set E *of a concrete collection. This holds for all* Abstract Element Sets $E^{(A)} \in \mathcal{P}(TId)$ *and for all* Tuple Identifiers $t \in TId$.

$$\left\{E \cup \{e\} : E \in \gamma_E^{May}\left(E^{(A)}\right)\right\} \subseteq \gamma_E^{May}\left(E^{(A)} \cup \{t\}\right)$$

Proof. We show this directly and start with $\left\{E \cup \{e\} : E \in \gamma_E^{May}(E^{(A)})\right\}$.

We first apply the definition of γ_E^{May} (1). Then we can use that $\{X \cup R : X \subseteq T\} \subseteq \{X : X \subseteq T \cup R\}$ holds for all sets T and R, which we have proven in Appendix B (3). For transformation (4) we use that $e \in \gamma_{TId}(t)$ and therefore $\{e\} \subseteq \gamma_{TId}(t)$.

$$\begin{aligned}
\left\{E \cup \{e\} : E \in \gamma_E^{May}\left(E^{(A)}\right)\right\} &\overset{(1)}{=} \left\{E \cup \{e\} : E \in \left\{x : x \subseteq \bigcup_{i \in E^{(A)}} \gamma_{TId}(i)\right\}\right\} \\
&\overset{(2)}{=} \left\{E \cup \{e\} : E \subseteq \bigcup_{i \in E^{(A)}} \gamma_{TId}(i)\right\} \\
&\overset{(3)}{\subseteq} \left\{E : E \subseteq \left(\left(\bigcup_{i \in E^{(A)}} \gamma_{TId}(i)\right) \cup \{e\}\right)\right\} \\
&\overset{(4)}{\subseteq} \left\{E : E \subseteq \left(\left(\bigcup_{i \in E^{(A)}} \gamma_{TId}(i)\right) \cup \gamma_{TId}(t)\right)\right\} \\
&\overset{(5)}{=} \left\{E : E \subseteq \bigcup_{i \in E^{(A)} \cup \{t\}} \gamma_{TId}(i)\right\} \\
&\overset{(6)}{=} \gamma_E^{May}\left(E^{(A)} \cup \{t\}\right)
\end{aligned}$$

□

Lemma 3. *Adding a* Tuple Identifier t *to the* Abstract Element Set $E^{(A)}$ *of an abstract collection over-approximates not changing the* Element Set *of a concrete collection. This holds for all* Abstract Element Sets $E^{(A)} \in \mathcal{P}(TId)$ *and for all* Tuple Identifiers $t \in TId$.

$$\left\{E : E \in \gamma_E^{May}\left(E^{(A)}\right)\right\} \subseteq \gamma_E^{May}\left(E^{(A)} \cup \{t\}\right)$$

Proof. This proof is straight-forward and similar to the proof of Lemma 2.

$$\begin{aligned}
\left\{E : E \in \gamma_E^{May}\left(E^{(A)}\right)\right\} &= \left\{E : E \subseteq \bigcup_{i \in E^{(A)}} \gamma_{TId}(i)\right\} \\
&\subseteq \left\{E : E \subseteq \left(\left(\bigcup_{i \in E^{(A)}} \gamma_{TId}(i)\right) \cup \gamma_{TId}(t)\right)\right\} \\
&= \left\{E : E \subseteq \bigcup_{i \in E^{(A)} \cup \{t\}} \gamma_{TId}(i)\right\} \\
&= \gamma_E^{May}\left(E^{(A)} \cup \{t\}\right)
\end{aligned}$$

□

With these two Lemmas we are now able to prove that $add_C^{(May)}$ is sound.

Theorem 1. *Adding a* Tuple Identifier t *to an abstract collection* $c^{(A)}$ *in an* Abstract Environment $\left(env_C^{(A)}, env_V^{(A)}\right)$ *over-approximates adding any* Element $e \in \gamma_{TId}(t)$ *to any collection in* $c \in \gamma_{HId}\left(c^{(A)}\right)$ *in all* Collection Environments $(env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)$. *This holds for all* Tuple Identifiers $t \in TId$, *for all abstract collections* $c^{(A)} \in dom\left(env_C^{(A)}\right)$ *and for all abstract environments* $(env_C^{(A)}, env_V^{(A)}) \in Env^{(A)}$.

$$\begin{gathered}
\left\{add_C(c, e, env_C) : env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)\right\} \\
\subseteq \\
\gamma_C^{May}\left(add_C^{(A)}\left(c^{(A)}, t, env_C^{(A)}\right), env_V^{(A)}\right)
\end{gathered}$$

Proof. We are going to show that if $z \in \left\{add_C(c, e, env_C) : env_c \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$ it follows that $z \in \gamma_C^{May}\left(add_C^{(May)}\left(c^{(A)}, t, env_C^{(A)}\right), env_V^{(A)}\right)$ which, by the definition of $\subseteq$, proves the Theorem.

$$z \in \left\{add_C(c, e, env_C) : env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$$

First we apply the definition of add_C.

$$z \in \left\{ env_C[c \to env_C(c) \cup \{e\}] : env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right) \right\}$$

We can see that in each *Collection Environment* in $\gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)$ only the entry for the concrete collection c is changed. Therefore we use Lemma 1 to isolate the part of $\gamma_C^{(May)}$ that generates the entries for the concrete collection c in the *Collection Environment*.

$$\gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right) =$$

$$\left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}(i) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) - \{c\} \wedge \\ & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \\ \cup \left\{ \begin{array}{ll} (c, c') : & c' \in \gamma_E^{May}(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

The part that generates the entries for the concrete collection c in the collection environments is:

$$\left\{ \begin{array}{ll} (c, c') : & c' \in \gamma_E^{May}(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\}$$

Since the changes of the add_C operation only affect the entries of collection c in the *Collection Environments* and because all entries of collection c in the *Collection Environments* are generated from the above set, we can directly apply this changes to (c, c'). From this follows:

$$z \in \left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}(i) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) - \{c\} \wedge \\ & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \\ \cup \left\{ \begin{array}{ll} (c, c' \cup \{e\}) : & c' \in \gamma_E^{May}(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

From Lemmas 2 and 3 we know that $\gamma_E^{May}\left(env_C^{(A)}\left(c^{(A)}\right) \cup \{t\}\right)$ is a superset of $\left\{c' : c' \in \gamma_E^{May}\left(env_C^{(A)}\left(c^{(A)}\right)\right)\right\}$ and of $\left\{c' \cup \{e\} : c' \in \gamma_E^{May}\left(env_C^{(A)}\left(c^{(A)}\right)\right)\right\}$.
Using these two results we can conclude:

$$z \in \left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}(i) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) - \{c\} \wedge \\ & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \cup \{t\} \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \\ \cup \left\{ \begin{array}{ll} (c,c') : & c' \in \gamma_E^{May}(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \cup \{t\} \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

Finally, we can simplify this expression.

$$z \in \left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ \\ x \subseteq \bigcup_{i \in dom\left(env_C^{(A)}\right)} \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{l} t' : \\ t' \in env_C^{(A)}\left[c^{(A)} \to env_C^{(A)}\left(c^{(A)}\right) \cup \{t\}\right](i) \wedge \\ notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right\}$$

By applying the definition of γ_C^{May} we get

$$z \in \gamma_C^{May}\left(env_C^{(A)}\left[c^{(A)} \to env_C^{(A)}\left(c^{(A)}\right) \cup \{t\}\right], env_V^{(A)}\right)$$

But this is the same as $z \in \gamma_C^{May}\left(add_C^{(May)}\left(c^{(A)}, t, env_C^{(A)}\right), env_V^{(A)}\right)$, which is what we wanted to prove. □

Soundness - Must Analysis

The proof for the *Must Analysis* is very similar to the proof for the *May Analysis*.

Since the abstract $add_C^{(Must)}$ operation is defined differently depending on whether the provided collection and *Tuple Identifier* are summary identifiers or not, we will distinguish these two cases in the proof.

For both cases we will show that $add_C^{(Must)}$ is sound. This will then allow us to show that $add_C^{(Must)}$ is sound for all *Tuple Identifiers* and for all abstract collections.

Case 1: Non Summary collections
In this case we assume that the provided abstract collection $c^{(A)}$ is not a summary collection ($|\gamma_{HId}\left(c^{(A)}\right)| = 1$) and that the provided *Tuple Identifier* t is not a summary identifier ($|\gamma_{TId}(t)| = 1$).

Similar to the proof of the *May Analysis* we will first show the soundness for a single collection and then use this to show that the operation is sound for complete collection environments.

Lemma 4. *Adding a* Tuple Identifier t *to the* Abstract Element Set $E^{(A)}$ *of an abstract collection over-approximates adding an* Element $e \in \gamma_{TId}(t)$ *to the* Element Set *of a concrete collection. This holds for all* Abstract Element Sets $E^{(A)} \in \mathcal{P}(TId)$ *and for all* ***non-summary*** Tuple Identifiers $t \in TId$.

$$\left\{E \cup \{e\} : E \in \gamma_E^{Must}\left(E^{(A)}\right)\right\} \subseteq \gamma_E^{Must}\left(E^{(A)} \cup \{t\}\right)$$

Proof. We show this directly and we start with $\left\{E \cup \{e\} : E \in \gamma_E^{Must}(E^{(A)})\right\}$.

First we apply the definition of γ_E^{Must} and then reformulate the resulting expression.

We first apply the definition of γ_E^{Must} (1). Then we can use the fact that $\{X \cup R : X \supseteq T\} \subseteq \{X : X \supseteq T \cup R\}$ holds for all sets T and R, which we have proven in Appendix B (3). For transformation (4) we use that in this case $\{e\} = \gamma_{TId}(t)$, since $e \in \gamma_{TId}(t)$ and $|\gamma_{TId}(t)| = 1$.

$$\begin{aligned}
\left\{E \cup \{e\} : E \in \gamma_E^{Must}\left(E^{(A)}\right)\right\} &\overset{(1)}{=} \left\{E \cup \{e\} : E \in \left\{x : x \supseteq \textstyle\bigcup_{i \in E^{(A)}} \gamma_{TId}(i)\right\}\right\} \\
&\overset{(2)}{=} \left\{E \cup \{e\} : E \supseteq \textstyle\bigcup_{i \in E^{(A)}} \gamma_{TId}(i)\right\} \\
&\overset{(3)}{\subseteq} \left\{E : E \supseteq \left(\left(\textstyle\bigcup_{i \in E^{(A)}} \gamma_{TId}(i)\right) \cup \{e\}\right)\right\} \\
&\overset{(4)}{=} \left\{E : E \supseteq \left(\left(\textstyle\bigcup_{i \in E^{(A)}} \gamma_{TId}(i)\right) \cup \gamma_{TId}(t)\right)\right\} \\
&\overset{(5)}{=} \left\{E : E \supseteq \textstyle\bigcup_{i \in E^{(A)} \cup \{t\}} \gamma_{TId}(i)\right\} \\
&\overset{(6)}{=} \gamma_E^{Must}\left(E^{(A)} \cup \{t\}\right)
\end{aligned}$$

□

With this Lemma we are now able to show that $add_C^{(Must)}$ is sound, if a non-summary *Tuple Identifier* is added to a non-summary collection.

Lemma 5. *Adding a* ***non-summary*** Tuple Identifier t *to an abstract* ***non-summary*** *collection* $c^{(A)}$ *in an* Abstract Environment $\left(env_C^{(A)}, env_V^{(A)}\right)$ *over-approximates adding any* Element $e \in \gamma_{TId}(t)$ *to any collection* $c \in \gamma_{HId}\left(c^{(A)}\right)$ *in all* Collection Environments $env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)$. *This holds for all* ***non-summary*** Tuple Identifiers $t \in TId$, *for all abstract* ***non-summary*** *collections* $c^{(A)} \in dom\left(env_C^{(A)}\right)$ *and for all abstract environments* $\left(env_C^{(A)}, env_V^{(A)}\right) \in Env^{(A)}$.

$$\begin{gathered}
\left\{add_C(c, e, env_C) : env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)\right\} \\
\subseteq \\
\gamma_C^{Must}\left(add_C^{(Must)}\left(c^{(A)}, t, env_C^{(A)}\right), env_V^{(A)}\right)
\end{gathered}$$

Proof. We are going to show that if $z \in \left\{add_C(c,e,env_C) : env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$ it follows that $z \in \gamma_C^{Must}\left(add_C^{(Must)}\left(c^{(A)}, t, env_C^{(A)}\right), env_V^{(A)}\right)$ which, by the definition of $\subseteq$, proves the Lemma.

$$z \in \left\{add_C(c,e,env_C) : env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$$

First we apply the definition of add_C.

$$z \in \left\{env_C[c \to env_C(c) \cup \{e\}] : env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$$

We again use Lemma 1 to isolate the part of $\gamma_C^{(Must)}$ that generates the entries for the concrete collection c in the *Collection Environments*. In this case we furthermore know that $c^{(A)}$ is not a summary collection and therefore the only element in $\gamma_{HId}\left(c^{(A)}\right)$ is the concrete collection c. From this follows that $\gamma_{HId}\left(c^{(A)}\right) - \{c\} = \emptyset$ which allows us to further adapt $\gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)$.

$$\gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right) =$$

$$\left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{Must}(E) \wedge \\ & E = \left\{ \begin{array}{l} t' : \\ t' \in env_C^{(A)}(i) \wedge \\ notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) - \{c\} \wedge \\ & c' \in \gamma_E^{Must}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \\ \cup \left\{ \begin{array}{ll} (c,c') : & c' \in \gamma_E^{Must}(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\} = $$

$$\left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{Must}(E) \wedge \\ & E = \left\{ \begin{array}{l} t' : \\ t' \in env_C^{(A)}(i) \wedge \\ notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (c,c') : & c' \in \gamma_E^{Must}(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

We can now replace $\gamma_C\left(env_C^{(A)}, env_V^{(A)}\right)$ with this expression and again apply the change to the *Element Set* of c directly to (c,c').

$$z \in \left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{Must}(E) \wedge \\ & E = \left\{ \begin{array}{l} t' : \\ t' \in env_C^{(A)}(i) \wedge \\ notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (c, c' \cup \{e\}) : & c' \in \gamma_E^{Must}(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

From Lemma 4 we know that $\left\{c' \cup \{e\} : c' \in \gamma_E^{Must}\left(env_C^{(A)}\left(c^{(A)}\right)\right)\right\} \subseteq \gamma_E^{Must}\left(env_c^{(A)}\left(c^{(A)}\right) \cup \{t\}\right)$. Therefore we can conclude

$$z \in \left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{Must}(E) \wedge \\ & E = \left\{ \begin{array}{l} t' : \\ t' \in env_C^{(A)}(i) \wedge \\ notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (c, c') : & c' \in \gamma_E^{Must}(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \cup \{t\} \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

Similar to the proof of Theorem 1 we can simplify this to

$$z \in \gamma_C^{Must}\left(env_C^{(A)}\left[c^{(A)} \to env_C^{(A)}\left(c^{(A)}\right) \cup \{t\}\right], env_V^{(A)}\right)$$

But since in this case we assumed that $c^{(A)}$ and t are not summary identifiers, this is equal to $z \in \gamma_C^{Must}\left(add_C^{(Must)}\left(c^{(A)}, t, env_C^{(A)}\right), env_V^{(A)}\right)$ which is exactly what we wanted to prove. □

Case 2: Summary collections
In the second case we assume that the provided abstract collection $c^{(A)}$ is a summary collection ($|\gamma_{HId}\left(c^{(A)}\right)| > 1$) and / or that the provided *Tuple Identifier* t is a summary identifier ($|\gamma_{TId}(t)| > 1$).

In this case the abstract semantic of the $add_C^{(Must)}$ operation returns the unchanged *Collection Environment*. Hence, we need to show that this over-approximates the concrete semantics of add_C which adds an *Element* to the collection.

We are going to show first in Lemma 6, that this is sound for a single collection and then in Lemma 7, that it is sound for a complete collection environment.

Lemma 6. *Not changing the* Abstract Element Set $E^{(A)}$ *of an abstract collection over-approximates adding an* Element e *to the* Element Set *of a concrete collection. This holds for all* Elements $e \in Elem$ *and for all* Abstract Element Sets $E^{(A)} \in \mathcal{P}(TId)$.

$$\left\{E \cup \{e\} : E \in \gamma_E^{Must}\left(E^{(A)}\right)\right\} \subseteq \gamma_E^{Must}\left(E^{(A)}\right)$$

Proof. We show this directly and we start with $\left\{E \cup \{e\} : E \in \gamma_E^{Must}\left(E^{(A)}\right)\right\}$.

We first apply the definition of γ_E^{Must} (1). For transformation (3) we use that $\{X \cup R : X \supseteq T\} \subseteq \{X : X \supseteq T \cup R\}$ holds for all sets T and R, which we have proven in Appendix B.

$$\begin{aligned}
\left\{E \cup \{e\} : E \in \gamma_E^{Must}\left(E^{(A)}\right)\right\} &\overset{(1)}{=} \left\{E \cup \{e\} : E \in \left\{x : x \supseteq \textstyle\bigcup_{i \in E^{(A)}} \gamma_{TId}(i)\right\}\right\} \\
&\overset{(2)}{=} \left\{E \cup \{e\} : E \supseteq \textstyle\bigcup_{i \in E^{(A)}} \gamma_{TId}(i)\right\} \\
&\overset{(3)}{\subseteq} \left\{E : E \supseteq \left(\left(\textstyle\bigcup_{i \in E^{(A)}} \gamma_{TId}(i)\right) \cup \{e\}\right)\right\} \\
&\overset{(4)}{\subseteq} \left\{E : E \supseteq \textstyle\bigcup_{i \in E^{(A)}} \gamma_{TId}(i)\right\} \\
&\overset{(5)}{=} \gamma_E^{Must}\left(E^{(A)}\right)
\end{aligned}$$

□

With this Lemma we can now prove that $add_C^{(Must)}$ is sound for collection environments, if the added *Tuple Identifier* and/or the abstract collection is a summary identifier.

Lemma 7. *Applying* $add_C^{(Must)}$ *to an* Abstract Environment $\left(env_C^{(A)}, env_V^{(A)}\right)$ *with an abstract collection* $c^{(A)}$ *and a* Tuple Identifier t *over-approximates adding an* Element $e \in \gamma_{TId}(t)$ *to any concrete collection* $c \in \gamma_{HId}\left(c^{(A)}\right)$ *in all* Collection Environments $env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)$ *if* $c^{(A)}$ *and/or* t *is a summary identifier. This holds for all* Tuple Identifiers $t \in TId$, *for all abstract collections* $c^{(A)} \in dom\left(env_C^{(A)}\right)$ *and for all* Abstract Environments $\left(env_C^{(A)}, env_V^{(A)}\right) \in Env^{(A)}$.

$$\begin{gathered}
\left\{add_C(c, e, env_C) : env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)\right\} \\
\subseteq \\
\gamma_C^{Must}\left(add_C^{(Must)}\left(c^{(A)}, t, env_C^{(A)}\right), env_V^{(A)}\right)
\end{gathered}$$

Proof. We are going to show that if $z \in \left\{add_C(c, e, env_C) : env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$ it follows that $z \in \gamma_C^{Must}\left(add_C^{(Must)}\left(c^{(A)}, t, env_C^{(A)}\right), env_V^{(A)}\right)$ which, by the definition of $\subseteq$, proves the Lemma.

$$z \in \left\{add_C(c, e, env_C) : env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$$

We first apply the definition of add_C

$$z \in \left\{env_C[c \to env_C(c) \cup \{e\}] : env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$$

We use Lemma 1 to isolate the part of γ_C^{Must} that generates the entries for the concrete collection c in the *Collection Environments*.

$$\gamma_C\left(env_C^{(A)}, env_V^{(A)}\right) =$$

$$\left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ \quad x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E(E) \wedge \\ & E = \left\{ \begin{array}{l} t' : \\ t' \in env_C^{(A)}(i) \wedge \\ notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) - \{c\} \wedge \\ & c' \in \gamma_E(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \\ \cup \left\{ \begin{array}{ll} (c,c') : & c' \in \gamma_E(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

We can replace $\gamma_C\left(env_C^{(A)}, env_V^{(A)}\right)$ with this expression and apply the change to the *Element Set* of c directly to (c, c').

$$z \in \left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ \quad x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E(E) \wedge \\ & E = \left\{ \begin{array}{l} t' : \\ t' \in env_C^{(A)}(i) \wedge \\ notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) - \{c\} \wedge \\ & c' \in \gamma_E(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \\ \cup \left\{ \begin{array}{ll} (c,c' \cup \{e\}) : & c' \in \gamma_E(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

From Lemma 6 we know $\left\{c' \cup \{e\} : c' \in \gamma_E^{Must}\left(env_C^{(A)}\left(c^{(A)}\right)\right)\right\} \subseteq \gamma_E^{Must}\left(env_C^{(A)}\left(c^{(A)}\right)\right)$. And we can therefore deduce

$$z \in \left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E(E) \wedge \\ & E = \left\{ \begin{array}{l} t' : \\ t' \in env_C^{(A)}(i) \wedge \\ notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) - \{c\} \wedge \\ & c' \in \gamma_E(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \\ \cup \left\{ \begin{array}{ll} (c,c') : & c' \in \gamma_E(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

Similar to the proof of Theorem 1 we can simplify this to

$$z \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)$$

But since in this case we assumed that $c^{(A)}$ and / or t is a summary identifier, this is equal to $z \in \gamma_C^{Must}\left(add_C^{(Must)}\left(c^{(A)}, t, env_C^{(A)}\right), env_V^{(A)}\right)$, which is exactly what we wanted to prove. □

Since we have proven for both summary and for non-summary identifiers that $add_C^{(Must)}$ is sound, we can now prove that $add_C^{(Must)}$ is sound for all collections and for all *Tuple Identifiers*.

Theorem 2. *Applying* $add_C^{(Must)}$ *on an* Abstract Environment $\left(env_C^{(A)}, env_V^{(A)}\right)$ *with an abstract collection* $c^{(A)}$ *and a* Tuple Identifier t *over-approximates applying* add_C *on any concrete collection* $c \in \gamma_{HId}\left(c^{(A)}\right)$ *and any* Element $e \in \gamma_{HId}(t)$ *for all* Collection Environments $env_C \in \gamma_C^{Must}\left(c^{(A)}, env_V^{(A)}\right)$. *This holds for all abstract collections* $c^{(A)} \in dom\left(env_C^{(A)}\right)$, *for all* Tuple Identifiers $t \in TId$ *and for all* Abstract Environments $\left(env_C^{(A)}, env_V^{(A)}\right) \in Env_C^{(A)}$.

$$\left\{ add_C(c,e,env_C) : env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right) \right\} \subseteq \gamma_C^{Must}\left(add_C^{(Must)}\left(c^{(A)}, t, env_C^{(A)}\right), env_V^{(A)}\right)$$

Proof. We distinguish two cases. In the first case $c^{(A)}$ and t are non-summary identifiers and in the second $c^{(A)}$ and / or t is a summary identifier. The first case is proven by Lemma 5 and the second case is proven by Lemma 7. We can therefore conclude, that the Theorem is proven for all abstract collections and for all *Tuple Identifiers*. □

5.2.3 Adding a Value at a given Key

This operation adds a value to a collection at a given key. We assume for this operation that the key was not previously in the collection.

For the concrete semantics this means, that first an *Element* e for the new collection element is created. Then e is added to the *Element Set* of collection c in the *Collection Environment* and finally the key k is assigned to the key and the value v is assigned to the value field of e in the *Value Environment*.

$$
\begin{array}{ll}
\textbf{addElement}: & (\textbf{Ref} \times \textbf{V} \times \textbf{V}) \to ((\textbf{Env}_\textbf{C} \times \textbf{Env}_\textbf{V}) \to (\textbf{Env}_\textbf{C} \times \textbf{Env}_\textbf{V})) \\
addElement & (c,k,v)(env_C, env_V) = (env_C^{(1)}, env_V^{(2)}): \\
 & e = createObject(T_C) \wedge \\
 & env_C^{(1)} = add_C(c, e, env_C) \wedge \\
 & env_V^{(1)} = assign_V((e, "key"), k, env_V) \wedge \\
 & env_V^{(2)} = assign_V((e, "value"), v, env_V^{(1)})
\end{array}
$$

where T_C is the type of the collection c.

The abstract semantic operation to add a key-value-pair to an abstract collection are identical for the *May* and the *Must Analysis*. First the *Tuple Identifier* t for the new collection element is retrieved. Then t is added to the *Abstract Element Set* of the abstract collection c and finally the key k is assigned to the key and the value v is assigned to the value field of t in the *Abstract Value Environment*.

$$
\begin{array}{ll}
\textbf{addElement}^{(\textbf{A})}: & (\textbf{HId} \times \textbf{Expression} \times \textbf{Expression}) \to \left(\textbf{Env}^{(\textbf{A})} \to \textbf{Env}^{(\textbf{A})}\right) \\
addElement^{(A)} & (c,k,v)(env_C, env_V) = (env_C^{(1)}, env_V^{(2)}): \\
 & t = createObject^{(A)}(T_C) \wedge \\
 & env_C^{(1)} = add_C^{(A)}(c, t, env_C) \wedge \\
 & env_V^{(1)} = assign_V^{(A)}((t, "key"), k, env_V) \wedge \\
 & env_V^{(2)} = assign_V^{(A)}((t, "value"), v, env_V^{(1)})
\end{array}
$$

Soundness

Since we only concatenate semantic operations for which we know, that they are sound in both the *May* and the *Must Analysis*, we can conclude that $addElement^{(A)}$ is sound as well.

5.2.4 Find Elements by Key

The operation $findElmentsByKey$ finds elements in a collection whose key field matches a given key.

This is a supporting operation, which does not need to be proven sound in the same way as other semantic operators. But we will show relationships between the abstract and the concrete operations that we can later use to prove other semantic operations.

The concrete semantics of $findElementsByKey$ takes a collection identifier c, a key k and an *Environment* (env_V, env_C). It looks for an *Element* e, such that e is in the *Element Set* of c and $(e, "key")$ equals k in the *Value Environment*. If it finds such an *Element* it returns a set only containing this *Element*. Otherwise it returns the empty set.

$$
\begin{array}{ll}
\textbf{findElementsByKey}: & (\textbf{Ref} \times \textbf{V}) \to ((\textbf{Env}_\textbf{C} \times \textbf{Env}_\textbf{V}) \to \mathcal{P}(\textbf{Elem})) \\
findElementsByKey & (c,k)(env_C, env_V) = \{e : e \in env_C(c) \wedge equals_V((e, "key"), k, env_V)\}
\end{array}
$$

May Analysis

In the *May Analysis* we want to find all the elements that might be in the collection and that might have the given key. Thus we want to over-approximate the elements in a collection that have the given key in any *Collection Environment*.

The $findElementsByKey^{(May)}$ function therefore returns for a key k all *Tuple Identifiers* that are in the *Abstract Element Set* of an abstract collection c and whose key field might be equal to k in the *Abstract Value Environment*. Furthermore we must also ensure, that the value field of the *Tuple Identifier* are not bottom in the *Abstract Value Environment*, since this would mean that there is no value stored at key k and it would not be sound to return such a *Tuple Identifier*.

$$\begin{array}{ll} \textbf{findElementsByKey}^{(\textbf{May})}: & (\textbf{HId} \times \textbf{Expression}) \to \left(\left(\textbf{Env}_{\textbf{C}}^{(\textbf{A})} \times \textbf{Env}_{\textbf{V}}^{(\textbf{A})}\right) \to \mathcal{P}(\textbf{TId})\right) \\ findElementsByKey^{(May)} & (c,k)(env_C, env_V) = \\ & \left\{ \begin{array}{ll} t: & t \in env_C(c) \wedge \\ & equals_V^{(A)}((t, \text{"}key\text{"}), k, env_V) \supseteq \{true\} \wedge \\ & notBottom(t, env_V) \end{array} \right\} \end{array}$$

We want to show, that the set of *Tuple Identifiers* returned by $findElementsByKey^{May}$ over-approximates the set of *Elements* that have the given key in any of the concrete environments abstracted by the abstract environment that was passed to $findElementsByKey^{May}$.

To be able to show this, we first need to prove the following Lemma.

Lemma 8. *The concretization of the* Abstract Element Set *of an abstract collection* $c^{(A)}$ *in an* Abstract Environment $\left(env_C^{(A)}, env_V^{(A)}\right)$ *over-approximates the* Element Sets *of any collection* $c \in \gamma_{HId}(c^{(A)})$ *in all* Collection Environments $env_C \in \gamma_C\left(env_C^{(A)}, env_V^{(A)}\right)$. *This holds for all* Abstract Environments $\left(env_C^{(A)}, env_V^{(A)}\right) \in Env^{(A)}$ *and for all abstract collections* $c^{(A)} \in HId$. *Furthermore this holds for the* May *and for the* Must Analysis $\left(\gamma_E \in \left\{\gamma_E^{May}, \gamma_E^{Must}\right\}, \gamma_C \in \left\{\gamma_C^{May}, \gamma_C^{Must}\right\}\right)$

$$\begin{array}{c} \left\{ env_C(c) : env_C \in \gamma_C\left(env_C^{(A)}, env_V^{(A)}\right) \right\} \\ \subseteq \\ \gamma_E\left(\left\{ t : t \in env_C^{(A)}\left(c^{(A)}\right) \wedge notBottom\left(t, env_V^{(A)}\right) \right\}\right) \end{array}$$

Proof. From Lemma 1 we know

$$\gamma_C\left(env_C^{(A)}, env_V^{(A)}\right) =$$

$$\left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E(E) \wedge \\ & E = \left\{ \begin{array}{l} t' : \\ t' \in env_C^{(A)}(i) \wedge \\ notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) - \{c\} \wedge \\ & c' \in \gamma_E(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \\ \cup \left\{ \begin{array}{ll} (c, c') : & c' \in \gamma_E(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

We can see that $env_C(c) \in \gamma_E\left(\left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\}\right)$. And from this follows directly

$$\left\{ env_C(c) : env_C \in \gamma_C\left(env_C^{(A)}, env_V^{(A)}\right) \right\} \subseteq \gamma_E\left(\left\{ t : t \in env_C^{(A)}\left(c^{(A)}\right) \wedge notBottom\left(t, env_V^{(A)}\right) \right\}\right)$$

□

With this Lemma we are now able to prove the previously described relationship between $findElementByKey$ and $findElementByKey^{May}$.

Theorem 3. *For each* Element $e \in \gamma_{TId}(t)$ *that is found by the concrete* $findElementsByKey$ *operation in any* Environment $(env_C, env_V) \in \gamma_{May}\left(env_C^{(A)}, env_V^{(A)}\right)$, *the* Tuple Identifier t, *that abstracts* e, *is in the result of* $findElementsByKey^{(May)}$ *in* $env^{(A)}$.

$$\begin{array}{c} \forall c \in \gamma_{HId}\left(c^{(A)}\right),\ k \in \gamma\left(k^{(A)}\right),\ k^{(A)} \in Expression, \\ c^{(A)} \in dom\left(env_C^{(A)}\right),\ \left(env_C^{(A)}, env_V^{(A)}\right) \in Env^{(A)} : \\ \bigcup \left\{ \begin{array}{ll} findElementsByKey(c, k, env_C, env_V) : & env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right) \wedge \\ & env_V \in \gamma_V\left(env_V^{(A)}\right) \end{array} \right\} \\ \subseteq \\ \bigcup_{t \in findElementsByKey^{(May)}\left(c^{(A)}, k^{(A)}, env_C^{(A)}, env_V^{(A)}\right)} \gamma_{TId}(t) \end{array}$$

Proof. We are going to show that if z is in the first set, it follows that z is in the second set which, by the definition of $\subseteq$, proves the Theorem.

$$z \in \bigcup \left\{ \begin{array}{ll} findElementsByKey(c,k,env_C,env_V): & env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right) \wedge \\ & env_V \in \gamma_V\left(env_V^{(A)}\right) \end{array} \right\}$$

First we apply the definition of $findByKey$.

$$z \in \bigcup \left\{ \begin{array}{ll} \{e' : e' \in env_C(c) \wedge equals_V((e', "key"), k, env_V)\}: & env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right) \wedge \\ & env_V \in \gamma_V\left(env_V^{(A)}\right) \end{array} \right\}$$

Using lemma 8 and the soundness of $equal_V^{(A)}$ it can be shown, that for all $env_V \in \gamma_V\left(env_V^{(A)}\right)$ the following holds:

$$\left\{\{e' : e' \in env_C(c) \wedge equals_V((e', "key"), k, env_V)\} : env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$$
$$\subseteq$$
$$\gamma_E^{May}\left(\left\{ \begin{array}{ll} t: & t \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom(t, env_V^{(A)}) \wedge \\ & \left(equals_V^{(A)}\left((t, "key"), k^{(A)}, env_V^{(A)}\right) \supseteq \{true\}\right) \end{array} \right\}\right)$$

Therefore we can conclude

$$z \in \bigcup \gamma_E^{May}\left(\left\{ \begin{array}{ll} t: & t \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom(t, env_V^{(A)}) \wedge \\ & \left(equals_V^{(A)}\left((t, "key"), k^{(A)}, env_V^{(A)}\right) \supseteq \{true\}\right) \end{array} \right\}\right)$$

We can then apply the definition of $findElementsByKey^{(May)}$

$$z \in \bigcup \gamma_E^{May}\left(findElementsByKey^{(May)}\left(c^{(A)}, k^{(A)}, env_C^{(A)}, env_V^{(A)}\right)\right)$$

And by the definition of γ_E^{May} we can write:

$$z \in \bigcup \left\{ x : x \subseteq \bigcup_{t \in findElementsByKey^{(May)}\left(c^{(A)}, k^{(A)}, env_C^{(A)}, env_V^{(A)}\right)} \gamma_{TId}(t) \right\}$$

Since $\bigcup\{X : X \subseteq S\} = S$ for all sets S, we can finally deduce

$$z \in \bigcup_{t \in findElementsByKey^{(May)}\left(c^{(A)}, k^{(A)}, env_C^{(A)}, env_V^{(A)}\right)} \gamma_{TId}(t)$$

□

Must Analysis

Finding elements in a collection by key in the *Must Analysis* means that we want to find all the elements that must be in the collection and that must have the given key. Thus we want to under-approximate the elements that have the given key in all *Collection Environments*.

The $findElementsByKey^{(Must)}$ function therefore returns for a key k all *Tuple Identifiers* that are in the *Abstract Element Set* of an abstract collection c and whose key field must be equal to k in the *Abstract Value Environment*. Analogous to $findElementsByKey^{(May)}$ we must ensure, that the value field of the *Tuple Identifier* is not bottom in the *Abstract Value Environment*.

$$\begin{array}{ll} \mathbf{findElementsByKey^{(Must)}}: & (\mathbf{HId} \times \mathbf{Expression}) \to \left(\left(\mathbf{Env_C^{(A)}} \times \mathbf{Env_V^{(A)}}\right) \to \mathcal{P}(\mathbf{TId})\right) \\ findElementsByKey^{(Must)} & (c,k)(env_C, env_V) = \\ & \left\{ \begin{array}{ll} t: & t \in env_C(c) \wedge \\ & equals_V^{(A)}((t, \text{"}key\text{"}), k, env_V) = \{true\} \wedge \\ & noitBottom(t, env_V) \end{array} \right\} \end{array}$$

We want to show that the set of *Tuple Identifiers* returned by $findElementsByKey^{Must}$ under-approximates the set of *Elements* that have the given key in all the concrete environments abstracted by the abstract environment that was passed to $findElementsByKey^{Must}$.

Theorem 4. *Every* Element*, that is in the concretization of the result of* $findElementsByKey^{(Must)}$ *in an* Abstract Environment $\left(env_C^{(A)}, env_V^{(A)}\right)$*, must be found in all concrete* Environments $(env_C, env_V) \in \gamma_{Must}\left(env_C^{(A)}, env_V^{(A)}\right)$ *by the concrete* $findElementsByKey$ *operation.*

$$\begin{array}{c} \forall c \in \gamma_{HId}\left(c^{(A)}\right),\ k \in \gamma\left(k^{(A)}\right),\ k^{(A)} \in Expression, \\ c^{(A)} \in dom\left(env_C^{(A)}\right),\ \left(env_C^{(A)}, env_V^{(A)}\right) \in Env^{(A)}: \\ \bigcup_{t \in findElementsByKey^{(Must)}\left(c^{(A)}, k^{(A)}, env_C^{(A)}, env_V^{(A)}\right)} \gamma_{TId}(t) \\ \subseteq \\ \bigcap \left\{ \begin{array}{ll} findElementsByKey(c,k,env_C,env_V): & env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right) \wedge \\ & env_V \in \gamma_V\left(env_V^{(A)}\right) \end{array} \right\} \end{array}$$

Proof. We are going to show that if z is in the first set, it follows that z is in the second set which, by the definition of $\subseteq$, proves the theorem.

$$z \in \bigcup_{t \in findElementsByKey^{(Must)}\left(c^{(A)}, k^{(A)}, env_C^{(A)}, env_V^{(A)}\right)} \gamma_{TId}(t)$$

Since $S = \bigcap\{X : X \supseteq S\}$ for all sets S, we can write

$$z \in \bigcap \left\{ x : x \supseteq \bigcup_{t \in findElementsByKey^{(Must)}\left(c^{(A)}, k^{(A)}, env_C^{(A)}, env_V^{(A)}\right)} \gamma_{TId}(t) \right\}$$

By applying the definition of γ_E^{Must} we get

$$z \in \bigcap \gamma_E^{Must}\left(findElementsByKey^{(Must)}\left(c^{(A)}, k^{(A)}, env_C^{(A)}, env_V^{(A)}\right)\right)$$

If we then use the definition of $findElementsByKey^{(Must)}$ we can conclude

$$z \in \bigcap \gamma_E^{Must}\left(\left\{\begin{array}{ll} t: & t \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom(t, env_V^{(A)}) \wedge \\ & \left(equals_V^{(A)}\left((t, \text{"}key\text{"}), k^{(A)}, env_V^{(A)}\right) = \{true\}\right) \end{array}\right\}\right)$$

By using lemma 8 and the soundness of $equal_V^{(A)}$ it can be shown that for all $env_V \in env_V^{(A)}$ the following holds:

$$\left\{\{e' : e' \in env_C(c) \wedge equals_V((e', \text{"}key\text{"}), k, env_V)\} : env_C \in \gamma_C^{Must}\left(env_C^{(A)}\right)\right\}$$
$$\subseteq$$
$$\gamma_E^{Must}\left(\left\{\begin{array}{ll} t: & t \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom(t, env_V^{(A)}) \wedge \\ & \left(equals_V^{(A)}\left((t, \text{"}key\text{"}), k^{(A)}, env_V^{(A)}\right) = \{true\}\right) \end{array}\right\}\right)$$

And therefore we can deduce

$$z \in \bigcap \left\{\begin{array}{lll} \{e' : e' \in env_C(c) \wedge equals_V((e', \text{"}key\text{"}), k, env_V)\} : & env_C \in \gamma_C^{Must}\left(env_C^{(A)}\right) \wedge \\ & env_V \in \gamma_V\left(env_V^{(A)}\right) \end{array}\right\}$$

By applying the definition of $findElementsByKey$ we finally get

$$z \in \bigcap \left\{\begin{array}{lll} findElementsByKey(c, k, env_C, env_V) : & env_C \in \gamma_C^{Must}\left(env_C^{(A)}\right) \wedge \\ & env_V \in \gamma_V\left(env_V^{(A)}\right) \end{array}\right\}$$

□

5.2.5 Find Elements by Value

The $findElementsByValue$ methods are defined analogously to the $findElementsByKey$ methods.

$\mathbf{findElementsByValue}:$	$(\mathbf{Ref} \times \mathbf{V}) \to ((\mathbf{Env_C} \times \mathbf{Env_V}) \to \mathcal{P}(\mathbf{Elem}))$
$findElementsByValue$	$(c, v)(env_C, env_V) =$
	$\{e : e \in env_C(c) \wedge equals_V((e, \text{"}value\text{"}), v, env_V)\}$

$\mathbf{findElementsByValue^{(May)}}:$	$(\mathbf{HId} \times \mathbf{Expression}) \to \left(\left(\mathbf{Env_C^{(A)}} \times \mathbf{Env_V^{(A)}}\right) \to \mathcal{P}(\mathbf{TId})\right)$
$findElementsByValue^{(May)}$	$(c, v)(env_C, env_V) =$
	$\left\{\begin{array}{ll} t: & t \in env_C(c) \wedge \\ & equals_V^{(A)}((t, \text{"}value\text{"}), v, env_V) \supseteq \{true\} \wedge \\ & notBottom(t, env_V) \end{array}\right\}$

$$\begin{array}{ll} \textbf{findElementsByValue}^{(\textbf{Must})}: & (\textbf{HId} \times \textbf{Expression}) \to \left(\left(\textbf{Env}_{\textbf{C}}^{(\textbf{A})} \times \textbf{Env}_{\textbf{V}}^{(\textbf{A})}\right) \to \mathcal{P}(\textbf{TId})\right) \\ findElementsByValue^{(Must)} & (c, v, env_C, env_V) = \\ & \left\{ \begin{array}{ll} t: & t \in env_C(c) \wedge \\ & equals_V^{(A)}((t, \text{"}value\text{"}), v, env_V) = \{true\} \wedge \\ & noitBottom(t, env_V) \end{array} \right\} \end{array}$$

5.2.6 Assume Keys not equal to k

This function is used to assume for a set of collection elements that their key field is not equal to a provided key k. It is an extension of the $assume_V$ operation and only operates on the *Abstract Value Environment*.

In the concrete semantics we define the $assumeNot$ operation recursively for a set of *Elements* and a key k. If the set of *Elements* is empty we can simply return the unchanged *Value Environment*. Otherwise we extract one *Element* e from the set. For this *Element* we assume that its key field is not equal to k. We then recursively call $assumeNot$ for the remaining *Elements*.

$$\begin{array}{ll} \textbf{assumeNot}: & (\mathcal{P}(\textbf{Elem}) \times \textbf{Value}) \to (\textbf{Env}_{\textbf{V}} \to \textbf{Env}_{\textbf{V}}) \\ assumeNot & (\emptyset, k)(env_V) = env_V \\ assumeNot & (E, k)(env_V) = env_V^{(2)}: \\ & e \in E \wedge \\ & env_V^{(1)} = assume_V(((e, \text{"}key\text{"}) \neq k), env_V) \wedge \\ & env_V^{(2)} = assumeNot(E - \{e\}, k, env_V^{(1)}) \end{array}$$

The abstracts semantics for the $assumeNot$ operation is identical for the *May* and the *Must Analysis*. It is defined analogously to the concrete $assumeNot$ operation.

$$\begin{array}{ll} \textbf{assumeNot}^{(\textbf{A})}: & (\mathcal{P}(\textbf{TId}) \times \textbf{Expression} \times \textbf{Env}_{\textbf{V}}^{(\textbf{A})}) \to \textbf{Env}_{\textbf{V}}^{(\textbf{A})} \\ assumeNot^{(A)} & (\emptyset, k, env_V) = env_V \\ assumeNot^{(A)} & (tuples, k, env_V) = env_V^{(2)}: \\ & t \in tuples \wedge \\ & env_V^{(1)} = assume_V^{(A)}((t, \text{"}key\text{"}) \neq k, env_V) \wedge \\ & env_V^{(2)} = assumeNot^{(A)}(tuples - \{t\}, k, env_V^{(1)}) \end{array}$$

Soundness

Since we are only concatenating $assume_V$ operations and we know from the specification of the value domain, that this operation has to be sound, we can conclude that $assumeNot$ is sound as well.

5.2.7 Removing a Value at a given Key

This operation removes a value from a collection at a given key.

In the concrete semantics we remove an element at key k from collection c as follows: For each *Element* e that is in the *Element Set* of c, we assume $(e, \text{"}key\text{"}) \neq k$ in the *Value Environment*.

$$\begin{array}{ll} \textbf{removeElement}: & (\textbf{Ref} \times \textbf{V}) \to ((\textbf{Env}_{\textbf{C}} \times \textbf{Env}_{\textbf{V}}) \to (\textbf{Env}_{\textbf{C}} \times \textbf{Env}_{\textbf{V}})) \\ removeElement & (c, k)(env_C, env_V) = (env_C, assumeNot(env_C(c), k, env_V) \end{array}$$

The abstract semantics to remove an element at k from an abstract collection c is defined similar. For each *Tuple Identifier* t that is in the *Abstract Element Set* of c, we assume $(t, \text{"}key\text{"}) \neq k$ in

the *Abstract Value Environment*.

$$\begin{array}{ll} \textbf{removeElement}^{(\mathbf{A})}: & (\mathbf{HId} \times \mathbf{Expression}) \to \left(\left(\mathbf{Env_C^{(A)}} \times \mathbf{Env_V^{(A)}}\right) \to \left(\mathbf{Env_C^{(A)}} \times \mathbf{Env_V^{(A)}}\right)\right) \\ removeElement^{(A)} & (c,k)(env_C, env_V) = (env_C, assumeNot^{(A)}(env_C(c), k, env_V) \end{array}$$

Notice that if the only possible value of the key field of a *Tuple Identifier* is k, then after the remove operation, the key field has the value bottom in the *Value Domain*. This means that also the value should no longer be in the abstract collection, since there is no possible key that points to it. Because the $findElementByKey$ and the concretization functions handle this case correctly, we do not need to remove the *Tuple Identifier* from the *Abstract Element Set* here.

Soundness

Since we only concatenate operations, which we have already proven to be sound, we can conclude that $removeElement^{(A)}$ is sound as well.

5.2.8 Decrease Keys

This function decreases all the keys in a collection that are greater than a given key.

In the concrete semantics we first collect all *Elements* e in the *Element Set* of the collection c such that the value of $(e, "key")$ in the *Value Environment* is greater than the provided value k. We then recursively decrease the value of the key field of all these *Elements* in the *Value Environment* by one.

$$\begin{array}{ll} \textbf{decreaseKeys}: & (\mathbf{Ref} \times \mathbf{V}) \to ((\mathbf{Env_C} \times \mathbf{Env_V}) \to (\mathbf{Env_C} \times \mathbf{Env_V})) \\ decreaseKeys & (c,k)(env_C, env_V) = decreaseKeys_V(E, env_V) \\ & \text{where } E = \left\{ \begin{array}{ll} e: & e \in env_C(c) \wedge \\ & assume_V\left((e, "key") > k, env_V\right) \neq \bot \end{array} \right\} \end{array}$$

$$\begin{array}{ll} \textbf{decreaseKeys}_\mathbf{V}: & (\mathcal{P}(\mathbf{Elem})) \to (\mathbf{Env_V} \to \mathbf{Env_V}) \\ decreaseKeys_V & (\emptyset)(env_V) = env_V \\ decreaseKeys_V & (E)(env_V) = env_V^{(2)}: \\ & e \in E \wedge \\ & env_V^{(1)} = assign_V((e, "key"), (e, "key") - 1, env_V) \wedge \\ & env_V^{(2)} = decreaseKeys_V\left(E - \{e\}, env_V^{(1)}\right) \end{array}$$

In the abstract semantics we first collect all *Tuple Identifiers* t in the *Abstract Element Set* of the abstract collection c such that the value of $(t, "key")$ in the *Abstract Value Environment* may be greater than the provided value k. We are then decreasing the value of the key field of all these *Tuple Identifiers* in the *Abstract Value Environment* by one recursively.

$$\begin{array}{ll} \textbf{decreaseKeys}^{(\mathbf{A})}: & (\mathbf{HId} \times \mathbf{Expression}) \to \left(\left(\mathbf{Env_C^{(A)}} \times \mathbf{Env_V^{(A)}}\right) \to \mathbf{Env_V^{(A)}}\right) \\ decreaseKeys^{(A)} & (c,k)(env_C, env_V) = decreaseKeys_V^{(A)}(T, env_V) \\ & \text{where } T = \left\{ \begin{array}{ll} t: & t \in env_C(c) \wedge \\ & assume_V^{(A)}\left((t, "key") > k, env_V\right) \neq \bot \end{array} \right\} \end{array}$$

$$\begin{array}{ll}
\mathbf{decreaseKeys_V^{(A)}}: & (\mathcal{P}(\mathbf{TId})) \to \left(\mathbf{Env_V^{(A)}} \to \mathbf{Env_V^{(A)}}\right) \\
decreaseKeys_V^{(A)} & (\emptyset)(env_V) = env_V \\
decreaseKeys_V^{(A)} & (tuples)(env_V) = env_V^{(2)}: \\
 & t \in tuples \wedge \\
 & env_V^{(1)} = assign_V^{(A)}((t, "key"), (t, "key") - 1, env_V) \wedge \\
 & env_V^{(2)} = decreaseKeys_V^{(A)}\left(tuples - \{t\}, env_V^{(1)}\right)
\end{array}$$

Soundness

Analogous to the $findElementsByKey$ function we can prove that the set of *Tuple Identifiers* passed to $decreaseKeys_V^{(A)}$ over-approximates the set of *Elements* passed to $decreaseKeys_V$. The $decreaseKeys_V$ operation concatenates $assign_V$ for each element in the set of passed *Elements / Tuple Identifiers*. We already know that the $assign_V$ operation is sound.

We can therefore conclude that the $dcecreaseKeys$ operation is sound.

5.2.9 Assigning an Empty Collection

This operation assigns an empty collection to a collection identifier in the collection environment.

For the concrete semantics this means, that an empty *Element Set* $\emptyset$ is assigned to a collection c.

$$\begin{array}{ll}
\mathbf{assignEmpty_C}: & \mathbf{Ref} \to (\mathbf{Env_C} \to \mathbf{Env_C}) \\
assignEmpty_C & (c)(env_C) = env_C[c \to \emptyset]
\end{array}$$

In the abstract semantics we need to distinguish whether the abstract collection to which we assign the empty set is a summary collection or not. For non-summary collections we can simply assign the empty *Abstract Element Set* to the abstract collection. But if the abstract collection is a summary collection this would not be sound. We need to handle this case for the *May* and the *Must Analysis* individually.

The *May Analysis* represents all collection elements that might be in a collection. If an abstract collection is a summary collection, representing multiple distinct collections, the *May Analysis* represents all elements that may be in any of these collections. Since we can not know which of the elements in the abstract summary collection abstract the elements of the emptied concrete collection, we have to keep all elements in the abstract collection.

$$\begin{array}{ll}
\mathbf{assignEmpty_C^{(May)}}: & \mathbf{HId} \to \left(\mathbf{Env_C^{(A)}} \to \mathbf{Env_C^{(A)}}\right) \\
assignEmpty_C^{(May)} & (c)(env_C) = \begin{cases} env_C[c \to \emptyset] & \text{if } |\gamma_{HId}(c)| = 1 \\ env_C & \text{otherwise} \end{cases}
\end{array}$$

The *Must Analysis* represents all elements that certainly are in a collection. For a summary collection this means it contains all elements that certainly are contained in all collections represented by the summary collection. Since one of the concrete collections is now empty, no element is certainly contained in all concrete collections and we need to remove all elements from the abstract collection (Assigning an empty *Abstract Element Set* to the abstract collection).

$$\begin{array}{ll}
\mathbf{assignEmpty_C^{(Must)}}: & \mathbf{HId} \to \left(\mathbf{Env_C^{(A)}} \to \mathbf{Env_C^{(A)}}\right) \\
assignEmpty_C^{(Must)} & (c)(env_C) = env_C[c \to \emptyset]
\end{array}$$

Soundness - May Analysis

In the abstract semantics of the *May Analysis* we distinguish between summary collections and non summary collections.

We are going to prove the soundness for these two cases and can then proof the soundness $assignEmpty_C^{(May)}$ for all collections.

Case 1: Non-Summary Collections

In this case we assume that the abstract collection is a non-summary collection. This means $|\gamma_{HId}\left(c^{(A)}\right)| = 1$.

We will first prove, that the empty *Abstract Element Set* over-approximates the empty *Element Set*. We can then use this to show that $assignEmpty_C^{(May)}$ is sound for all non-summary collections.

Lemma 9. *The empty* Element Set $\emptyset$ *is in the concretization of the empty* Abstract Element Set $\emptyset$*.*

$$\emptyset \in \gamma_E^{May}(\emptyset)$$

Proof. γ_E^{May} is defined as $\gamma_E^{May}\left(c^{(A)}\right) = \left\{x : x \subseteq \bigcup_{t \in c^{(A)}} \gamma_{HId}(t)\right\}$. The set retrieved from $\gamma_E^{May}(\emptyset)$ is therefore the set containing all the subsets of $\bigcup_{t \in \emptyset} \gamma_{HId}(t)$. Since the empty set is a subset of every set, $\gamma_E^{May}(\emptyset)$ must contain the empty set. □

With this Lemma we are now able to prove the soundness of the $assignEmpty^{(May)}$ operation for non-summary collections.

Lemma 10. *Emptying an abstract* ***non-summary*** *collection* $c^{(A)}$ *in an* Abstract Environment $\left(env_C^{(A)}, env_V^{(A)}\right)$ *over-approximates emptying a concrete collection* $c \in \gamma_{HId}\left(c^{(A)}\right)$ *in all* Collection Environments $env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)$*. This holds for all abstract* ***non-summary*** *collections* $c^{(A)} \in dom\left(env_C^{(A)}\right)$ *and for all* Abstract Environments $\left(env_C^{(A)}, env_V^{(A)}\right) \in Env^{(A)}$*.*

$$\left\{assignEmpty_C(c, env_C) : env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$$
$$\subseteq$$
$$\gamma_C^{May}\left(assignEmpty_C^{(May)}\left(c^{(A)}, env_C^{(A)}\right), env_V^{(A)}\right)$$

Proof. We are going to show that if $z \in \left\{assignEmpty_C(c, env_C) : env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$ it follows that $z \in \gamma_C^{May}\left(assignEmpty_C^{(May)}\left(c^{(A)}, env_C^{(A)}\right), env_V^{(A)}\right)$ which, by the definition of $\subseteq$, proves the lemma.

$$z \in \left\{assignEmpty_C(c, env_C) : env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$$

First, we apply the definition of $assignEmpty_C$.

$$z \in \left\{ env_C[c \to \emptyset] : env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right) \right\}$$

We use lemma 1 to isolate the part of $\gamma_C^{(May)}$ that generates the entries for the concrete collection c in the *Collection Environments*. In this case we furthermore know that $c^{(A)}$ is not a summary collection and therefore the only element in $\gamma_{HId}(c^{(A)})$ is the concrete collection c. From this follows that $\gamma_{HId}\left(c^{(A)}\right) - \{c\} = \emptyset$ which allows us to further adapt $\gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)$ as we have already seen in the proof of lemma 5.

$$\gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right) =$$

$$\left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\begin{array}{l} \bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{l} t' : \\ t' \in env_C^{(A)}(i) \wedge \\ notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \\ \cup \left\{ \begin{array}{ll} (c, c') : & c' \in \gamma_E^{May}(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

We can now replace $\gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)$ with this expression.

$$z \in \left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\begin{array}{l} \bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{l} t' : \\ t' \in env_C^{(A)}(i) \wedge \\ notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \\ \cup \{(c, \emptyset)\} \end{array} \right) \end{array} \right\}$$

From lemma 9 we know that $\emptyset \in \gamma_E^{May}(\emptyset)$ and therefore $\{\emptyset\} \subseteq \gamma_E^{May}(\emptyset)$. This allows us to conclude:

$$z \in \left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\begin{array}{l} \bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{l} t' : \\ t' \in env_C^{(A)}(i) \wedge \\ notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \\ \cup \left\{ (c, c') : c' \in \gamma_E^{May}(\emptyset) \right\} \end{array} \right) \end{array} \right\}$$

Similar to the proof of Theorem 1 we can simplify this to

$$z \in \gamma_C^{May}\left(env_C^{(A)}\left[c^{(A)} \to \emptyset\right], env_V^{(A)}\right)$$

But since in this case we know that $c^{(A)}$ is a summary collection, this is equal to $z \in \gamma_C^{May}\left(assignEmpty_C^{(May)}\left(c^{(A)}, env_C^{(A)}\right), env_V^{(A)}\right)$ which is exactly what we wanted to prove. □

Case 2: Summary Collections

In this case we assume that the collection is a summary collection. This means $|\gamma_{HId}\left(c^{(A)}\right)| > 1$. We will first prove, that the empty *Element Set* is abstracted by any *Abstract Element Set*. We can then use this to show, that $assignEmpty_C^{(May)}$ is sound for summary collections.

Lemma 11. *The empty* Element Set $\emptyset$ *is abstracted by every* Abstract Element Set $E^{(A)} \in \mathcal{P}(TId)$.

$$\emptyset \in \gamma_E^{May}(E^{(A)})$$

Proof. γ_E^{May} is defined as $\gamma_E^{May}\left(E^{(A)}\right) = \left\{x : x \subseteq \bigcup_{t \in E^{(A)}} \gamma_{HId}(t)\right\}$. The *Element Sets* contained in $\gamma_E^{May}\left(E^{(A)}\right)$ are all the subsets of $\bigcup_{t \in E^{(A)}} \gamma_{HId}(t)$. Because the empty set is a subset of every set we can conclude that $\emptyset \in \gamma_E^{May}\left(E^{(A)}\right)$ holds for every *Abstract Element Set* $E^{(A)}$. □

With this Lemma we are now able to prove the soundness of the $assignEmpty^{(May)}$ operation for summary collections.

Lemma 12. *Emptying an abstract* ***summary*** *collection* $c^{(A)}$ *in an* Abstract Environment $\left(env_C^{(A)}, env_V^{(A)}\right)$ *over-approximates emptying a concrete collection* $c \in \gamma_{HId}\left(c^{(A)}\right)$ *in all* Collection Environments $env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)$. *This holds for all abstract* ***summary*** *collections* $c^{(A)} \in dom\left(env_C^{(A)}\right)$ *and for all* Abstract Environments $\left(env_C^{(A)}, env_V^{(A)}\right) \in Env^{(A)}$.

$$\left\{assignEmpty_C(c, env_C) : env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)\right\} \subseteq \gamma_C^{May}\left(assignEmpty_C^{(May)}\left(c^{(A)}, env_C^{(A)}\right), env_V^{(A)}\right)$$

Proof. We are going to show that if $z \in \left\{assignEmpty_C(c, env_C) : env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$ it follows that $z \in \gamma_C^{May}\left(assignEmpty_C^{(May)}\left(c^{(A)}, env_C^{(A)}\right), env_V^{(A)}\right)$ which, by the definition of $\subseteq$, proves the Lemma.

$$z \in \left\{assignEmpty_C(c, env_C) : env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$$

First, we apply the definition of $assignEmpty_C$

$$z \in \left\{env_C[c \to \emptyset] : env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$$

We use Lemma 1 to isolate the part of γ_C^{May} that generates the entries for the concrete collection c in the *Collection Environments*.

$$\gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right) =$$

$$\left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}(i) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) - \{c\} \wedge \\ & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \\ \cup \left\{ \begin{array}{ll} (c, c') : & c' \in \gamma_E^{May}(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

We can use this to restate the expression.

$$z \in \left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}(i) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) - \{c\} \wedge \\ & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \\ \cup \{(c, \emptyset)\} \end{array} \right) \end{array} \right\}$$

From Lemma 11 we know that $\emptyset \in \gamma_E^{May}\left(env_C^{(A)}\left(c^{(A)}\right)\right)$ and therefore $\{\emptyset\} \subseteq \gamma_E^{May}\left(env_C^{(A)}\left(c^{(A)}\right)\right)$ and we can deduce:

$$z \in \left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}(i) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) - \{c\} \wedge \\ & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \\ \cup \left\{ \begin{array}{ll} (c, c') : & c' \in \gamma_E^{May}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

Similar to the proof of Theorem 1 we can simplify this to

$$z \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)$$

But since in this case $c^{(A)}$ is a summary collection, this is equal to
$z \in \gamma_C^{May}\left(assignEmpty_C^{(May)}\left(c^{(A)}, env_C^{(A)}\right), env_V^{(A)}\right)$ which is exactly what we wanted to prove. □

With these Lemmas we are now able to prove that $assignEmpy_C^{(May)}$ is sound for all collections.

Theorem 5. *Emptying an abstract collection* $c^{(A)}$ *in an* Abstract Environment $\left(env_C^{(A)}, env_V^{(A)}\right)$ *over-approximates emptying a concrete collection* $c \in \gamma_{HId}\left(c^{(A)}\right)$ *in all* Collection Environments $env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)$. *This holds for all abstract collections* $c^{(A)} \in dom\left(env_C^{(A)}\right)$ *and for all* Abstract Environments $\left(env_C^{(A)}, env_V^{(A)}\right) \in Env^{(A)}$.

$$\left\{assignEmpty_C(c, env_C) : env_C \in \gamma_C^{May}\left(env_C^{(A)}, env_V^{(A)}\right)\right\} \subseteq \gamma_C^{May}\left(assignEmpty_C^{(May)}\left(c^{(A)}, env_C^{(A)}\right), env_V^{(A)}\right)$$

Proof. We distinguish the two cases that $c^{(A)}$ is a non-summary collection and that $c^{(A)}$ is a summary collection. The first case is proven by Lemma 10 and the second case is proven by Lemma 12. We can therefore conclude, that the Theorem is proven for all abstract collections. □

Soundness - Must Analysis

The soundness proof of $assignEmpty_C^{(Must)}$ does not distinguish between summary and non-summary collections. We therefore have to prove only one case.

We will first show that an empty abstract collection abstracts every possible concrete collection in the *Must Analysis*. We can then show that $assignEmpty_C^{(Must)}$ is sound for complete *Collection Environments*.

Lemma 13. *Every* Element Set $E \in \mathcal{P}(Elem)$ *is in the concretization of the empty* Abstract Element Set $\emptyset$.

$$E \in \gamma_E^{Must}(\emptyset)$$

Proof. By the definition of γ_E^{Must} we know that $\gamma_E^{Must}(\emptyset) = \left\{x : x \supseteq \bigcup_{t\in\emptyset} \gamma_{HId}(t)\right\}$. And because $\bigcup_{t\in\emptyset} \gamma_{HId}(t) = \emptyset$ the sets contained in the set retrieved from $\gamma_E^{Must}(\emptyset)$ are all the supersets of the empty set. Since every set is a superset of the empty set, we can conclude that $E \in \gamma_E^{Must}(\emptyset)$ holds for every *Element Set* E. □

We can now use this Lemma to show the soundness of $assignEmpty_C^{(Must)}$ for complete collection environments.

Theorem 6. *Emptying an abstract collection* $c^{(A)}$ *in an* Abstract Environment $\left(env_C^{(A)}, env_V^{(A)}\right)$ *over-approximates emptying a concrete collection* $c \in \gamma_{HId}\left(c^{(A)}\right)$ *in all* Collection Environments $env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)$. *This holds for all abstract collections* $c^{(A)} \in dom\left(env_C^{(A)}\right)$ *and for all* Abstract Environments $\left(env_C^{(A)}, env_V^{(A)}\right) \in Env^{(A)}$.

$$\left\{assignEmpty_C(c, env_C) : env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)\right\} \subseteq \gamma_C^{Must}\left(assignEmpty_C^{(Must)}\left(c^{(A)}, env_C^{(A)}\right), env_V^{(A)}\right)$$

Proof. We are going to show that if $z \in \left\{assignEmpty_C(c, env_C) : env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$ it follows that $z \in \gamma_C^{Must}\left(assignEmpty_C^{(Must)}\left(c^{(A)}, env_C^{(A)}\right), env_V^{(A)}\right)$ which, by the definition of $\subseteq$, proves the Theorem.

$$z \in \left\{assignEmpty_C(c, env_C) : env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$$

First, we apply the definition of $assignEmpty_C$.

$$z \in \left\{env_C[c \to \emptyset] : env_C \in \gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right)\right\}$$

We use Lemma 1 to isolate the part of γ_C^{Must} that generates the entries for the concrete collection c in the *Collection Environments*.

$$\gamma_C^{Must}\left(env_C^{(A)}, env_V^{(A)}\right) =$$

$$\left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \left\{c^{(A)}\right\}} \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{Must}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}(i) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r', c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) - \{c\} \wedge \\ & c' \in \gamma_E^{Must}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \\ \cup \left\{ \begin{array}{ll} (c, c') : & c' \in \gamma_E^{Must}(E) \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \end{array} \right) \end{array} \right\}$$

We can use this to restate the expression.

$$z \in \left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \{c^{(A)}\}} \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{Must}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}(i) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) - \{c\} \wedge \\ & c' \in \gamma_E^{Must}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}\left(c^{(A)}\right) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \\ \cup \{(c, \emptyset)\} \end{array} \right) \end{array} \right\}$$

Because we know from Lemma 13 that any *Element Set* is in $\gamma_E^{Must}(\emptyset)$, we can deduce:

$$z \in \left\{ \begin{array}{l} [a \to b : (a,b) \in x] : \\ x \subseteq \left(\begin{array}{l} \left(\bigcup_{i \in dom\left(env_C^{(A)}\right) - \{c^{(A)}\}} \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}(i) \wedge \\ & c' \in \gamma_E^{Must}(E) \wedge \\ & E = \left\{ \begin{array}{ll} t' : & t' \in env_C^{(A)}(i) \wedge \\ & notBottom\left(t', env_V^{(A)}\right) \end{array} \right\} \end{array} \right\} \right) \\ \cup \left\{ \begin{array}{ll} (r',c') : & r' \in \gamma_{HId}\left(c^{(A)}\right) - \{c\} \wedge \\ & c' \in \gamma_E^{Must}(\emptyset) \wedge \end{array} \right\} \\ \cup \left\{ \begin{array}{ll} (c,c') : & c' \in \gamma_E^{Must}(\emptyset) \end{array} \right\} \end{array} \right) \end{array} \right\}$$

Similar to the proof of Theorem 1 we can simplify this to

$$z \in \gamma_C^{Must}\left(env_C^{(A)}\left[c^{(A)} \to \emptyset\right], env_V^{(A)}\right)$$

But this is equal to $z \in \gamma_C^{Must}\left(assignEmpty_C^{(Must)}\left(c^{(A)}, env_C^{(A)}\right), env_V^{(A)}\right)$ which is exactly what we wanted to prove. □

5.3 Create Collection

After we have introduced the basic operations, we can now use them to define the collection operations of TouchDevelop. We start with the operation to create a new collection. An empty collection can be created using the collections module of TouchDevelop as shown in listing 5.1.

Listing 5.1: Create Link collection

```
1 var c := collections -> create link collection
```

In the concrete semantics we first create a *Reference* c for the new collection and then assign an empty *Element Set* to c in the *Collection Environment*.

createCollection :	$\mathbf{T_C} \to ((\mathbf{Env_C} \times \mathbf{Env_V}) \to (\mathbf{Env_C} \times \mathbf{Env_V}))$
$createCollection$	$(t_C)(env_C, env_V) = (env_C^{(1)}, env_V^{(1)}) :$
	$c = createObject(t_C) \wedge$
	$env_C^{(1)} = assignEmpty_C(c, env_C)$

t_C denotes the type of the new collection.

The abstract semantics to create a new collection is defined identically for the *May* and for the *Must Analysis*. We first retrieve a *Heap Identifier* c for the new collection and then assign an empty *Abstract Element Set* to c in the *Abstract Collection Environment*.

$$\begin{array}{ll} \textbf{createCollection}^{(\mathbf{A})}: & \mathbf{T_C} \to \left(\left(\mathbf{Env_C^{(A)}} \times \mathbf{Env_V^{(A)}}\right) \to \left(\mathbf{Env_C^{(A)}} \times \mathbf{Env_V^{(A)}}\right)\right) \\ createCollection^{(A)} & (t_C)(env_C, env_V) = (env_C^{(1)}, env_V^{(1)}): \\ & c = createObject^{(A)}(t_C) \wedge \\ & env_C^{(1)} = assignEmpty_C^{(A)}(c, env_C) \end{array}$$

Notice that $assignEmpty_C^{(A)}$ corresponds to $assignEmpty_C^{(May)}$ in the *May Analysis* and to $assignEmpty_C^{(Must)}$ in the *Must Analysis*.

5.3.1 Soundness

Since we only concatenate operations, that were already proven sound for both the *May* and the *Must Analysis*, we can conclude that $createCollection^{(A)}$ is sound.

5.4 Contains Key

The $containsKey$ operation checks whether a given key is contained in a collection or not.

In the concrete semantics we define for a single environment, that the key k is in the collection c if there exists an *Element* e in the *Element Set* of c, such that $(e, "key")$ is equal to k in the *Value Environment*.

$$\begin{array}{ll} \textbf{containsKey}: & (\mathbf{Ref} \times \mathbf{V}) \to ((\mathbf{Env_C}, \mathbf{Env_V}) \to \{\mathbf{true}, \mathbf{false}\}) \\ containsKey & (c,k)(env_C, env_V) = \begin{cases} true & \text{if } |F| > 0 \\ false & \text{otherwise} \end{cases} \\ & \text{where } F = findElementsByKey(c, k, env_C, env_V) \end{array}$$

Now let's look at the result of the operation, if it is applied on a set of environments.

$$\begin{array}{ll} \textbf{containsKey}: & (\mathbf{Ref} \times \mathbf{V}) \to (\mathcal{P}(\mathbf{Env}) \to \mathcal{P}(\{\mathbf{true}, \mathbf{false}\})) \\ containsKey & (c,k)(envs) = \{containsKey(c, k, env_C, env_V) : (env_C, env_V) \in envs\} \end{array}$$

Three results are possible:

- If we get the result $\{true\}$, then k must be in c, since in every *Environment* it is contained in c.
- If we get the result $\{false\}$, then k is certainly not contained in c, since in every *Environment* it is not contained in c.
- If we get the result $\{true, false\}$, then we can not say whether k is in the collection or not, since in some *Environments* it is contained in c and in some it is not.

Let's now look at how we can abstract this operation: Remember that the *May Analysis* only tracks which elements may be in a collection and is therefore only able to determine if an element is not in the collection. In contrast, the *Must Analysis* only tracks which elements must be in a collection and is therefore only able to determine whether an element is in the collection.

We will define the abstract semantics of $containsKey$ for the *May Analysis* and the *Must Analysis* individually. And to gain as much precision as possible we will also define an operation

that operates on the cartesian product of the domains of the *May* and the *Must Analysis*. This operation is used when we use both the *May* and the *Must Analysis* to analyze a script.

Boolean Domain

As we have seen, the result of *containsKey* is a set of boolean values ($\mathcal{P}(\{true, false\})$). In the abstract semantics the result of $containsKey^{(A)}$ will be an abstract boolean value. These abstract boolean values are defined as $B^{(A)} = \{\top, true, false, \bot\}$.

The function γ_B defines how they are concretized:

$$\gamma_B : B^{(A)} \to \mathcal{P}(\{true, false\})$$
$$\gamma_B(b) = \begin{cases} \emptyset & \text{if } b = \bot \\ \{true\} & \text{if } b = true \\ \{false\} & \text{if } b = false \\ \{true, false\} & \text{if } b = \top \end{cases}$$

5.4.1 Contains Key - May Analysis

The contains key operation for the *May Analysis* is able to tell whether an element whether an element with the given key is certainly not in the collection. This is the case, if no element that possibly is in the collection can have the given key.

The abstract semantics for the *containsKey* operation in the *May Analysis* therefore looks for a *Tuple Identifier* t in the *Abstract Element Set* of c such that $(t, \text{"}key\text{"})$ may be equal to the provided key k in the *Abstract Value Environment*. If it finds such a *Tuple Identifier*, this means that there might be an element with key k in the collection and therefore it returns top. But if it does not find such a *Tuple Identifier*, then there can be no element with key k in the collection and it returns $false$.

$$\mathbf{containsKey^{(May)}} : \quad (\mathbf{HId} \times \mathbf{Expression}) \to \left(\left(\mathbf{Env_C^{(A)}} \times \mathbf{Env_V^{(A)}}\right) \to \mathbf{B^{(A)}}\right)$$
$$containsKey^{(May)} \quad (c, k)(env_C, env_V) = \begin{cases} false & \text{if } |F| = 0 \\ \top & \text{otherwise} \end{cases}$$
$$\text{where } F = findElementsByKey^{(May)}(c, k, env_C, env_V)$$

Soundness

We prove the soundness of $containsKey^{(May)}$ by showing that it over-approximates the concrete $containsKey$ operation.

Theorem 7. *Applying* $containsKey^{(May)}$ *on an* Abstract Environment $\left(env_C^{(A)}, env_V^{(A)}\right)$ *with an abstract collection* $c^{(A)}$ *and an abstract key* $k^{(A)}$ *over-approximates applying* $containsKey$ *on all* Environments $env \in \gamma_{May}\left(env_C^{(A)}, env_V^{(A)}\right)$ *with any collection* $c \in \gamma_{HId}\left(c^{(A)}\right)$ *and any concrete key* $k \in \gamma\left(k^{(A)}\right)$*. This holds for all abstract collections* $c^{(A)} \in dom\left(env_C^{(A)}\right)$*, for all abstract keys* $k^{(A)} \in Expression$ *and for all* Abstract Environments $\left(env_C^{(A)}, env_V^{(A)}\right) \in Env^{(A)}$*.*

$$\begin{gathered}\left\{containsKey(c,k,env) : env \in \gamma_{May}\left(env^{(A)}\right)\right\} \\ \subseteq \\ \gamma_B\left(containsKey^{(May)}\left(c^{(A)}, k^{(A)}, env^{(A)}\right)\right)\end{gathered}$$

Proof. In the function $containsKey^{(May)}$ we distinguish two cases:

- The result of $findElementsByKey^{(May)}$ is the empty set ($|F| = 0$).
- The result of $findElementsByKey^{(May)}$ is not the empty set.

We are showing for both cases that the Theorem holds.

Case 1: In the first case we assume, that the result of $findElementsByKey^{(May)}$ is the empty set. In Theorem 3 we have already shown, that the concretization of the result of $findElementsByKey^{(May)}$ is a super set of the result of $findElementsByKey$. And because in this case the result of $findElementsByKey^{(May)}$ is the empty set, the result of $findElementsByKey$ must also be the empty set in all environments $env \in \gamma_{May}\left(env^{(A)}\right)$.

Therefore we know that $containsKey$ returns false in every environment $env \in \gamma_{May}\left(env^{(A)}\right)$. And from this follows that

$$\left\{containsKey(c,k,env) : env \in \gamma_{May}\left(env^{(A)}\right)\right\} = \{false\}$$

But we also know, that in this case $containsKey^{(May)}\left(c^{(A)}, k^{(A)}, env^{(A)}\right) = false$. And because $\gamma_B(false) = \{false\}$ we can conclude for the first case

$$\left\{containsKey(c,k,env) : env \in \gamma_{May}\left(env^{(A)}\right)\right\} \subseteq \gamma_B\left(containsKey^{(May)}\left(c^{(A)}, k^{(A)}, env^{(A)}\right)\right)$$

.

Case 2: In the second case $containsKey^{(May)}\left(c^{(A)}, k^{(A)}, env^{(A)}\right) = \top$. And Since $\gamma_B(\top) = \{true, false\}$, we can conclude

$$\gamma_B\left(containsKey^{(May)}\left(c^{(A)}, k^{(A)}, env^{(A)}\right)\right) \supseteq \left\{containsKey(c,k,env) : env \in \gamma_{May}\left(env^{(A)}\right)\right\}$$

Because we have proven the Theorem for both cases, we can deduce that $containsKey^{(May)}$ is sound. □

5.4.2 Contains Key - Must Analysis

The contains key operation for the *Must Analysis* is able to tell whether an element with the given key is certainly contained in the collection. This is the case, when an element that must be in the collection has the requested key.

The abstract semantics for the *containsKey* operation in the *Must Analysis* therefore looks for a *Tuple Identifier* t in the *Abstract Element Set* of c such that $(t, \text{"}key\text{"})$ must be equal to the provided key k in the *Abstract Value Environment*. If it finds such a *Tuple Identifier*, this means that there must be an element with key k in the collection and therefore it returns *true*. But if it does not find such a *Tuple Identifier*, then it can not tell whether an element with key k is in the collection and returns top.

$$\begin{aligned}
&\mathbf{containsKey^{(Must)}}: && (\mathbf{HId} \times \mathbf{Expression}) \to \left(\left(\mathbf{Env_C^{(A)}} \times \mathbf{Env_V^{(A)}}\right) \to \mathbf{B^{(A)}}\right) \\
&containsKey^{(Must)} && (c,k)(env_C, env_V) = \begin{cases} true & \text{if } |F| > 0 \\ \top & \text{otherwise} \end{cases} \\
& && \text{where } F = findElementsByKey^{(Must)}(c, k, env_C, env_V)
\end{aligned}$$

Soundness

We prove the soundness of $containsKey^{(Must)}$ by showing that it over-approximates the concrete *containsKey* operation.

Theorem 8. *Applying* $containsKey^{(Must)}$ *on an* Abstract Environment $\left(env_C^{(A)}, env_V^{(A)}\right)$ *with an abstract collection* $c^{(A)}$ *and an abstract key* $k^{(A)}$ *over-approximates applying containsKey on all* Environments $env \in \gamma_{Must}\left(env_C^{(A)}, env_V^{(A)}\right)$ *with any collection* $c \in \gamma_{HId}\left(c^{(A)}\right)$ *and any concrete key* $k \in \gamma\left(k^{(A)}\right)$*. This holds for all abstract collections* $c^{(A)} \in dom\left(env_C^{(A)}\right)$*, for all abstract keys* $k^{(A)} \in Expression$ *and for all* Abstract Environments $\left(env_C^{(A)}, env_V^{(A)}\right) \in Env^{(A)}$*.*

$$\begin{gathered}
\left\{containsKey(c,k,env) : env \in \gamma_{Must}\left(env^{(A)}\right)\right\} \\
\subseteq \\
\gamma_B\left(containsKey^{(Must)}\left(c^{(A)}, k^{(A)}, env^{(A)}\right)\right)
\end{gathered}$$

Proof. In the function $containsKey^{(Must)}$ we distinguish two cases:

- The result of $findElementsByKey^{(Must)}$ is the empty set ($|F| = 0$).
- The result of $findElementsByKey^{(Must)}$ is not the empty set ($|F| > 0$).

We are showing for both cases that the theorem holds.

Case 1: In the first case $containsKey^{(Must)}\left(c^{(A)}, k^{(A)}, env^{(A)}\right) = \top$. And Since $\gamma_B(\top) = \{true, false\}$, we can conclude

$$\left\{containsKey(c,k,env) : env \in \gamma_{Must}\left(env^{(A)}\right)\right\} \subseteq \gamma_B\left(containsKey^{(Must)}\left(c^{(A)}, k^{(A)}, env^{(A)}\right)\right)$$

Case 2: In the second case we assume, that $findElementsByKey^{(Must)}$ is not the empty set.

From Theorem 4 we know, that if $findElementsByKey^{(Must)}$ does not return an empty set then $findElementsByKey$ does not return an empty set in any environment $env \in \gamma_{May}\left(env^{(A)}\right)$. This means that

$$\left\{containsKey(c,k,env) : env \in \gamma_{Must}\left(env^{(A)}\right)\right\} = \{true\}$$

We furthermore know, that in this case $containsKey^{(Must)}\left(c^{(A)}, k^{(A)}, env^{(A)}\right) = true$ And because $\gamma_B(true) = \{true\}$ we can conclude for the second case

$$\left\{containsKey(c,k,env) : env \in \gamma_{Must}\left(env^{(A)}\right)\right\} \subseteq \gamma_B\left(containsKey^{(Must)}\left(c^{(A)}, k^{(A)}, env^{(A)}\right)\right)$$

.

Because we have proven the Theorem for both cases, we can deduce that $containsKey^{(Must)}$ is sound. □

5.4.3 Contains Key - May and Must Analysis

Remember that we combine the *May* and *Must Analysis* with the cartesian product. Usually the abstract semantics are applied for the two domains separately. In the case of contains key however, we can gain more precision if we combine the results of the two domains.

Because the combination of the *May* and the *Must Analysis* is defined as the cartesian product the input for the combined $containsKey$ operation are two *Abstract Environments* where one belongs to the *May Analysis* and the other to the *Must Analysis*.

If the *Must Analysis* can tell that the collection must contain an element with key k, then we return true. If the *May Analysis* says that there can be no element in the collection with key k, then we return false. Otherwise we can not tell whether the element is in the collection or not and therefore we return top.

$$\begin{array}{ll} \textbf{containsKey}^{(\mathbf{A})}: & (\mathbf{HId} \times \mathbf{Expression}) \to \left(\left(\mathbf{Env}^{(\mathbf{A})} \times \mathbf{Env}^{(\mathbf{A})}\right) \to \mathbf{B}^{(\mathbf{A})}\right) \\ containsKey^{(A)} & (c,k)(env^{(May)}, env^{(Must)}) = \\ & \begin{cases} false & \text{if } containsKey^{(May)}\left(c,k,env^{(May)}\right) = false \\ true & \text{if } containsKey^{(Must)}\left(c,k,env^{(Must)}\right) = true \\ \top & \text{otherwise} \end{cases} \end{array}$$

Soundness

We have already shown that the $containsKey^{(May)}$ and the $containsKey^{(Must)}$ operations are sound. In the first two cases of $containsKey^{(A)}$ we simply propagate their results. In the third case we return top which is always sound, since it over-approximates any set of boolean values that might be returned from the concrete $containsKey$ semantics. We can therefore conclude that $containsKey^{(A)}$ is sound.

5.5 Contains Value

The $containsValue$ operation checks whether a given value is contained in a collection or not.

The operations are defined analogously to the $containsKey$ operations.

$$\begin{array}{ll} \textbf{containsValue}: & (\mathbf{Ref} \times \mathbf{V}) \to ((\mathbf{Env_C}, \mathbf{Env_V}) \to \{\mathbf{true}, \mathbf{false}\}) \\ containsValue & (c,v)(env_C, env_V) = \begin{cases} true & \text{if } |F| > 0 \\ false & \text{otherwise} \end{cases} \\ & \text{where } F = findElementsByValue(c, v, env_C, env_V) \end{array}$$

$$\textbf{containsValue}^{(\textbf{May})}: \ (\textbf{HId} \times \textbf{Expression}) \to \left(\left(\textbf{Env}_{\textbf{C}}^{(\textbf{A})} \times \textbf{Env}_{\textbf{V}}^{(\textbf{A})}\right) \to \textbf{B}^{(\textbf{A})}\right)$$

$$containsValue^{(May)} \quad (c, v)(env_C, env_V) = \begin{cases} false & \text{if } |F| = 0 \\ \top & \text{otherwise} \end{cases}$$
$$\text{where } F = findElementsByValue^{(May)}(c, v, env_C, env_V)$$

$$\textbf{containsValue}^{(\textbf{Must})}: \ (\textbf{HId} \times \textbf{Expression}) \to \left(\left(\textbf{Env}_{\textbf{C}}^{(\textbf{A})} \times \textbf{Env}_{\textbf{V}}^{(\textbf{A})}\right) \to \textbf{B}^{(\textbf{A})}\right)$$

$$containsValue^{(Must)} \quad (c, v)(env_C, env_V) = \begin{cases} true & \text{if } |F| > 0 \\ \top & \text{otherwise} \end{cases}$$
$$\text{where } F = findElementsByValue^{(Must)}(c, v, env_C, env_V)$$

$$\textbf{containsValue}^{(\textbf{A})}: \ (\textbf{HId} \times \textbf{Expression}) \to \left(\left(\textbf{Env}^{(\textbf{A})} \times \textbf{Env}^{(\textbf{A})}\right) \to \textbf{B}^{(\textbf{A})}\right)$$

$$containsValue^{(A)} \quad (c, v)(env^{(May)}, env^{(Must)}) = \begin{cases} false & \text{if } containsValue^{(May)}\left(c, v, env^{(May)}\right) = false \\ true & \text{if } containsValue^{(Must)}\left(c, v, env^{(Must)}\right) = true \\ \top & \text{otherwise} \end{cases}$$

5.5.1 Soundness

The soundness of the *containsValue* operations can be shown analogously to the soundness of the *containsKey* operations.

5.6 Add

Add appends a value at the end of a list. As described earlier the key of an element in a list corresponds to its position in the list. The key of the first element in the list is 0. For the add operation this means that the key of the element equals the collection length before the add operation. In the script shown in listing 5.2 for example the key-value pair (0, 1) is added to a previously empty list.

Listing 5.2: Add element to list

```
1 $c := collections->create_number_collection();
2 $c->add(1);
```

The length of a collection in the concrete domain is equal to the number of elements in the *Element Set* of c. In the concrete semantics we therefore add a the given value v to the concrete collection c in the *Collection Environment* env_C at key $|env_C(c)|$.

$$\textbf{add}: \ (\textbf{Ref} \times \textbf{V}) \to ((\textbf{Env}_{\textbf{C}} \times \textbf{Env}_{\textbf{V}}) \to (\textbf{Env}_{\textbf{C}} \times \textbf{Env}_{\textbf{V}}))$$

$$add \quad (c, v)(env_C, env_V) = addElement(c, |env_C(c)|, v, env_C, env_V)$$

The abstract semantics of *add* for the *May* and the *Must Analysis* are defined identically. The key of the of the added abstract element corresponds to the abstracted collection length. The formalization of the collection length abstraction is out of the scope of this thesis. We therefore assume here that a sound abstraction of the collection length exists and that the length of a collection c is tracked in the *Abstract Value Environment* with the identifier $(c, \text{"}length\text{"})$. We can therefore simply add the new element at key $(c, \text{"}length\text{"})$

$$\textbf{add}^{(\textbf{A})}: \ (\textbf{HId} \times \textbf{Expression}) \to \left(\left(\textbf{Env}_{\textbf{C}}^{(\textbf{A})} \times \textbf{Env}_{\textbf{V}}^{(\textbf{A})}\right) \to \left(\textbf{Env}_{\textbf{C}}^{(\textbf{A})} \times \textbf{Env}_{\textbf{V}}^{(\textbf{A})}\right)\right)$$

$$add^{(A)} \quad (c, v)(env_C, env_V) = addElement^{(A)}(c, (c, \text{"}length\text{"}), v, env_C, env_V)$$

5.6.1 Soundness

Since we assume that the collection length is abstracted soundly and since we have already shown that $addElement^{(A)}$ is sound, we can conclude that $add^{(A)}$ is sound as well.

5.7 Set At

Set at adds an element to a map at a given key. If the key is already present in the map, then the existing value is replaced with the new value. Listing 5.3 shows an example where the value *ETHZ* is added to a map at key *Zurich*.

Listing 5.3: Set value at key in a map

```
1 $c := collections->create_string_map();
2 $c->add("Zurich", "ETHZ");
```

We can express this operation using basic operations that we have already defined: In the concrete as well as in the abstract semantics we first remove a potentially present element at the given key and then add the given key-value pair to the collection.

$$
\begin{array}{ll}
\textbf{setAt}: & (\textbf{Ref} \times \textbf{V} \times \textbf{V}) \to (\textbf{Env} \to \textbf{Env}) \\
setAt & (c,k,v)(env) = env^{(2)}: \\
 & env^{(1)} = removeElement(c, key, env) \wedge \\
 & env^{(2)} = addElement(c,k,v,env^{(1)})
\end{array}
$$

$$
\begin{array}{ll}
\textbf{setAt}^{(\textbf{A})}: & (\textbf{HId} \times \textbf{Expression} \times \textbf{Expression}) \to \left(\textbf{Env}^{(\textbf{A})} \to \textbf{Env}^{(\textbf{A})}\right) \\
setAt^{(A)} & (c,k,v)(env) = env^{(2)}: \\
 & env^{(1)} = removeElement^{(A)}(c,k,env) \wedge \\
 & env^{(2)} = addElement^{(A)}(c,k,v,env^{(1)})
\end{array}
$$

5.7.1 Soundness

Since we only concatenate operations which we have already proven sound, we can conclude that $setAt^{(A)}$ is sound as well.

5.8 Remove At (List)

If *remove at* is called on a list, the element at the given position is removed from the collection. If the provided index is out of bounds, the list remains unchanged. Listing 5.4 shows an example where the first element of a collection is removed. Since the key in a list represents the position of the element in the list, the key of all the elements that have a greater key than the removed element's key need to be decreased by one.

Listing 5.4: Remove element at key in a list

```
1 $messages->remove_at(0);
```

For both, the concrete and the abstract semantics we can build this operation based on basic operations which we have already defined: First we check whether an element with the given key exists in the collection by using the *containsKey* operation. If this is not the case we simply return the unchanged environment. Otherwise we remove the element with key k from the

collection c and then we decrease the keys of all elements in c that have a greater key than k by one. We can do this by concatenating the *removeElement* and the *decreaseKeys* operations.

$$\begin{array}{ll}
\textbf{removeAt}: & (\textbf{Ref} \times \textbf{V}) \to ((\textbf{Env}_\textbf{C} \times \textbf{Env}_\textbf{V}) \to (\textbf{Env}_\textbf{C} \times \textbf{Env}_\textbf{V})) \\
\mathit{removeAt} & (c,k)(env_C, env_V) = \begin{cases} (env_C^{(1)}, env_V^{(2)}) & \text{if } \mathit{containsKey}(c,k,env_C,env_V) \\ (env_C, env_V) & \text{otherwise} \end{cases} \\
& (env_C^{(1)}, env_V^{(1)}) = \mathit{removeElement}(c,k,env_C,env_V) \wedge \\
& env_V^{(2)} = \mathit{decreaseKeys}(c,k,env_C^{(1)}, env_V^{(1)})
\end{array}$$

$$\begin{array}{ll}
\textbf{removeAt}^{(\textbf{A})}: & (\textbf{HId} \times \textbf{Expression}) \to \left(\left(\textbf{Env}_\textbf{C}^{(\textbf{A})} \times \textbf{Env}_\textbf{V}^{(\textbf{A})}\right) \to \left(\textbf{Env}_\textbf{C}^{(\textbf{A})} \times \textbf{Env}_\textbf{V}^{(\textbf{A})}\right)\right) \\
\mathit{removeAt}^{(A)} & (c,k)(env_C, env_V) = \\
& \begin{cases} (env_C^{(1)}, env_V^{(2} & \text{if } \mathit{containsKey}^{(A)}(c,k,env_C,env_V) = \{true\} \\ (env_C, env_V) & \text{if } \mathit{containsKey}^{(A)}(c,k,env_C,env_V) = \{false\} \\ \begin{pmatrix} env_C \sqcup_C env_C^{(1)}, \\ env_V \sqcup_V env_V^{(2)} \end{pmatrix} & \text{otherwise} \end{cases} \\
& (env_C^{(1)}, env_V^{(1)}) = \mathit{removeElement}^{(A)}(c,k,env_C,env_V) \wedge \\
& env_V^{(2)} = \mathit{decreaseKeys}^{(A)}(c,k,env_C^{(1)}, env_V^{(1)})
\end{array}$$

5.8.1 Soundness

To show that $\mathit{removeAt}^{(A)}$ is sound, we look at the three different cases that the $\mathit{removeAt}^{(A)}$ operation distinguishes and show for each case that it is sound.

- In the first case we know that $\mathit{containsKey}^{(A)}(c,k,env_C,env_V) = \{true\}$. Because we already know that the $\mathit{containsKey}^{(A)}$ operation is sound, we know that in the concrete semantics $\mathit{containsKey}(c,k,env_C,env_V) = true$ in all *Collection Environments*. This means that in the concrete and in the abstract the same concatenation of operations is returned. For each of the concatenated operations, we know that it is sound. We can therefore conclude for the first case that $\mathit{removeAt}^{(A)}$ is sound .
- In the second case we know that $\mathit{containsKey}^{(A)}(c,k,env_C,env_V) = \{false\}$. Because $\mathit{containsKey}^{(A)}$ is sound, we also know that in the concrete semantics $\mathit{containsKey}(c,k,env_C,env_V) = false$ in all *Collection Environments*. This means that in both, the concrete and the abstract semantics the unchanged state is returned. We can therefore conclude that $\mathit{removeAt}^{(A)}$ is sound for the second case as well.
- In the third case we know that $\mathit{containsKey}^{(A)}(c,k,env_C,env_V) = \{true, false\}$. We therefore can not say whether the concrete *removeAt* operation returns the changed or the unchanged environment. But in this case, the abstract $\mathit{removeAt}^{(A)}$ operation returns the least upper bound of the changed and the unchanged environment. Therefore the third case is sound as well.

We have shown for all three cases of the $\mathit{removeAt}^{(A)}$ operation that they are sound. We can therefore conclude that $\mathit{removeAt}^{(A)}$ is sound.

5.9 Remove At (Map)

If *remove at* is called on a map, the element at the given key is removed from the collection. If the key does not exist in the map, the map remains unchanged. Listing 5.5 shows an example where an element at key *Zurich* is removed from a string map.

Listing 5.5: Remove element at key in a map

```
1 $universities->remove_at("Zurich");
```

For both, the concrete and the abstract semantics we can build this operation based on operations that we have already defined.

First we check whether an element with the given key exists in the collection by using the *containsKey* operation. If this is not the case we simply return the unchanged environment. Otherwise we remove the element with key k from the collection c using the *removeElement* operation.

$$\begin{aligned}
&\textbf{removeAt}: && (\textbf{Ref} \times \textbf{V}) \to ((\textbf{Env}_\textbf{C} \times \textbf{Env}_\textbf{V}) \to (\textbf{Env}_\textbf{C} \times \textbf{Env}_\textbf{V})) \\
&\mathit{removeAt} && (c,k)(env_C, env_V) = \begin{cases} (env_C^{(1)}, env_V^{(1)}) & \text{if } \mathit{containsKey}(c,k,env_C,env_V) \\ (env_C, env_V) & \text{otherwise} \end{cases} \\
& && (env_C^{(1)}, env_V^{(1)}) = \mathit{removeElement}(c,k,env_C,env_V)
\end{aligned}$$

$$\begin{aligned}
&\textbf{removeAt}^{(\textbf{A})}: && (\textbf{HId} \times \textbf{Expression}) \to \left(\left(\textbf{Env}_\textbf{C}^{(\textbf{A})} \times \textbf{Env}_\textbf{V}^{(\textbf{A})}\right) \to \left(\textbf{Env}_\textbf{C}^{(\textbf{A})} \times \textbf{Env}_\textbf{V}^{(\textbf{A})}\right)\right) \\
&\mathit{removeAt}^{(A)} && (c,k)(env_C, env_V) = \\
& && \begin{cases} (env_C^{(1)}, env_V^{(1)}) & \text{if } \mathit{containsKey}^{(A)}(c,k,env_C,env_V) = \{true\} \\ (env_C, env_V) & \text{if } \mathit{containsKey}^{(A)}(c,k,env_C,env_V) = \{false\} \\ \begin{pmatrix} env_C \sqcup_C env_C^{(1)}, \\ env_V \sqcup_V env_V^{(1)} \end{pmatrix} & \text{otherwise} \end{cases} \\
& && (env_C^{(1)}, env_V^{(1)}) = \mathit{removeElement}^{(A)}(c,k,env_C,env_V)
\end{aligned}$$

5.9.1 Soundness

The soundness of $\mathit{removeAt}^{(A)}$ for maps can be shown analogously to the soundness of $\mathit{removeAt}^{(A)}$ for lists.

5.10 At

The *at* function returns the value that is stored in a collection at a given key. If the key is not present in the collection, *invalid* is returned.

In the concrete semantics of *at* we use the *containsKey* function to determine whether an element with key k is contained in a collection c or not. If the *containsKey* function returns false, then the concrete semantics of *at* returns $\{invalid\}$. Otherwise a set containing the value field of the *Element* with key k is returned (this *Element* can be found using the $findElementsByKey$ operation).

$$\begin{aligned}
&\textbf{at}: && (\textbf{Ref} \times \textbf{Expression}) \to ((\textbf{Env}_\textbf{C} \times \textbf{Env}_\textbf{V}) \to (\mathcal{P}(\textbf{Elem} \times \{\text{"}\textbf{value}\text{"}\}) \cup \{\textbf{invalid}\})) \\
&\mathit{at} && (c,k)(env_C, env_V) = \begin{cases} V & \text{if } \mathit{containsKey}(c,k,env_C,env_V) \\ \{invalid\} & \text{otherwise} \end{cases} \\
& && \text{where } V = \{(r, \text{"}value\text{"}) : r \in findElementsByKey(c,k,env_C,env_V)\}
\end{aligned}$$

In the abstract semantics we use the $\mathit{containsKey}^{(A)}$ function to determine whether a key k is contained in the collection or not.

- If k is certainly contained in the abstract collection ($\mathit{containsKey}^{(A)}$ returns $\{true\}$), then the value fields of all the *Tuple Identifiers* whose key field might be equal to k is returned. These *Tuple Identifiers* can be found using the operation $findElementsByKey^{(May)}$.

- If k is certainly not contained in the abstract collection ($containsKey^{(A)}$ returns $\{false\}$), $\{invalid\}$ is returned.
- If we don't know whether k is contained in the collection ($containsKey^{(A)}$ returns $\{true, false\}$), the least upper bound of the two other results (value fields of *Tuple Identifiers* and $\{invalid\}$) is returned.

$$\mathbf{at^{(A)}}: \quad (\mathbf{HId} \times \mathbf{Expression}) \to \left(\left(\mathbf{Env_C^{(A)}} \times \mathbf{Env_V^{(A)}}\right) \to (\mathcal{P}(\mathbf{TId} \times \{\text{"}\mathbf{value}\text{"}\}) \cup \{\mathbf{invalid}\})\right)$$

$$at^{(A)} \quad (c,k)(env_C, env_V) = \begin{cases} V & \text{if } containsKey^{(A)}(c,k,env_C,env_V) = \{true\} \\ \{invalid\} & \text{if } containsKey^{(A)}(c,k,env_C,env_V) = \{false\} \\ V \cup \{invalid\} & \text{otherwise} \end{cases}$$

where $V = \{(t, \text{"}value\text{"}) : t \in findElementsByKey^{(May)}(c,k,env_C,env_V)\}$

5.10.1 Soundness

To show that $at^{(A)}$ is sound we look at the three possible results of the operation separately. For each case we argue that the result of $at^{(A)}$ over-approximates the result of at.

- In the first case $containsKey^{(A)}$ returns $\{true\}$. Since we have shown that $containsKey^{(A)}$ over-approximates $containsKey$ we know that $containsKey$ in the concrete at function must return *true* in all concrete *Collection Environments*. This means that the concrete semantics returns the result of $findElementsByKey$ over all concrete *Collection Environments* and the abstract semantics returns the result of $findElementsByKey^{(May)}$. In Theorem 3 we have shown that $findElementsByKey^{(May)}$ over-approximates $findElementsByKey$. Therefore, this is case sound.
- In the second case $containsKey^{(A)}$ returns $\{false\}$. For the same reason as in the first case, the $containsKey$ operation must return *false*. This means that both the abstract and the concrete semantics return $\{invalid\}$. Therefore, the second case is sound as well.
- In the third case $containsKey^{(A)}$ returns $\{true, false\}$. This means that in the concrete semantics, $containsKey$ can either return $\{invalid\}$ or the result of $findElementsByKey$. But since the abstract semantic returns the least upper bound of $\{invalid\}$ and the result of $findElementsByKey^{(Must)}$, this case is also sound.

We have argued for all three cases that the output of $at^{(A)}$ over-approximates the output of at. And therefore we can conclude, that $at^{(A)}$ is sound.

Chapter 6

Implementation

The analyses were implemented as an extension to the existing analysis infrastructure of Sample. They use the already available mechanisms to run an analysis based on a least fixed point computation. This chapter describes how we integrated the analyses in Sample.

6.1 Implementation of Analyses

Since Sample already provides an implementation for the value domain, the implementation only needs to cover the *Abstract Collection Environment*. We extend the already existing non-relational Heap Domain in Sample, such that it is also able to represent the *Abstract Collection Environment*. We have implemented an individual Heap Domain for both the *May* and the *Must Analysis*. Additionally we also implemented a Heap Domain that combines the *May* and the *Must Heap Domain* as a cartesian product.

These Heap Domains offer basic operations such as adding a *Tuple Identifier* to a collection. All those operations are not specific to TouchDevelop and could also be used to implement collection analyses for other programming languages.

Figure 6.1 shows the structure of the domains for an analysis that uses the *May Analysis* as well as the *Must Analysis*.

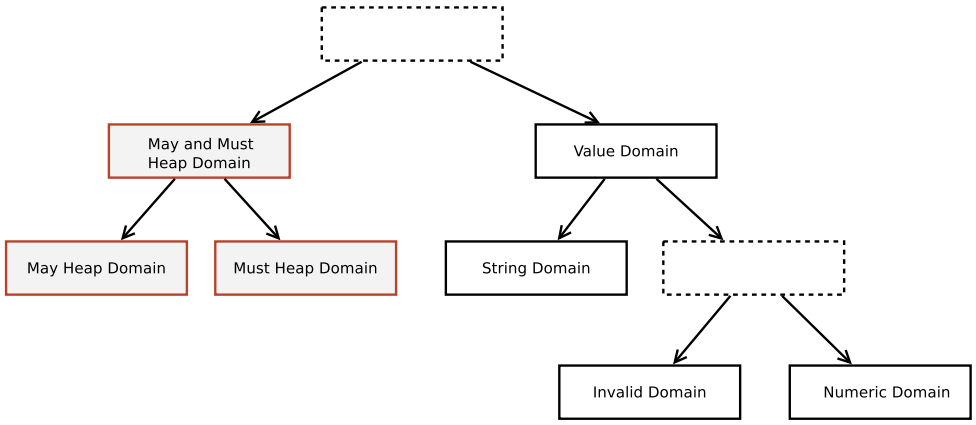

Figure 6.1: Structure of the domains with the new domains for the May and the Must Analysis

To run an analysis we combine one of the three Heap Domains with the Value Domain and pass them as an argument to Samples least fixpoint computation. Depending upon which Heap Domain is passed the *May*, the *Must* or the *May and Must Analysis* is executed.

6.2 Implementation of Semantics

The abstract semantics is the part of the implementation that is specific to TouchDevelop. They operate on a state (a heap and a value domain) and use the basic operations that are defined by these domains.

To implement the abstract semantics, we grouped the collections based on the operations that they offer and defined abstract classes that implement common semantic operations. We distinguish between linear collections (sets and lists) and maps. The main difference is that the first are accessed with a linear numeric key and the latter can be accessed with any type of key. We further defined an abstract class for mutable collections, which are linear collections that also offer update operations.

The specific collection types (e.g. StringMap) inherit their semantic operations from these abstract classes. The class structure with a few examples of specific collection types is depicted in figure 6.2.

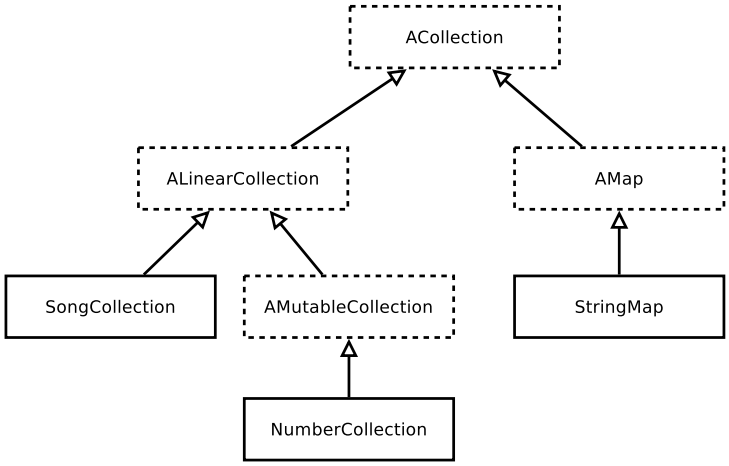

Figure 6.2: Class Structure for the collection types

6.3 Abstraction of Collection Length

In Sample there already exists an abstraction of the collection length. This abstraction is implemented using the same value domain that is used to abstract values of collection elements. For each abstract collection the abstracted collection length is tracked with an individual identifier. We implemented the abstract semantics of our analysis such that they update this identifier when a collection is altered.

We also used the abstraction of the collection length to make the abstract semantics more precise. When for example a list or a set is accessed we need to be able to determine whether the provided index is in the bounds. Since lists and sets have a linear index that starts at 0, we know that if the provided index is between 0 and the collection length minus 1, then there must be an element at that index. We can use that to determine whether an access to a list or a set can return an invalid object or not. We will see in Chapter 7 that this helps us to improve the precision in many cases.

Chapter 7

Evaluation

The evaluation presented in this chapter shall show the benefits and the limitations of the implemented technique. We compare four different analyses:

- *Smashing Analysis:* The Smashing Analysis is the analysis that was used before this thesis to abstract collection elements in Sample. It abstracts all elements of a collection with one summary node. Although this analysis already existed in Sample before, we adapted it such that it can handle non linear keys (like they are used in maps) as well. This means instead of having just one summary node for all the collection values, there are now two summary nodes per collection. One represents all the keys and the other one all the values of a collection. We use this analysis as a baseline to show how the newly added analyses compare to it in terms of precision and performance
- *May Analysis:* This is the analysis that we described in this thesis as the May Analysis. In the implementation of the analysis we summarize collection elements based on the program point where they were added. We therefore are able to distinguish collection elements that are added at different program points. The May Analysis tracks which elements may be in a collection. Hence, it can be used to prove that an element is certainly not contained in a collection.
- *May And Must Analysis (standard lub):* The third analysis combines the May and the Must Analysis as described in this thesis. Additionally to the properties of the May Analysis, this analysis can be used to show that an element certainly is contained in a collection. In this analysis we use the standard least upper bound operator for the *Must Analysis*.
- *May And Must Analysis (extended lub):* This analysis is the same as the third analysis except that we use the extended least upper bound operator for the *Must Analysis* as described in Chapter 4.

We split the evaluation of our analysis into three parts. First we run the analyses on a set of hand constructed TouchDevelop scripts. To present how the analysis works on real world scripts we show the results on a few real world scripts from the TouchDevelop cloud. To finally see the implications of our analyses on a large number of TouchDevelop scripts we run the analysis on over 5000 scripts from the TouchDevelop cloud and compared them in terms of precision and performance.

7.1 Case Studies

We are first going to show how the analyses compare to each other on a set of hand constructed TouchDevelop scripts. The scripts are all fully functional but they do not have a meaningful purpose. We constructed these scripts such that they are minimal but still illustrate how the analyses work.

7.1.1 Key Access

The script shown in listing 7.1 constructs a *Number Map* with two elements, where one of them is 0. Then the non-zero element is accessed and stored in the variable x. One can easily see that x can never be 0 and therefore the division on line 7 is always safe.

Listing 7.1: Key access

```
1 action main() {
2     $c := collections->create_number_map();
3     $c->set_at(100, 2);
4     $c->set_at(200, 0);
5
6     $x := $c->at(100);
7     $y := 4 / $x;
8 }
```

This script highlights the difference between the *Smashing Analysis* the *May Analysis* and the *Must Analysis*.

Remember that in the *Smashing Analysis* all elements are abstracted with one summary node. Hence, we can not distinguish between the two elements in the map. With this analysis we are therefore only able to prove that x is between 0 and 2. Notice that even a more precise *Numerical Domain* would at most allow us to prove that x is either 0 or 2. Because we can not prove that $x \neq 0$, we can also not prove that the division at line 7 is safe. The *Smashing Analysis* therefore raises a false alarm, that at line 7 there might be a division by zero.

Let's now look at the *May Analysis*. The *May Analysis* is able to distinguish between the two elements in the collection. Therefore it would be able to prove that $x \neq 0$. But the *May Analysis* only tracks whether an element might be in a collection. Remember that the semantics of the *Number Map's at* operation returns 0 if a key, that is not in the map, is accessed. Since the *May Analysis* can not say that a key certainly is in the map, it must also consider the possibility that 0 is returned from any collection access. Therefore we are again only able to prove that x is either 0 or 2 and the *May Analysis* raises a false alarm as well.

The *May and Must Analysis* finally can not only distinguish the two elements but can also prove that at key 100 there must be an element. It therefore has not to consider the possibility that the collection access returns 0. Hence, it can prove that $x = 2$ and therefore also that the division at line 7 is safe. No false alarm is raised.

7.1.2 Elements Added in Two Branches

The script in listing 7.2 shows a conditional where in both branches an element is added to the collection at key *Switzerland*. Then the value stored at *Switzerland* is retrieved and posted to the wall.

Listing 7.2: Add an element to a map in two branches

```
1  action main() {
2      $s := collections->create_string_map();
3
4      if (wall->ask_boolean("Choose", "Zurich, Bern")) then {
5          $s->set_at("Switzerland", "Zurich");
6      }
7      else {
8          $s->set_at("Switzerland", "Bern");
9      }
10
11     $c := $s->at("Switzerland");
12
13     $c->post_to_wall();
14 }
```

With this example we want to highlight the effect of the extended least upper bound operator. The property that we are interested in is whether the variable c can be invalid. We can easily see that this can never be the case, since there is always a value stored at key *Switzerland* and therefore the collection access at line 11 never returns an invalid object.

However with the *Smashing* and the *May Analysis* we are not able to tell whether the key *Switzerland* is contained in the list since they do not track this information. They both raise a false alarm at line 13 that a method is called on a possibly invalid object.

We need the *May and Must Analysis* to prove this property. However if we use the standard least upper bound operator for the *Must Analysis* we loose the information that an element must be in the key collection when we join the two abstract collections after the *then* and *else* branch. Why this is the case is explained in section 4.8.5. Therefore the *May and Must Analysis* with the standard least upper bound operator also raises a false alarm at line 13.

As soon as we activate the extended join operation for the *Must Analysis* we are able to show that there must be an element in the collection with key *Switzerland* and we therefore can prove that c is never invalid. The *May and Must Analysis* with the extended join operation therefore does not raise a false alarm for this script.

One might assume that we get the same results for a script as shown in listing 7.3 where in both branches of a conditional an element is added to a list.

Listing 7.3: Add an element to a list in in two branches

```
1  action main() {
2      $s := collections->create_string_collection();
3
4      if (wall->ask_boolean("Choose", "Zurich, Bern")) then {
5          $s->add("Zurich");
6      }
7      else {
8          $s->add("Bern");
9      }
10
11     $c := $s->at(0);
12     $c->post_to_wall();
13 }
```

On first sight this seems reasonable, since both elements are added at key 0 which makes this example very similar to the previous one. However this example will show that the extended join operation does not have as big of an impact on the precision as expected.

In this example it is important to notice that we are using a collection that has a linear key. This means, that if for example we know that the size of a collection is at least 2 there must exist elements at index 0 and 1. We have described earlier that the abstraction of the collection length already exists in Sample. In this example that abstraction gives us the information, that after the two branches of the conditional are joined, the length of the list is equal to 1. The semantics of the at operation for collections with linear keys checks whether the provided index certainly is between 0 and the collection length minus 1. In our case we know that the collection length is 1 and hence the collection access at index 0 can never return an invalid object. With this additional information of the collection length we are able to prove that the variable c is never invalid and therefore all three analyses do not raise a false alarm.

7.1.3 Relations between Variables and Collection Elements

In the script shown in listing 7.4 we add a number that we received as an argument to a collection. Since we received the number as an argument, we know nothing about the value of it. We then check whether the parameter is contained in the list. Only if we can prove that the parameter is certainly in the list we are also able to prove that k is equal to 1 at line 10 and therefore no division by zero can happen.

Listing 7.4: Add a parameter to collection

```
1  action create(x: Number) {
2      $c := collections->create_number_collection();
3      $c->add($x);
4
5      $k := 0;
6      if ($c->contains($x)) then {
7          $k := 1;
8      }
9
10     $x := 10 / $k;
11 }
```

We use this example to show that our analysis can benefit from a relational value domain. Sample provides relational numerical domains such as Octagons [15] by using the APRON library [12]. This means that it can track relations between different identifiers. In our example it is able to track that the collection element at position 0 is equal to x.

The *May* and the *Smashing Analysis* both raise a false alarm in this example, because they can not track whether an element is certainly contained in a collection.

But the *Must Analysis* combined with the relational value domain can capture that an element which has the same value as x is certainly contained in the collection. Therefore we can prove, that the condition $\$c \rightarrow \mathit{contains}(\$x)$ always evaluates to true and that k is always 1 at line 10. Hence, he *Must Analysis* does not raise a false alarm in this example.

This shows that our analysis can use the benefits of relational value domains and that the implemented technique can become more precise if the used value domain is more precise. If we would use a non-relational numerical value domain which is not able to represent that the collection entry at key 0 is equal to x, it would have been impossible to prove the desired property.

7.1.4 Contains Key

Listing 7.5 shows a script where a *String Map* is passed as an argument to an action. We don't know anything about the content of that map. But since we check whether there is an element at key *Zurich* before we access it, we can be sure that this collection access never returns an invalid object.

Listing 7.5: Contains key

```
1 action main(c: String_Map) {
2     if ($c->keys()->contains("Zurich")) then {
3         $c->at("Zurich")->post_to_wall();
4     }
5 }
```

With this script we want to show how the *assume* function works in our analysis. When the *then* branch of the conditional at line 2 is entered, we can assume that an element at key *Zurich* is contained in the collection. Therefore a new collection element with key *Zurich* is added to the abstract collection. If we then access the element inside the *then* branch, the *Must Analysis* is able to prove that there must be an element at key *Zurich*. Therefore it can be shown that the collection access at line 3 never returns an invalid object. Hence, the *Must Analysis* does not raise a false alarm.

The *May Analysis* and the *Smashing Analysis* do not track the information whether an element must be in a collection or not. Hence, they can not prove that the collection access at line 3 never returns an invalid object. Both analyses raise a false alarm because a method on a possibly invalid object is called.

7.1.5 Remove

The script shown in listing 7.6 takes a *String Map* as a parameter and removes the element at key *Zurich*. It then checks whether the key is contained in the collection and if this is the case, it performs a division by zero. However, one can observe, that the condition at line 3 does never evaluate to true, because the key has been removed and therefore the division by zero is never executed.

Listing 7.6: Remove an element from a collection

```
1 action main(c: String_Map) {
2     $c->remove("Zurich");
3     if($c->keys()->contains("Zurich")) then {
4         $x := 5 / 0;
5     }
6 }
```

With this example we want to show, how our analysis depends on the value domain. When the *remove* operation is called at line 2, our analysis assumes for each key in the collection, that it does not equal *Zurich*. But the value domain for strings that is used in Sample can not represent the fact that an identifier does not equal a certain string. Therefore this assume operation has no effect and all three analyses raise a false alarm.

If the string domain would be able to represent that an identifier does not equal a certain string, we could prove that a division by zero will never occur.

7.1.6 Object List

The script shown in listing 7.7 creates a Message with the text *Hello world!* and adds it to an empty collection. We then check whether the message at position 0 in the list has the text *Hello world!*. Only if this is the case a division by zero is avoided. But we can observe that the condition at line 7 will always evaluate to true and therefore a division by zero can never occur.

Listing 7.7: Message list

```
1  action main() {
2      $msgs := collections->create_message_collection();
3      $msg := social->create_message("Hello world!");
4      $msgs->add($msg);
5
6      $k := 0;
7      if ($msgs->at(0)->message()->equals("Hello world!")) then {
8          $k := 2;
9      }
10
11     $x := 4 / $k;
12 }
```

With this example we want to show that our analysis can handle objects as well as primitive types.

The *Smashing Analysis*, the *May Analysis* and the *Must Analysis* are all able to prove, that no division by zero occurs and do not raise a false alarm. Notice, that the *Smashing Analysis* and the *May Analysis* benefit from the collection length abstraction in order to determine that there must be an element in the collection at key 0. Without the collection length abstraction, only the *Must Analysis* would be able to prove desired property.

7.1.7 Adding Elements with a Loop

The script shown in listing 7.8 adds 10 elements to a Number Map using a loop. It then divides 4 by the value of the element stored at key 1 in the Number Map. Remember that the *at* method of the *Number Map* returns 0 if the provided key does not exist. In this example would lead to a division by zero. We can however see, that at key 1 there must be the value 2 and therefore a division by zero can never happen.

Listing 7.8: Add elements to a map using a loop

```
1  action main() {
2      $c := collections->create_number_map();
3
4      for 0 <= i < 10 do {
5          $c->set_at($i, $i+1);
6      }
7
8      $x := 4 / $c->at(1);
9  }
```

With this example we want to show, how the analysis behaves if multiple elements are added to a collection in a loop. Our analysis abstracts all *Elements* that were added at the same program

point with one *Tuple Identifier*. This means that all *Elements* that are added to the collection at line 6 are abstracted by the same *Tuple Identifier*.

In this example the *Smashing Analysis* and the *May Analysis* will both have one element in the abstract collection after the loop. This abstract collection element summarizes all elements added inside the loop.

The *Must Analysis* on the other hand will have no elements in the abstract collection, since it is not sound to add summary elements to an abstract collection in the *Must Analysis*. This means, that we can not track that the elements added inside the loop must be in the collection. Hence, we are also not able to prove, that the collection access at line 8 never returns 0 and that the division at line 8 is never a division by 0.

All three analyses therefore raise a false alarm in this example.

One easy way to improve the precision for such cases would be to perform a loop unrolling. Then the elements would be added at individual program points and the *Must Analysis* could track them.

Let's now look at a variation of this example, that uses a list instead of a map. The script shown in listing 7.9 adds 10 elements to a number collection. We then access the first element. If we access a list at an index that does not exist, an invalid object is returned. However, we can see that in this example the collection access at line 8 never returns an invalid object. We can therefore safely use the retrieved number and post it to the wall.

Listing 7.9: Add elements to a list using a loop

```
1 action main() {
2     $c := collections->create_number_collection();
3
4     for 0 <= i < 10 do {
5         $c->add($i);
6     }
7
8     $c->at(1)->post_to_wall();
9 }
```

In contrast to maps, lists have linear keys. We can therefore use the information gained from the collection length abstraction. In our example we know that the collection length at line 8 must be 10. Hence, we can prove that the collection access at index 1 never returns an invalid object.

Therefore all three analyses do not raise a false alarm for this example.

7.2 Real World Scripts

In this Section we want to demonstrate how the *Must Analysis* can improve the precision of real world scripts retrieved from the TouchDevelop cloud.

To make the scripts easier to read we only present those parts of the source code that is relevant for the example. Each script has an identifier that is printed in the title of each example. With this identifier the complete scripts can be found in the TouchDevelop cloud using the following URL pattern: `https://www.touchdevelop.com/api/<identifier>/text`.

7.2.1 Random Wisdom - csnh

Random Wisdom is a simple script from the TouchDevelop cloud. The user can click a button to display a random wisdom on it's screen. The random wisdom is retrieved as a JSON Object from a web service. A JSON Object is not a typical collection but since its fields can be accessed using their names as key we can represent it as a map from Strings to JSON Objects.

Listing 7.10: Random Wisdom (csnh) script from TouchDevelop cloud

```
1  action main() {
2    wall->add_button("sync", "New Wisdom");
3    code->wisdom;
4  }
5
6  action wisdom() {
7    $response := web->create_request("http://...")->send;
8    $js := $response->content_as_json->field("quote")->to_string;
9    wall->clear;
10   $js->post_to_wall;
11   meta private;
12 }
13
14 event shake() {
15   code->wisdom;
16 }
17
18 event tap_wall_Page_Button(item: Page_Button) {
19   code->wisdom;
20 }
```

Once the script has retrieved the JSON Object, it directly accesses the *quote* field, in which the text of the random wisdom is supposed to be stored. This is not safe, since the web service could change its API and rename the field in which the random wisdom is stored. If this happens, the script crashes. This is a very common error in TouchDevelop scripts. In fact we could not find any script in which this corner case is handled correctly.

In this case all analyses correctly raise an alarm.

Corrected Version - vrhzzzsb

So far the alarm could correctly be detected. However to achieve this the *Must Analysis* would not have been needed. It gets interesting as soon as the developer corrects his script based on the alarm raised by the analysis.

To simulate this, we published a version of the script where the described corner case is handled correctly by checking whether the field *quote* actually exists in the JSON response before accessing it. The source code of the corrected *wisdom* action is shown in listing 7.11.

Listing 7.11: Corrected Random Wisdom (vrhzzzsb) script from the TouchDevelop cloud

```
1  action wisdom() {
2    $response := web->create_request("http://...")->send;
3    $js := $response->content_as_json;
4    wall->clear;
5    if $js->keys->contains("quote") then {
6      $js->field("quote")->post_to_wall;
7    }
8    else {
9      "No wisdom found" ->post_to_wall;
10   }
11   meta private;
12 }
```

With this corrected version of the script the *May Analysis* still shows an alarm. However this alarm is now a false alarm. The *May Analysis* does not have enough precision to prove that the collection access is safe, since it is never able to determine whether a key certainly is present in a collection or not.

The *Must Analysis* however can determine that inside the *then* branch at line 6 in the source code, there must be an element in the collection with the key *quote*. And therefore it is able to prove that the access to the key *quote* is safe and does not raise a false alarm.

We have found out that in most TouchDevelop scripts JSON Objects are not handled correctly. Therefore most alarms that are currently raised for scripts containing JSON Objects are not false alarms. However, if the developers correct their programs, we would not be able to remove those alarms (which then would be false alarms) without the *Must Analysis*.

7.2.2 Accent Colors - mbly

This TouchDevelop script presents the user with a palette of colors. The user can pick a color to show the ARGB (Alpha, Red, Green, Blue) values of the picked color.

Listing 7.12: Accent colors (mbly) script from the TouchDevelop cloud

```
1  // ... //
2
3  action list_colors() returns(list: String_Map) {
4    $list := collections->create_string_map;
5
6    $list->set_at("Lime", "ffA4c400");
7    $list->set_at("Lime 7", "Ffa2c139");
8    // ... //
9    $list->set_at("Mauve", "Ff76608a");
10   $list->set_at("Sienna", "Ff7a3b3f");
11   meta private;
12 }
13
14
```

```
15 action show() {
16   $cols := code->list_colors();
17   foreach s in $cols where true do {
18     // ... //
19     $tb->set_background($cols->at($s)->to_color);
20     // ... //
21  }
22 }
23
24 // ... //
```

In this example the script first creates a string map containing mappings from color names to their ARGB values. It then iterates over that map of strings using a *foreach* loop. The loop variable, in this example *s*, contains in each iteration a different key of the map. At line 19 the map is accessed at this key *s*. Obviously, this access will never return an invalid object.

However, the *May Analysis* is not able to prove this property, because it can not determine if a key must be present in a map. Hence, it raises a false alarm. In contrast to that, the *Must Analysis* is able to capture that there must be an element with the given key in the map and it does not raise a false alarm.

7.2.3 JSpaceTV Library - xpbx

This TouchDevelop script is a library that allows to remotely control a TV. It uses a global String Map to store properties. This is a pattern that is commonly used in TouchDevelop scripts.

Listing 7.13 shows the parts of the scripts that store and access the properties.

Listing 7.13: JSpaceTV library (xpbx) script from the TouchDevelop cloud

```
1  // ... //
2
3  var audio:String_Map {}
4
5  // ... ///
6
7  private action init_path_globals () returns  {
8    // ... //
9    (data->audio()) := (collections->create_string_map());
10   data->audio()->set_at("","/1/audio/volume");
11   data->audio()->set_at("audio","/1/audio/volume");
12   data->audio()->set_at("volume","/1/audio/volume");
13   // ... //
14 }
15
16 // ... //
17
18 action volume_get_from_TV() returns(json: Json_Object) {
19   $path := data->audio->at("audio");
20   $json := code->jointSpaceRequest_get($path);
21   meta private;
22 }
```

We can see that only the *Must Analysis* is able to capture that the key *audio* must be contained in the String Map $data \rightarrow audio$ and that the collection access at line 20 therefore never returns an invalid object. Hence, only the *Must Analysis* is able to avoid a false alarm in this case.

However, there currently exists a problem in Sample that by mistake detects all global variables as summary nodes. In our example this means that the collection $data \rightarrow audio$ is considered a summary collection which prevents us from adding elements to the collection in the *Must Analysis* and hence also from avoiding he false alarm. To be able to prove the desired property in this script we need to tell Sample manually that $data \rightarrow audio$ is not a summary collection.

7.3 Experiments

To further evaluate the precision and the performance of the implemented technique we run the analysis on 5107 real world scripts obtained from the TouchDevelop cloud. These are all the root scripts in the TouchDevelop cloud which were created before May 23. 2013 and were reported as not having compilation errors. We are interested in the precision and in the performance of the analysis.

All the tests were executed on the following environment: Intel Core 2 Quad CPU Q9550 @ 2.83 GHz, 4 GB RAM, Ubuntu 12.04, Java SE Runtime Environment 1.7.0.

7.3.1 Precision

For the precision we only considered the scripts on which Sample did not raise an error. In total these were 4422 scripts. On the other scripts either the compilation of the script was erroneous, the Java Runtime Environment crashed, the analysis threw an exception or could not be finished within 60 seconds. Out of these 4422 scripts 1909 use collections (including JSON Objects) and could therefore potentially benefit from the *Must Analysis*.

We analyzed all the scripts with each of the four analyses and measured the total number of warnings produced. This gives us an indication about the precision of the analyses, since a more precise analysis does raise less false alarms. The total number of warnings raised by each of the analyses for the 4422 scripts are denoted in table 7.1 and visualized in the diagram in figure 7.1

Table 7.1: Number of warnings measured on 4422 scripts

Analysis	**Number of warnings**
Smashing Analysis	8653
May Analysis	8658
May and Must Analysis (standard lub)	7503
May and Must Analysis (extended lub)	7503

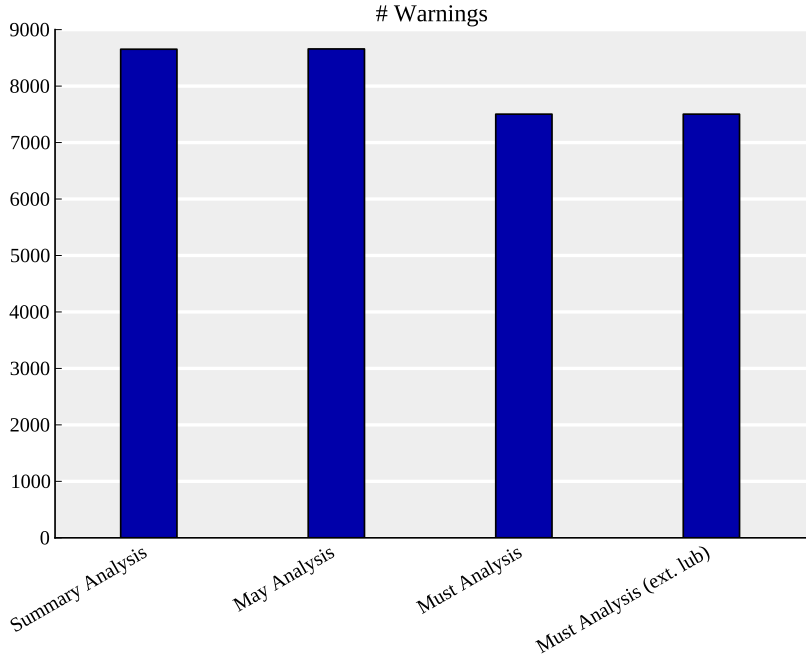

Figure 7.1: Number of warnings measured on 4422 scripts

We can see that the *Must Analysis* compared to the *May Analysis* reduces the number of warnings remarkably. It has 1150 warnings less than the *May Analysis*. In total 61 scripts produce less false alarms with the *Must Analysis* than they do with the *May Analysis*. Most of these false warnings were produced by the *May Analysis* because it was not able to prove that iterating over the keys of a map and then accessing the map at that key can never return an invalid object. Whenever this object then is accessed or passed as an argument a false alarm is raised. This case is also described in the example in Section 7.2.2.

Furthermore we can observe, that there is almost no increase in precision between the *Smashing Analysis* and the *May Analysis*. One factor that influences this result, is that all the elements that are added to a collection inside a loop are summarized as one element. In all these cases there is no difference between the *Smashing Analysis* and the *May Analysis*. We expect the *May Analysis* to become more precise in real world scripts, if a loop unrolling as explained in the example in Section 7.1.7 would be performed.

We can also see, that the extended least upper bound operator was not able to reduce the number of warnings in real world scripts. This mostly can be explained due to the fact that it only brings an advantage in precision in very specific cases as explained in Section 7.1.2.

Problems and Further Improvements

We could reduce the number of false alarms in 61 scripts. We initially expected to be able to decrease the number of false alarms in more scripts. The reasons that this was not the case are manifolded:

- A lot of scripts do not correctly handle special collection types such as JSON Objects or XML Objects. Often the collections are directly accessed at a key without checking if it is actually contained in the collection. This means that many warnings considering collections are correct warnings. However, if the programmers would correct their scripts

in these cases, only the *Must Analysis* would be precise enough to avoid the false alarms, as explained in the example in section 7.2.1. Among the analyzed scripts there are 329 scripts that use JSON Objects and for which this scenario applies.

- If a collection is stored in a global variable, Sample mistakenly detects this collection as a summary collection. Because it is not sound to add elements to a summary collection in the *Must Analysis* we are not able track which elements must be in these collections. Although the analysis is still sound in these cases, we loose precision and are often not able to prove the desired property.

- The *Must Analysis* usually only brings an advantage in precision, if the collections that we are analyzing are maps. If the collection has a linear key (is a list or a set) we can use the length abstraction to determine if at an accessed index there must be an element in the collection. This already allows to prove that a collection access never returns an invalid object.

- At the moment we do not perform a loop unrolling. This means that we can not gain any information for the *Must Analysis* from elements that are added to a collection inside a loop. We expect the results to improve if a loop unrolling is implemented.

- The precision for *String Maps* could be improved by using a relational String Domain. Some scripts check whether a string that is retrieved non-deterministically (e.g. through user input) is contained as a key in a map and then access the collection at that key. Only a relational String Domain would allow us to be able to prove that such an access never returns an invalid object.

7.3.2 Performance

For the performance measurements we analyzed 4635 scripts. Those were all the scripts where the analysis either completed successfully or could not finish in under a minute in one or more of the four analyses.

In the future the analysis shall be integrated into the TouchDevelop programming environment to analyze scripts while they are written. To be of use for the developer, the analysis should not take longer than a minute to be executed. When we run our experiments we therefore set a timeout of 60 seconds for each script. We measured for each analysis, how many of the 4635 scripts timed out. The results are denoted in table 7.2 and visualized in figure 7.2

Table 7.2: The number of scripts scripts where the analysis run longer than a minute (out of out of 4635 scripts)

Analysis	**Number of scripts timed out**
Smashing Analysis	116
May Analysis	173
May and Must Analysis (standard lub)	195
May and Must Analysis (extended lub)	213

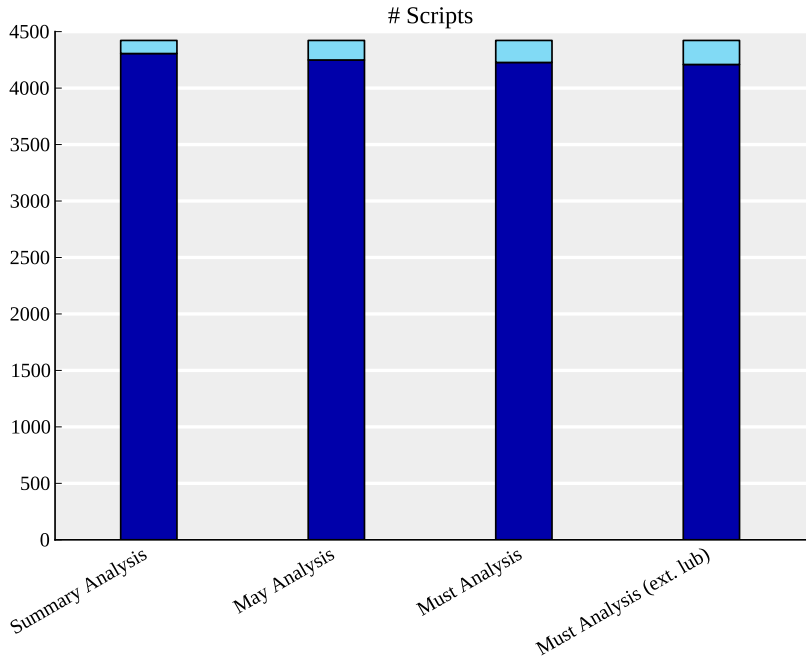

Figure 7.2: Number of scripts timed out (upper bar) compared to number of scripts not timed out (lower bar) with a timeout of 60 seconds

Furthermore we measured for each of the four analyses the average runtime per script over all the 4422 scripts that did not time out in any of the four analyses. The results of this measurement are denoted in table 7.3 and visualized in figure 7.3.

Table 7.3: The average runtime of the analyses over 4422 scripts

Analysis	**Average runtime [sec]**
Smashing Analysis	0.62
May Analysis	0.75
May and Must Analysis (standard lub)	0.82
May and Must Analysis (extended lub)	1.10

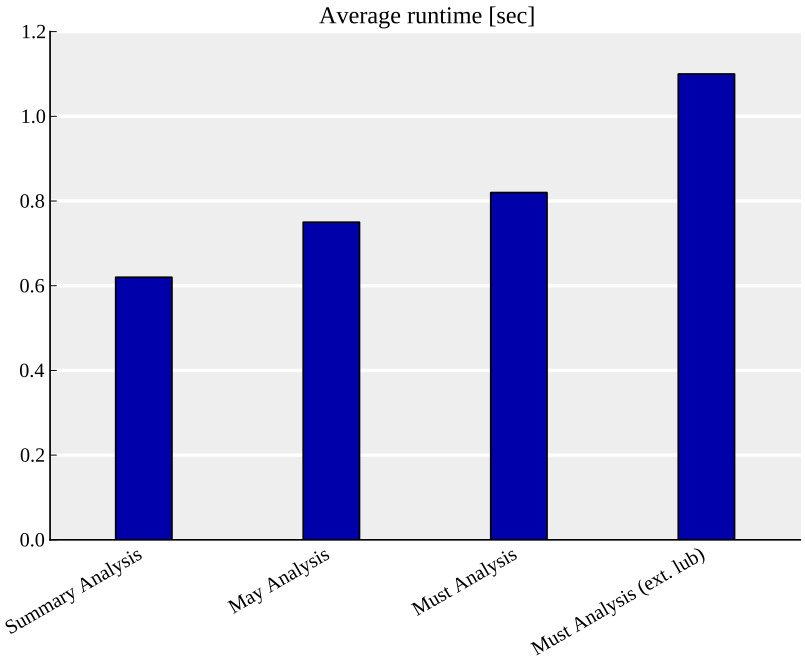

Figure 7.3: Average run time of the analyses measured on 4422 scripts

The *Smashing Analysis* which is expectedly the fastest analysis is not able to analyze 2.6 percent of the 4422 scripts in under a minute, whereas the slowest analysis the *May and Must Analysis* with the extended least upper bound operator is not able to analyze 4.8 percent of the scripts in under a minute. Although this is a significant difference, the *May and Must Analysis* is still able to analyze most of the scripts in under a minute. From the average runtime measurements we can learn that the extended least upper bound operator for the *Must Analysis* slows down the analysis significantly. Since we have also seen that the extended least upper bound does not bring an improvement in precision, we can conclude that at the moment it does not make sense to use the extended least upper bound operator for the *Must Analysis* in practice.

Further Improvements

The current implementation handles the *May* and the *Must Analysis* independently. This also means that we use different *Tuple Identifiers* for the *May* and for the *Must Analysis*. Therefore the number of *Tuple Identifiers* that we have to track doubles, if we combine the two analyses. This slows down the analysis remarkably. Instead of keeping separate *Tuple Identifiers* for the *May* and the *Must Analysis* it would be possible to keep the same *Tuple Identifiers* for both analyses and only track which *Tuple Identifiers* must be in the collection and which might be in the collection. With this change we would reduce the number of identifiers that need to be tracked by the value domain significantly. We expect that with this change the performance of the *May* and *Must Analysis* would improve.

Chapter 8

Conclusion

We successfully designed, implemented and formalized two analyses: One that over-approximates the elements in a collection and one that under-approximates the elements in a collection. We presented a technique that works with different collection types such as maps, lists or sets. We also defined, implemented and proven sound the concrete and the abstract semantics of collection operations for the TouchDevelop programming language.

We implemented the analyses as an extension to Sample which allowed us to use the value and heap domains as well as the analysis framework that were already implemented. Although the semantics for the collection operations are specific to TouchDevelop, the rest of the implementation is done in a generic fashion such that it could be used for other programming languages as well.

We could show on hand constructed and on real world TouchDevelop scripts that the *Must Analysis* improves the precision significantly. Compared to the baseline (the *Smashing Analysis*) we could reduce the number of warnings measured on 4422 scripts from 8653 to 7503. Additionally, we found out that a lot of TouchDevelop scripts are implemented poorly and do not handle corner cases correctly. As a consequence, a lot of alarms concerning collections are not false alarms. It can be expected that the *Must Analysis* will have a bigger impact on the precision once the developers start to correct their TouchDevelop scripts based on the provided alarms. We have also seen that the *May Analysis* which distinguishes individual elements based on allocation site does not bring a big improvement in precision when analyzing TouchDevelop scripts.

As expected, the performance of the analysis decreases when we do not summarize all collection elements with a single tuple. It also decreases significantly if we additionally to the *May Analysis* also run the *Must Analysis*. But we were still able to show that we can still analyze over 95% of the scripts scripts from the TouchDevelop cloud in reasonable time. We also found out that the extended least upper bound operator for the *Must Analysis* slows down the average run time of the analysis significantly, but does not bring an improvement in precision for real world scripts. It therefore does not make sense at the moment to use this extended least upper bound operator in practice.

We can finally conclude that the *Must Analysis* is a valuable asset when analyzing TouchDevelop scripts.

8.1 Related Work

There has already been work done in the field of statically analyzing the content of collections.

Gopan et al. [9] for example presented a technique to analyze operations on arrays, such that they can capture numeric properties of array elements. They partition array elements based on numeric relationships between array indices and the value of variables. All elements of an array that belong to the same partition are abstracted by one abstract element. The goal of this partitioning is to isolate the array elements that are assigned to. This allows to perform strong instead of weak updates and therefore improve the precision. Similar to our approach, they distinguish individual elements of the array to improve the precision for array operations, but in contrast to the work presented in this thesis it is only able to handle arrays. Furthermore our approach focuses on the property whether an element must be in a collection or not.

Lev-Ami et al. [13] present a technique called TVLA (3-Valued Logic Analysis) to automatically derive abstract semantics from concrete semantics based on logic predicates. They describe the shape of the heap in the abstract state with predicates using 3-valued logic. For example the information whether node u points to node v is represented with a predicate $u(v)$. Similar to our analysis they can then describe that u *must* point to v ($u(v) = 1$) that u *may* point to v ($u(v) = 1/2$) or that u can not point to v ($u(v) = 0$). There has also been work done to combine shape analysis such as TVLA with value domains [7], [10]. These techniques all analyze the actual implementations of collections. This can become inefficient or imprecise if the collections are implemented with complex data structures. In contrast to these techniques we used a high-level representation of collections which allows to define simpler and more efficient operations.

Marron et al. [14] also introduced a method that avoids analyzing the implementation of collections by abstracting the collections based on their semantics. They extend an existing heap analysis with semantics for high-level collection operations. They represent the semantics of collection operations with a shape analysis framework. These operations allow to refine summarized elements in the shape analysis such that strong instead of weak updates can be performed and therefore the precision for collection operations can be increased. We pursuit a similar approach by not considering the actual implementation of the collections in the library but rather defining a high-level representation of them. In contrast to the work of Marron et al. we make use of the value domain to refine the semantics for collection operations.

8.2 Future Work

In this last Section we want to show a few possibilities, how the presented analyses could be further extended and improved.

- *Capture collection elements added inside a loop:* We have shown in the case studies that the presented analysis is currently not able to capture collection elements that are added to a collection inside a loop. To be able to do this, it would be necessary to either implement loop unrolling or to further refine the currently used abstraction of *Elements*.
- *Reduce Number of Tuple Identifiers:* We have seen that the combined *May and Must Analysis* is significantly slower than the *May Analysis*. We have also elaborated that this is caused by the fact, that in the combined *May and Must Analysis* twice as many *Tuple Identifiers* as in the *May Analysis* need to be tracked. The efficiency of the presented technique could be further improved by using the same *Tuple Identifiers* for the *May* and the *Must Analysis*.

- *Support for composed keys:* Beside the standard collections TouchDevelop also supports more involved data structures called Records [18]. Records allow to have composed keys (e.g. a key composed of a String value and a Number value). Our approach could be extended to allow such types of keys as well.

Appendix A

Symbols

The following tables give an overview of the expressions used in this thesis.

A.1 Set Operators

Name	Symbol
Union	$\cup$
Intersection	$\cap$
Difference	$-$

A.2 Notations

A.2.1 Function update

We often need to create a new function based on another function, where only one key maps to a new value. We denote this in the following way: For a function $f : A \to B$ and $x \in A$, $y \in B$

$$f[x \mapsto y] \Leftrightarrow g(a) = \begin{cases} y & \text{if } a = x \\ f(a) & \text{otherwise} \end{cases}$$

A.3 Functions

Name	Description	Signature
α_{HId}	The abstraction function for Element abstraction	$\mathcal{P}(Ref) \to \mathcal{P}(HId)$
γ_{HId}	The concretization function for Element abstraction	$HId \to \mathcal{P}(Ref)$
α_V	The abstraction function for the value environment	$\mathcal{P}(Env_V) \to Env_V^A$
γ_V	The concretization function for the value environment	$Env_V^{(A)} \to \mathcal{P}(Env_V)$
γ_{May}	The concretization function for the May Analysis	$Env^{(A)} \to \mathcal{P}(Env)$
γ_C^{May}	The concretization function of the collection environment for the May Analysis	$Env_C^{(A)} \to \mathcal{P}(Env_C)$
γ_E^{May}	The concretization function of the Element Set for the May Analysis	$\mathcal{P}(TId) \to \mathcal{P}(\mathcal{P}(Ref))$
α_{May}	The abstraction function for the May Analysis	$\mathcal{P}(Env) \to Env^{(A)}$
α_C^{May}	The abstraction function of the collection environment for the May Analysis	$\mathcal{P}(Env_C) \to Env_C^{(A)}$
α_E^{May}	The abstraction function of the Element Set for the May Analysis	$\mathcal{P}(\mathcal{P}(Ref)) \to \mathcal{P}(TId)$
γ_{Must}	The concretization function for the Must Analysis	$Env^{(A)} \to \mathcal{P}(Env)$
γ_C^{Must}	The concretization function of the collection environment for the Must Analysis	$Env_C^{(A)} \to \mathcal{P}(Env_C)$
γ_E^{Must}	The concretization function of the Element Set for the Must Analysis	$\mathcal{P}(TId) \to \mathcal{P}(\mathcal{P}(Ref))$
α_{Must}	The abstraction function for the Must Analysis	$\mathcal{P}(Env) \to Env^{(A)}$
α_C^{Must}	The abstraction function of the collection environment for the Must Analysis	$\mathcal{P}(Env_C) \to Env_C^{(A)}$
α_E^{Must}	The abstraction function of the Element Set for the Must Analysis	$\mathcal{P}(\mathcal{P}(Ref)) \to \mathcal{P}(TId)$
$\sqcup_{May}$	The least upper bound operator on two Abstract Environments for the May Analysis	$(Env^{(A)} \times Env^{(A)}) \to Env^{(A)}$
$\sqcup_C^{May}$	The least upper bound operator on two Abstract Collection Environments for the May Analysis	$(Env_C^{(A)} \times Env_C^{(A)}) \to Env_C^{(A)}$
$\sqcup_{Must}$	The least upper bound operator on two Abstract Environments for the Must Analysis	$(Env^{(A)} \times Env^{(A)}) \to Env^{(A)}$
lubRep	The least upper bound operator with replacement on two Abstract Collection Environments for the Must Analysis	$\left(Env_C^{(A)} \times Env_C^{(A)}\right) \to \left(Env_C^{(A)} \times Replacement\right)$
$\sqcup_C^{Must}$	The least upper bound operator on two Abstract Collection Environments for the Must Analysis	$(Env_C^{(A)} \times Env_C^{(A)}) \to Env_C^{(A)}$

Name	Description	Signature
$getReplacements$	The function that creates a Replacement for the least upper bound of the Abstract Collection Environments for the Must Analysis	$Env_C^{(A)} \times Env_C^{(A)}) \rightarrow Replacement$
$assign_V$	The function to assign a value in the Value Environment	$(Elem \times \{"key","value"\}) \times V \times Env_V) \rightarrow Env_V$
$assign_V^{(A)}$	The function to assign a value in the Abstract Value Environment	$\left(\begin{array}{l} (TId \times \{"key","value"\}) \\ \times Expression \times Env_V^{(A)} \end{array} \right) \rightarrow Env_V^{(A)}$
$assume_V$	The function to assume an Expression in the Value Environment	$(Expression \times Env_V) \rightarrow Env_V$
$assume_V^{(A)}$	The function to assume an Expression in the Abstract Value Environment	$\left(Expression \times Env_V^{(A)}\right) \rightarrow Env_V^{(A)}$
$equals_V$	Checks whether an identifier equals an expression in the Value Environment	$((Elem \times \{"key","value"\}) \times V \times Env_V) \rightarrow \{true, false\}$
$equals_V^{(A)}$	Checks whether an identifier equals an expression in the Abstract Value Environment	$\left((TId \times \{"key","value"\}) \times Env_V^{(A)}\right) \rightarrow \{true, false\}$
$isBottom_V^{(A)}$	Checks whether even the key or the value field of a Tuple Identifier must be equal to bottom in the Abstract Value Environment	$(Env_V^{(A)} \times Replacement) \rightarrow Env_V^{(A)}$
$replace_V$	The function that applies replacement to the Abstract Value Environment	$(Env_V^{(A)} \times Replacement) \rightarrow Env_V^{(A)}$
$++$	Join operation of Replacements	$(Replacment \times Replacement) \rightarrow Replacement$

A.4 Sets

For sets we also state how a variable that represents an element of the set is usually called.

Name	Description	Variable names
Ref	Set of all memory locations of a program.	$r \in Ref$
$Elem$	Set of memory locations in which collection elements are stored ($Elem \subseteq Ref$)	$e \in Elem$
$\mathcal{P}(Elem)$	Concrete Element-Sets, defining which Elements are in a concrete collection.	$E \in \mathcal{P}(Elem)$
Env_C	The Concrete Collection Environment ($Ref \to \mathcal{P}(Ref)$)	$env_C \in Env_C$
V	The set of concrete values consisting of object-references and primitive values (Number, Boolean, String)	$v \in V$
Env_V	The Concrete Value Environment ($(Ref \times \{key, value\}) \to V$)	$env_V \in Env_V$
Env	The Concrete Environment ($Env_C \times Env_V$)	$env \in Env$ or $(env_C, env_V) \in Env$
HId	The Set of Heap Identifiers	$h \in HId$
TId	The Tuple Identifiers ($\mathcal{P}(HId)$)	$t \in TId$
$Expression$	The set of Expressions that are used in the value domain	$expr \in Expression$
$\mathcal{P}(TId)$	The Abstract Element-Set, defining which Tuple Identifiers are in an abstract collection	$c^{(A)} \in \mathcal{P}(TId)$
$Env_C^{(A)}$	The Abstract Collection Environment ($HId \to \mathcal{P}(TId)$)	$env_C^{(A)} \in Env_C^{(A)}$
$Env_V^{(A)}$	The Abstract Value Environment	$env_V^{(A)} \in Env_V^{(A)}$
$Env^{(A)}$	The Abstract Environment ($Env_C^{(A)} \times Env_V^{(A)}$)	$env^{(A)} \in Env^{(A)}$
$Replacement$	The replacements of Heap Identifiers ($\mathcal{P}(HId) \to \mathcal{P}(HId)$)	$rep \in Replacement$

Appendix B

General Proofs

Theorem 9. $\{X \odot R : X \supseteq T\} \subseteq \{X : X \supseteq (T \odot R)\}$ *for all sets* R *and* T *and* $\odot \in \{\cup, -\}$

Proof. We are going to show that if $z \in \{X \odot R : X \supseteq T\}$ it follows that $z \in \{X : X \supseteq (T \odot R)\}$ which by the definition of $\subseteq$ proves the theorem.

$$\begin{aligned} & z \in \{X \odot R : X \supseteq T\} \\ & \rightarrow (z = X \odot R) \wedge (X \supseteq T) \end{aligned}$$

Since $(X \supseteq T) \rightarrow ((X - R) \supseteq (T - R))$ and $(X \supseteq T) \rightarrow ((X \cup R) \supseteq (T \cup R))$ for all sets X, T and R

$$\begin{aligned} & (z = X \odot R) \wedge (X \supseteq T) \\ & \rightarrow (z = X \odot R) \wedge (X \odot R \supseteq T \odot R) \\ & \rightarrow z \supseteq (T \odot R) \end{aligned}$$

And since we have shown that $z \supseteq (T \odot R)$ we can conclude $z \in \{X : X \supseteq (T \odot R)\}$ □

Theorem 10. $\{X \odot R : X \subseteq T\} \subseteq \{X : X \subseteq (T \odot R)\}$ *for all sets* R *and* T *and* $\odot \in \{\cup, -\}$

Proof. We are going to show that if $z \in \{X \odot R : X \subseteq T\}$ it follows that $z \in \{X : X \subseteq (T \odot R)\}$ which by the definition of $\subseteq$ proves the theorem.

$$\begin{aligned} & z \in \{X \odot R : X \subseteq T\} \\ & \rightarrow (z = X \odot R) \wedge (X \subseteq T) \end{aligned}$$

Since $(X \subseteq T) \rightarrow ((X - R) \subseteq (T - R))$ and $(X \subseteq T) \rightarrow ((X \cup R) \subseteq (T \cup R))$ for all sets X, T and R

$$\begin{aligned} & (z = X \odot R) \wedge (X \subseteq T) \\ & \rightarrow (z = X \odot R) \wedge ((X \odot R) \subseteq (T \odot R)) \\ & \rightarrow z \subseteq (T \odot R) \end{aligned}$$

And since we have shown that $z \subseteq (T \odot R)$ we can conclude $z \in \{X : X \subseteq (T \odot R)\}$ □

Appendix C

Replacement Concatenation

In some situations it is desirable to concatenate two *Replacements*. The goal of the concatenation is to find a *Replacement* such that when it is applied to a value domain it produces the same result as if the two *Replacements* were applied consecutively.

Formally, let rep_1 and rep_2 be two *Replacements*, then the concatenation operation $\gg$ finds a *Replacement* $rep = rep_1 \gg rep_2$ such that for all *Abstract Value Environments* the following holds:

$$replace_V\left(replace_V\left(env_V^{(A)}, rep_1\right), rep_2\right) = replace_V\left(env_V^{(A)}, rep\right)$$

If we have for example two Replacements $rep_1 = [\{hId_1\} \to \{hId_2\}]$ and $rep_2 = [\{hId_2\} \to \{hId_3\}]$ the concatenation produces a new *Replacement* $rep = [\{hId_1\} \to \{hId_3\}]$.

The concatenation operator needs to exchange all ids in the right *Replacement*'s *from* set that occur in one of the right *Replacement*'s *to* sets with the value of the right *Replacement*'s *from* set. Furthermore the concatenation needs to preserve all *Replacements* that are not affected by an identifier exchange.

We formally define the concatenation operation $\gg$ on *Replacements* as follows:

$$\gg: (Replacement \times Replacement) \to Replacement$$

$$rep_1 \gg rep_2 = \left[\begin{array}{ll} f \to t: & (l \in dom(rep_1) \wedge f = l \wedge t = (rep_1(l) - (\bigcup_{I \in dom(rep_2)} I))) \vee \\ & (r \in dom(rep_2) \wedge f = replace(r, rep_1^{-1}) \wedge t = rep_2(r)) \vee \\ & (f \in ((\bigcup_{I \in dom(rep_1)} rep_1(I)) \cap (\bigcup_{I \in dom(rep_2)} I)) \wedge t = \emptyset) \end{array}\right]$$

where rep^{-1} is the inverse function of the *Replacement* rep and

$$replace : (\mathcal{P}(HId) \times Replacement) \to \mathcal{P}(HId)$$

$$replace(S, rep) = (S - (\cup_{I \in dom(rep)} I)) \cup (\cup_{I \in dom(rep)} rep(I))$$

Bibliography

[1] B. Blanchet et al. "Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, invited chapter". In: *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*. 2002, pp. 85–108.

[2] D. R. Chase, M. Wegman, and F. K. Zadeck. "Analysis of pointers and structures". In: *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. 1990, pp. 296–310.

[3] G. Costantini, P. Ferrara, and A. Cortesi. "Static Analysis of String Values". In: *Proceedings of the 13th International Conference on Formal Engineering Methods (ICFEM 2011)*. 2011, pp. 505–521.

[4] P. Cousot and R. Cousot. "Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *In Proceedings of POPL 77*. 1977, pp. 238–252.

[5] P. Cousot and R. Cousot. "Systematic Design of Program Analysis Frameworks". In: *In Proceedings of POPL 79*. 1979, pp. 269–282.

[6] P. Ferrara. "Static Type Analysis of Pattern Matching by Abstract Interpretation". In: *InProceedings of FMOODS 10*. 2010, pp. 186–200.

[7] P. Ferrara, R. Fuchs, and U. Juhasz. "TVAL+ : TVLA and Value Analyses Together". In: *Proceedings of the Tenth International Conference on Software Engineering and Formal Methods (SEFM 2012)*. 2012, pp. 63–77.

[8] P. Ferrara and P. Müller. "Automatic inference of access permissions". In: *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*. 2012, pp. 202–218.

[9] D. Gopan, T. Reps, and M. Sagiv. "A framework for numeric analysis of array operations". In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2005, pp. 338–350.

[10] D. Gopan et al. *Numeric Domains with Summarized Dimensions*. 2004.

[11] N. Horspool et al. "TouchDevelop Programming on a Phone. Version 1.1 for TouchDevelop 2.8". Microsoft Research, 2012.

[12] B. Jeannet and A. Miné. "Apron: A library of numerical abstract domains for static analysis". In: *Proc. of the 21st International Conference on Computer Aided Verification (CAV 2009)*. 2009, pp. 661–667.

[13] T. Lev-Ami and M. Sagiv. "TVLA: A System for Implementing Static Analyses". In: *Proceedings of the 7th International Symposium on Static Analysis*. 2000, pp. 280–301.

[14] M. Marron et al. “Heap analysis in the presence of collection libraries”. In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 2007, pp. 31–36.

[15] A. Miné. “The octagon abstract domain”. In: *Higher Order Symbol. Comput.* (2006), pp. 31–100.

[16] M. Zanioli, P. Ferrara, and A. Cortesi. “SAILS: static analysis of information leakage with Sample”. In: *Proceedings of the 27th ACM Symposium on Applied Computing (SAC 2012)*. 2012, pp. 1308–1313.

[17] *APRON numerical abstract domain library*. URL: `http://apron.cri.ensmp.fr/` (visited on 09/01/2013).

[18] Microsoft Research. *Introduction to records in TouchDevelop*. URL: `https://az31353.vo.msecnd.net/cpd/xeuo-records.pdf` (visited on 09/01/2013).

[19] Microsoft Research. *TouchDevelop*. URL: `http://research.microsoft.com/en-us/projects/touchdevelop/` (visited on 09/01/2013).

[20] Microsoft Research. *TouchDevelop API v2.11*. URL: `https://www.touchdevelop.com/help/api/200110` (visited on 03/27/2013).