Chair of
Software Engineering

PARALLELISM VISUALIZER FOR SCOOP

MASTER THESIS

Dominic Meier
ETH Zurich
meiedomi@student.ethz.ch

July 1, 2014 - December 31, 2014

Supervised by:
Mischael Schill
Prof. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Parallelism Visualizer for SCOOP

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Meier	Dominic

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Riniken, December 18, 2014	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Abstract

SCOOP (Simple Concurrent Object-Oriented Programming) is designed to increase a programmer's ability to reason statically about the correctness of a concurrent program. However, expectations regarding the degree of achievable parallelism may be misleading, as seemingly asynchronous calls may be executed synchronously by the runtime system in order to decrease the chances of deadlock scenarios. The aim of this thesis is to provide users of EVE – the research branch of EiffelStudio – with additional insights that help them understand and foresee the parallel behavior of their SCOOP programs.

We approached this goal by designing an inter-procedural user-guided abstract interpretation in conjunction with an add-on for EVE that allows users to visualize, navigate and manually refine the analysis' findings. The analysis keeps track of the attachment of variables to processor regions among and across routines and classifies every reachable routine call in the source code depending on whether it is or may be asynchronous at runtime or not. Results include the formal specification as well as the implementation of the abstract interpretation, in addition to the integration of the add-on into EVE.

Acknowledgments

I would like to thank Mischael Schill for supervising my thesis, especially for providing valuable feedback on the report and keeping me motivated. Furthermore, I would like to thank Julian Tschannen for giving me a quick overview of the software architecture in EiffelStudio.

Last but not least, I thank Bertrand Meyer and everyone in the chair of software engineering for having such a huge impact on my interests in the field of computer science throughout my education at ETH, and for giving me the opportunity to work on this master thesis.

Contents

Chapter 1

Introduction

Note that throughout this introduction, we assume the reader's familiarity with the core concepts of SCOOP[1].

1.1 Problem

Parallel programming in Eiffel with SCOOP is easy. SCOOP is best described via its name: **S**imple **C**oncurrent **O**bject-**O**riented **P**rogramming. But what does it even mean, *simple*?

There are many ways to quantify this term. But one word summarizes the benefits of SCOOP best: *Reasonability*. It measures the programmer's ability to reason statically (i.e. just by inspecting the source code) about the program's execution.

In many ways, SCOOP adheres to reasonability. However, not every aspect of a program's execution can be reasoned about in equal measure. SCOOP does especially well when it comes to reasoning about *correctness* of a program, but suffers a bit when it comes to the assessment of the *concurrent behavior* of a program.

Oftentimes, it is not clear if a seemingly asynchronous separate routine call – as suggested by the type system – is indeed asynchronous at runtime. There are many concepts behind the scenes that strive to reduce the number of possible deadlocks. The most notable mechanism is called *lock-passing*, which causes an otherwise asynchronous call to wait, while avoiding the possibility of a deadlock occurring later on.

Another uncertainty is the runtime type of entities that are statically declared with the `separate` modifier. This type modifier only says that the runtime object *may* be located on some other processor region. It may as well be on the same processor region, and therefore act as a synchronous actor relative to the current object. The source code, when inspected locally, does not give any hints concerning this ambiguity. A programmer would have to simulate the whole program in his or her head, in order to grasp the topology of SCOOP objects and regions. For authors of the program, it is often easier to know about this topology than it is for non-authors that may want to understand the code or adapt it.

Despite all the efforts that aim at reducing deadlock scenarios, SCOOP is still *not free* of them. As a matter of fact, the deadlocks that *can* occur, are at the same time extremely obscure. Without dismantling the program's execution traces, it is almost impossible to find these deadlocks' causes.

[1]A good introduction to SCOOP can be found here:
`https://docs.eiffel.com/book/solutions/concurrent-eiffel-scoop`, as retrieved on 2014/12/14

This situation poses a problem that asks for solutions.

1.2 Existing Solutions

Currently, there is only one possible solution to this problem. It comes under the name of *processor tags* and was first introduced by Piotr Nienaltowski in [8]. Processor tags are designed to statically assign names to processors. They can be used in the type declaration of separate entities. An example declaration looks like this:

```
x: separate <px> X
y: separate <px> Y
```

where `<px>` is the processor tag. Entities `x` and `y` therefore are declared to be located on the same processor region.

As soon as processor regions can be distinguished statically, more fine-grained control over the runtime behavior of the program is obtained. This is an advantage over the current SCOOP model[2], where it is not possible to create new objects on already existing regions.

Processor tags are also useful in other applications. In his dissertation [11], Scott West designed and implemented a static deadlock detection mechanism as a separate tool that only works in conjunction with processor tags.

Limitations of this Solution

Sadly, processor tags are not yet implemented in EiffelStudio. But even if they were, the problem is not entirely gone: Two distinct processor tags do not imply that the runtime processors they refer to are also distinct. And in the example declaration above, it is not clear if `x` and `y` are located on a different region than the `Current` object. Hence the still existing `separate` keyword. The only thing processor tags can guarantee is that two objects are located on the same processor region if their tags match.

A more general solution to the initial problem may therefore also benefit a future SCOOP implementation that supports processor tags.

1.3 Our Solution

The approach taken in this thesis is different than what processor tags do. We did not make any changes to the syntax and/or semantics of the existing SCOOP. Instead, we have developed a *static analysis* that inspects the source code of a SCOOP program (not just a single file or routine, but rather the whole system) and gathers useful information that help programmers get a more accurate picture of the processor region topology as well as the runtime nature of separate routine calls, which in turns helps evaluating the degree of parallelism of a SCOOP program.

Furthermore, we implemented an extension to EVE[3], the research branch of EiffelStudio, that allows programmers to visualize the findings of the static analysis and refine it with more precise information if desired. This turns the analysis into a *user-guided* one.

[2]the standard implementation of SCOOP in EiffelStudio 14.05
[3]**E**iffel **V**erification **E**nvironment

For the static analysis, we use an *inter-procedural abstract interpretation framework* [1], specifically designed for SCOOP. We call it *region analysis*. It is implemented entirely in sequential Eiffel, just as the visualization and interaction tool called *parallelism visualizer* that comes with it. Both are tightly integrated into the architecture of EVE.

1.4 Research Contribution

The main contribution of this thesis is a fully formalized specification of an instance of a user-guided abstract interpretation framework in the field of concurrent programming. It may be generalizable to other programming languages that offer a similar model as SCOOP, although the latter is quite unique in the field.

1.5 Outline

Apart from the introduction and conclusion, the thesis is broadly split into two main parts:

Theoretical part In the first part, we formally define the region analysis on a simplified version of SCOOP.

Practical part In the second part, we discuss choices and details of the analysis' implementation in EVE. Also, we are discussing the design of the visual tool in EVE.

A more detailed outline for these two parts is given in the following.

Theoretical Part

Throughout this part, we combine informal and formal descriptions in such a way, that it benefits the understanding.

Simplified SCOOP Language and Semantics Eiffel (with and without SCOOP enabled) is an advanced language with lots of features. In order to simplify and streamline our discussion and formalization, we first define a simplified version of SCOOP. This includes an informal specification of the syntax and some aspects of the semantics. This language will then be the reference for the entire theoretical part of the thesis.

Concrete and Abstract Domains In this chapter, we describe the concrete state of a SCOOP program, as well as the approximative abstract states that govern the analysis. We show that the abstract states form a finite complete lattice, and we define the representation function that abstracts a concrete state, and show that this approximation is safe.

Abstract Interpretation Framework This is the core of the thesis. We describe the overall inter-procedural abstract interpretation framework. All transfer functions are formally specified, first on the intra-procedural level, later on a inter-procedural level. Moreover, we introduce routine call classifications, which are the central information used by the visualization tool.

Practical Part

This part focuses on implementation aspects of the region analysis, and describes the parallelism visualizer in more detail.

Implementation In this chapter, we address implementation-specific adjustments to our abstract interpretation framework. We are quickly talking about the integration into EVE. The main focus however is on the data structures used to realize state graphs, and how they can be refined by the user. Finally, we present a modification of the region analysis to incorporate a simple deadlock detection scheme, as well as a more sophisticated set of ideas that would allow for a more advanced deadlock detection.

Parallelism Visualizer Finally, we describe the goals and design decisions of the visual tool. We give a few implementation-related comments and go over a step-by-step real-world example of the tools' usage in EVE.

Chapter 2

Simplified SCOOP Language and Semantics

The Eiffel programming language with enabled SCOOP is quite extensive, and its semantics are involved. In order to keep things simple and understandable, we focus on only a subset of all language constructs, and define their semantics in an informal way. This way, we can focus on only those aspects that are relevant for the region analysis, while still retaining the key elements of the real SCOOP.

Notice that the implementation of the region analysis operates on the full-fledged Eiffel SCOOP language, and not on this simplified version.

2.1 Simplified SCOOP Language

The following list informally yet precisely describes a simplified Eiffel language that is entirely based on the features and semantics of the original language. All retained elements are listed, while some thrown-away elements are mentioned for clarification. Everything that is not mentioned is not part of the simplified language. The accompanying SCOOP semantics will be described in-depth in a later section.

Type system
: Only non-generic types are considered. User-defined expanded types as well as predefined expanded types (integers, characters, etc.) are not considered. Every entity in the program is of reference type. However, `Void` values are still allowed. The only type modifier that is allowed is `separate`. This means in particular, that every type has exactly one corresponding class and every class `T` has exactly two corresponding types: `T` and `separate T`. Note that `detachable` remains unconsidered as well.

Inheritance
: Inheritance and its associated concepts (polymorphism, dynamic binding) are powerful mechanisms in object-oriented programming. However, we do not consider them for this simplified language, since this would hinder clarity and simplicity. Chapter 5 discusses how these mechanisms are incorporated into the analysis' implementation. In other words, the class hierarchy is completely flat, and classes as well as their derived types are pairwise non-conforming.

Conformance
Conformance is defined as follows: An entity of type `T` conforms to any entity of type `T` or `separate T`, and an entity of type `separate T` only conforms to an entity of the same type.

Features
As in the full language, features can be of two essential kinds: *Routines* and *attributes*. Routines are further classified as *procedures* (no result) and *functions* (compute a result). Creation procedures are not distinguished from any other procedure. Also, once routines, built-in (or external) routines, feature aliases, assigner commands and converters are not considered.

Agents
Agents (objects representing delayed operations) are not considered.

Entities
Five kinds of entities are considered:

- formal routine arguments (not writable)
- local variables (must be declared at the beginning of a routine body)
- class attributes
- `Current` (not writable)
- `Result` (has local scope and is only allowed inside function bodies)

In particular, no ad-hoc local entities (as in `attached as`, or cursor names in `across` loops) are considered.

Contracts
Contracts – preconditions, postconditions and invariants – are not considered. Preconditions have special semantics in SCOOP programs, where they act as waiting conditions. The region analysis does not give any insights into the runtime behavior of waiting conditions, and therefore they can be safely ignored. Also, there is still no agreed behavior on all the runtime effects of contracts in SCOOP programs. [9] discusses this topic.

Expressions
Legal expressions consist of either the literal value `Void` or a chain of one or more entity/feature accesses (separated by a dot). The involved accesses must be either functions (with potential actual arguments that are also legal expressions) or attributes, with the exception of the first access in the chain which can also be a local variable or a formal argument.
Since we are dealing with SCOOP programs, there is an additional rule for legal expressions. For every access that is part of a non-empty access chain, exactly one of the following conditions must hold:

- It is an access to a formal argument with a separate type at the beginning of the chain.
- It is an access with a non-separate type.

Statements
The kinds of statements considered are:

- simple loops (i.e. `until`-condition and body)
- simple branches (only `if` and `else`, no `elseif` or `inspect`)

Listing 2.1: Eiffel*: simplified SCOOP program*

```
class APPLICATION

create
  make

feature
  make
    local
      l_worker: separate WORKER
    do
      create l_worker.make
      create toolkit.make
      kick_on_worker (l_worker)
    end

  kick_on_worker (a_worker: separate WORKER)
    do
      if a_worker.is_motivated then
        until
          a_worker.done
        loop
          a_worker.do_work (toolkit)
        end
      else
        a_worker.be_lazy
      end
    end

  toolkit: separate TOOLKIT
end
```

- unqualified and qualified (i.e. applied on some expression) procedure calls with potential actual arguments
- assignments
- creations (consisting of a writable entity name, followed by some procedure name)

Note that the usual rules for assignments apply: Legal targets are only writable entities. The type of the source expression must conform to the type of the target.

Listing 2.1 shows a sample program containing at least one instance of all programming constructs present in the simplified SCOOP language.

2.2 SCOOP Semantics

Describing the semantics of SCOOP programs is a rather involved task in and by itself. This is why we are referring the reader to other publications/materials to get an in-depth introduction into SCOOP. Among them are [8, 5].

2.2.1 Ambiguous Semantics

There is not an agreed reference semantics for SCOOP programs. Some semantics have been proposed only theoretically, others actually exist in the form of an implementation. They all share the basic design decisions behind SCOOP, but there are still many subtleties by which they differ. The following list names two key properties by which different semantics can disagree.

Kinds of deadlocks It is possible that a given program may deadlock in one semantics, but can not in the other.

Degree of parallelism Some programs can have a higher degree of parallelism in one semantics than in the other, given that the number of physical cores and any other environmental properties are kept invariant.

Note that many negative aspects of concurrent programming are eliminated in SCOOP by design. Among them are *race conditions* and some forms of *starvation*.

The semantics we use for the region analysis is the one currently present in *EiffelStudio 14.05*. Since there is no written specification for this semantics, an extensive test-suite[1] was used to determine the exact behavior of the runtime in many different situations. The empirically evaluated semantics can best be described as a mix of the ones described by Benjamin Morandi in [5] and Scott West in [11] with some additional tweaks. In the following, we describe this semantics as precisely as possible, starting with the backbones of SCOOP.

2.2.2 The `separate` Keyword

The key element that differentiates a SCOOP program from a sequential Eiffel program is one keyword: `separate`. This keyword acts as a *type modifier*, i.e. it creates a new but related type from a given class type. Then, the parallel behavior of the program is determined by which types have this modifier and which do not. The `separate` keyword is also the only syntactic addition to the basic Eiffel language.

2.2.3 Stack, Heap, Regions and Processors

Like many other languages, Eiffel follows the stack and heap model. This means that entities can either be dynamically allocated on the heap – which is an area in the virtual memory space – at runtime using the `create` instruction, or they are automatically allocated on the call stack by the Eiffel compiler at compile time. Note that object attributes are always allocated on the heap before the associated creation procedure is called via `create`. Routine arguments and local variables are always automatically allocated on the call stack at the moment a routine gets called.

The SCOOP model refines the stack and heap model with two important notions: *Regions* and *processors*. A region is a slice of the heap's memory. All regions together form the heap, and no two regions overlap. Every object at runtime can only be allocated on one region for its entire lifetime. Note that a region may contain several objects.
A processor on the other hand is a logical independent computing unit (that may or may not be directly mapped to a physical CPU core) that conceptually runs concurrently to

[1]An Eiffel project containing various manual tests that trigger all kinds of special conditions by which different semantics may differ.

other processors. Behind the scenes, it is still possible that two distinct processors can run on the same physical core but in an interleaving fashion, just like threads can as well. Every processor maintains its own stack, and is tightly coupled with exactly one region on the heap. Therefore, a processor can only access objects that are allocated on its own region. Since every object and region is handled by exactly one processor, this processor is uniquely referred to as the object's or region's *handler*.

Thanks to this simple yet effective design decision, data races are completely eliminated, since shared memory accesses are not even possible. In fact, SCOOP closely follows the actor model[2]: Processors are actors watching over their own memory, and communicate with other actors via message passing.

2.2.4 Message Passing via Feature Calls

The message passing system is hidden behind standard feature calls. A non-separate feature call issued by some processor works just as in the sequential world: Actual arguments are pushed onto the call stack and execution jumps to the called routine's first statement. However, if the feature call is issued on a reference to an object that is located on a different processor region in the heap, special rules apply. A distinction is made between feature *separate call* and feature *application*:

Separate call The client processor constructs a call message that contains the target object reference, the name of the called feature and a copy of all actual arguments. This message is then enqueued by the runtime system to the so-called *request queue* of the processor that handles the target object – called the target processor.

Application The behavior of every processor in the system is as follows: It takes messages off its request queue one after the other and executes them (or rather, what they ask for) on its own call stack. This execution is then referred to as feature *application*.

If the client processor requires a result of the called separate feature (which in this case must be an attribute lookup or function call), it waits for the feature application to finish, retrieves the result, and continues its own execution. This kind of wait is referred to as *wait by necessity*.

2.2.5 Ensuring Exclusive Access

The main benefit of the SCOOP model is that it injects the underlying synchronization mechanisms tracelessly into a sequential program structure: Before a routine that was called by some client processor gets executed, its handling processor atomically acquires a *private frame for call messages* on each of the actual arguments' handling processors. Then, every separate call that gets issued to the same target processor inside the routine's body is guaranteed to be eventually applied by the target processor in that same order and without interruption. This property enables programmers to reason locally.

2.2.6 Queue of Queues

The *request queue* mentioned above is not just a single queue. In the model we are following here, it is a *queue of queues*, as introduced in [13].

[2]`http://en.wikipedia.org/wiki/Actor_model`, as retrieved on 2014/12/14

The top-level queue – referred to as *request queue* in the remainder – contains *private queues* that are associated with other client processors in the system. A client processor (the caller) always enqueues its call messages to its own private queue in the request queue of the target processor.
With this mechanism, a client processor never has to wait in order to issue new call messages, as no other processor has access to its private queue in a particular request queue on a particular target processor.

2.2.7 Locks and Lock-Passing

Lock-passing [7] is a more advanced feature of the SCOOP runtime that aims at avoiding certain types of deadlocks. Its existence is one of the main motivations for the usage of the region analysis, as it influences the runtime behavior – especially the degree of parallelism – substantially.

A *lock* in this context can be either of the following:

Call-Stack-Lock (CSL) Every processor in the system owns a unique CSL. Holding it gives a processor (any processor) the exclusive right to execute statements on the call stack of the owner of the lock. In particular, the owner of a CSL cannot make any progress if it is not the holder.

Request-Queue-Lock (RQL) Every RQL is associated with a private queue in a request queue of some processor in the system. It gives its holder the right to enqueue new call messages to this private queue. Every time a new private queue is created by the runtime system, a new RQL is created alongside it and given to the processor that initiated the creation of this private queue.

At any time, a processor can hold zero or more CSLs, and zero or more RQLs. Together, they are referred to as the processor's *locks*.

Lock-passing refers to the act of passing locks from one processor to another, i.e. changing the locks' holder. Lock-passing can only occur alongside issuing a new separate call message, and if it occurs, *all* currently held locks of the client processor are passed (never a subset of them). The following conditions must hold for it to happen:

- The call must be separate, because otherwise there is no call message created.
- The call must have at least one actual argument.
- At least one such actual argument must refer to an object that is handled by a processor for which the client processor holds a lock (CSL or RQL).

Whenever lock-passing is triggered, the client processor will *wait* for the call to be applied and completed by the target processor, essentially turning the call into a synchronous one. Then, the runtime system will make sure that the target processor hands the passed locks back to the client processor. So, lock-passing is a process that gets automatically inverted, eventually. Note that a lock can be passed multiple levels away from its original holder. This happens when the target processor itself calls another processor in a way that triggers lock-passing. And so on.

The upside of this mechanism is that several types of deadlocks can be avoided. The downside of it is two-fold: On one hand the degree of parallelism is reduced, since asynchronous calls are turned into synchronous ones, and on the other hand programmers cannot always know if lock-passing may happen at runtime by looking at part of the source code alone. The region analysis aims at improving this situation.

Chapter 3

Concrete and Abstract Domains

Before we specify the region analysis, we have to define the runtime states our simplified SCOOP language operates in. These states are part of a *concrete domain*. The elements of the *abstract domain* on the other hand are where the region analysis operates in. These two domains are related via a representation function that maps an element from the concrete domain to an element of the abstract domain, where the latter approximates the former.

3.1 Concrete Domain

The *concrete domain* is a mathematical description of a *part* of the state a SCOOP program can be in at runtime at a given program point. As described in chapter 2, the program state of a SCOOP program consists of a set of regions that each contains at least one object. Every region is handled by a distinct logical processor which maintains its own call stack. Since our analysis operates from the viewpoint of some executing routine – called the *target routine* – on some target object in the heap, we only focus on the topmost frame in the call stack, i.e. the one corresponding to the currently executed routine.

We disassemble this part of the stack into *formals* (also containing the `Current` pointer), *locals*, and *expression evaluation results*. Furthermore, we record the set of held locks (either call stack locks or request queue locks) of only the processor that executes the target routine. Note that for the formalization of the region analysis, the type of a lock is not important. Therefore, we unify them into just *locks*. Also, it does not matter who actually owns a lock. Only a lock's holder is important, since this is what influences lock-passing.

In the following, we develop the mathematical model that describes the concrete domain. We do not specify the SCOOP semantics formally (which can be done by using e.g. denotational semantics or structural operational semantics). This has been done before in [5] respectively [11].

We abbreviate the concrete domain with V, and define it as follows.

$$V := \mathbf{Formals} \times \mathbf{Locals} \times \mathbf{Stack} \times \mathcal{P}(\mathbf{Reg}) \times \mathcal{P}(\mathbf{Reg}),$$
$$(formals, locals, stack, locks, heap) \in V$$

Note that **Stack** only contains the temporary expression evaluation results. So, if we have an expression that consists of an access to a local variable, the effect of the evaluation of this expression will be that another element is pushed onto the $stack$, while $locals$ remains unchanged.

From now on, we use the notation $tuple_{\text{entry}}$ to access a certain element (here *entry*) in a certain tuple (here *tuple*). For this purpose, we have to name all entries of a tuple. We usually do this alongside the definition of the corresponding set, just like in the definition of V above.

$$\begin{aligned}
\textbf{Formals} &:= \textbf{Lab} \to \textbf{Loc} \\
\textbf{Locals} &:= \textbf{Lab} \to \textbf{Loc} \\
\textbf{Stack} &:= (\textbf{Loc} \cup \{\texttt{Void}\})^* \\
\textbf{Reg} &:= \textbf{Loc} \to \textbf{Lab} \to \textbf{Loc} \cup \{\texttt{Void}\}
\end{aligned}$$

Note that none of the above functions is a *total function* (that is, they are all partial functions). There might be objects that do not have any attributes. This is why **Reg** is allowed to map a location to the *empty function*, which does not map any attribute names, but nevertheless exists as a mathematical object.

The set **Lab** consists of all legal entity names. We adopt the rules of the Eiffel language itself, and use a regular expression for the definition:

$$\textbf{Lab} := \text{[A-Za-z][A-Za-z0-9\_]}^*$$

The set **Loc** contains all legal locations in the heap. A location is an address in the virtual memory space of the executing process, so we may set $\textbf{Loc} := \mathbb{N}$. Note that in reality, the address space is bounded. But for the sake of simplicity, we assume it is unbounded. The special value `Void` denotes the value of a non-attached reference, just as in Eiffel.

Not all states of V are actually *legal*. Whether or not a state is legal is determined by the set I_{concrete} of the following invariants. All of these invariants are viewed as functions that take a concrete state $v \in V$ and evaluate to either true or false, depending on whether or not v satisfies the proposition.

Invariant 3.1.1 (all regions are non-empty).

$$\forall r \in v_{\text{heap}} \colon \left|\text{dom}(r)\right| > 0$$

Invariant 3.1.2 (regions are pairwise disjunct).

$$\forall r_1, r_2 \in v_{\text{heap}}, r_1 \neq r_2 \colon \ \text{dom}(r_1) \cap \text{dom}(r_2) = \emptyset$$

Invariant 3.1.3 (no unknown locations).

$$\forall x \in \text{im}(v_{\text{formals}}) \cup \text{im}(v_{\text{locals}}) \cup v_{\text{stack}} \colon x \in \bigcup_{r \in v_{\text{heap}}} \text{dom}(r)$$

Invariant 3.1.4 (locks on existing regions).

$$\forall r \in v_{\text{locks}} \colon r \in v_{\text{heap}}$$

Invariant 3.1.5 (`Current` always exists).

$$\texttt{Current} \in \text{dom}(v_{\text{formals}})$$

Invariant 3.1.6 (all regions containing a formal label are locked)**.**

$$\forall r \in v_{\text{heap}} \colon (\exists l \in \text{im}(v_{\text{formals}}) \colon l \in \text{dom}(r)) \implies r \in v_{\text{locks}}$$

Invariant 3.1.7 (no formal has the same label as a local)**.**

$$\text{dom}(v_{\text{formals}}) \cap \text{dom}(v_{\text{locals}}) = \emptyset$$

Invariant 3.1.8 (heap attributes point to existing locations)**.**

$$\left(\bigcup_{r \in v_{\text{heap}}} \cup_{l \in \text{im}(r)} \text{im}(l) \right) \subseteq \left(\bigcup_{r \in v_{\text{heap}}} \text{dom}(r) \right)$$

In addition to the above invariants, there is a restriction stating that heap attribute labels must conform to the their object's static type. Also, attribute labels on the `Current` location cannot have the same name as any of the formal or locals.

Using these invariants, we define V_{legal} to be the set of all legal concrete states, thus:

$$V_{\text{legal}} \subset V \qquad \text{and} \qquad \forall v \in V_{\text{legal}} \colon (\forall i \in I_{\text{concrete}} \colon i(v))$$

Remember that these invariants only guarantee the consistency of the program state. It is up to the semantics to make sure that the program text transforms this state in the correct manner, while keeping it consistent.

3.1.1 Viewpoint-Adapted Concrete Domains

The concrete domain as defined so far is still too big. Nothing is known about the number and names of formals. Since a concrete state is always relative to a given routine's viewpoint, we further divide V_{legal} into subsets that conform to routine signatures (i.e. a sequence of formal argument names and corresponding types, and – if present – a result type).

Routine is defined as follows:

$$\begin{aligned} (\mathit{name}, \mathit{signature}) &\in \mathbf{Routine} := \mathbf{Lab} \times \mathbf{Signature} \\ (\mathit{formals}, \mathit{returntype}) &\in \mathbf{Signature} := (\mathbf{Lab} \times \mathbf{Type})^* \times (\mathbf{Type} \cup \text{NONE}) \\ (\mathit{classname}, \mathit{is\_separate}) &\in \mathbf{Type} := \mathbf{Lab} \times \mathbb{B} \end{aligned}$$

NONE stands for *no type*, and is used for routines that have no return value – also referred to as *procedures*.

We write V_{rout} to denote only the concrete states that conform to routine $\mathit{rout} \in \mathbf{Routine}$. We call these sets *viewpoint-adapted concrete domains*.

Definition 3.1.1 (viewpoint-adapted concrete domains)**.**

$$\begin{aligned} &\forall \mathit{rout} \in \mathbf{Routine} \colon \\ &\quad V_{\text{legal}} \supset V_{\text{rout}} := \{v \in V_{\text{legal}} \mid \text{dom}(v_{\text{formals}}) = \{l \mid \exists t \colon (l,t) \in (\mathit{rout}_{\text{signature}})_{\text{formals}}\}\} \end{aligned}$$

3.2 Abstract Domain

The *abstract domain* approximates the concrete domain in such a way that interesting properties about the topology of the processor regions are maintained, and that a static analysis actually terminates on both an inter- and intra-procedural level. An element of the abstract domain is called a *state graph*, since it represents a state and can be visualized as a graph. Figure 3.1 shows an example of such a state graph.

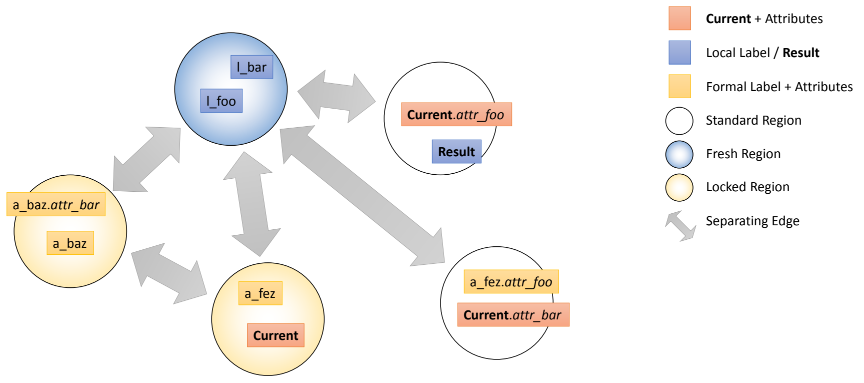


Figure 3.1: *visualization of a state graph*

Just as concrete states, state graphs must be interpreted from the viewpoint of a given routine. The following list describes the components of a state graph.

Abstract regions An abstract region represents a concrete processor region. However, two or more abstract regions may represent the *same* concrete processor region. A state graph consists of at least one such abstract region, and an abstract region only exists if at least one label is attached to it. Abstract regions can be marked with up to three different flags.

Flag *locked* An abstract region is locked, if the processor executing the target routine holds at least one lock (request queue lock or call stack lock) associated with the concrete region's handler. Note that this flag is a *must* flag, i.e. when it is present, we know that the processor holds a lock, and otherwise it may or may not hold one.

Flag *fresh* An abstract region is fresh, if the associated concrete region has been created in either the target routine, or any other routine higher in the call chain. This flag is a *must* flag, so it is possible that some actual fresh concrete regions are not marked as such in the abstract counterpart.

Flag *leaked* An abstract region is leaked, if it is fresh and the associated concrete region is reachable (or rather, a location inside it) via the heap (that is, with at least one attribute lookup, or directly via formals or locals). This flag is a *may* flag.

Labels The labels of a state graph closely resemble variable names in the concrete program. There are labels for formals, locals, `Current`, `Result`, and unqualified and qualified attributes. Every label exists at most once per state graph and is attached to some abstract region. The meaning of this is, that the variable that is represented by the label stores a reference to *some location* on the processor region represented by the label's abstract region. There are three types of labels:

Controlled labels stand for formal routine argument names, as well as the constant `Current`.

Attribute labels stand for attribute names of the objects represented by the controlled labels. Therefore, an attribute label is only defined together with an existing controlled label.

Local labels stand for local variable names present in the target routine.

Note that controlled labels cannot change the abstract region they are attached to, since in Eiffel, formal routine arguments are not writable. Attribute labels can only change their abstract region when they are on `Current` (i.e. unqualified in the target routine). However, the other attribute labels may get changed over time via routine calls.

Separating edges Abstract regions may be connected via undirected edges called *separating edges*. Two separated abstract regions are known to represent distinct concrete processor regions. Labels attached to one such abstract region cannot possibly be aliased by labels attached to the other abstract region.

Evaluation stack Since Eiffel expressions can be arbitrarily nested, we need to keep track of intermediate, temporary results that are not represented by named variables in the program, but are rather stored directly on the call stack. A state graph therefore maintains an *evaluation stack* whose elements are a special kind of label as well: Such a label is denoted with the number of its position on the call stack (the label with the highest number is at the top of the stack), and attached to the abstract region it points to. If the evaluation stack contains an alias of some of the controlled labels, the evaluation stack label consists not only of its position on the stack, but also of the controlled label it represents. This allows us to keep track of the actual call target's attributes of separate routine calls.

3.2.1 Motivation and Justification

Having introduced the abstract domain's elements – the state graphs – in detail, we now examine the underlying design decisions. There are three important requirements that must be fulfilled by the abstract domain:

1. It must be possible to describe the set of all valid concrete program states *soundly* with a finite number of state graphs.
2. The abstract domain must form a *complete lattice*, that is, given two arbitrary state graphs, there must exist some state graph that *best* describes all concrete states that are described by both other state graphs together – and possibly more.
3. The state graphs must allow us to answer the *questions* we are interested in as precisely as possible.

The first requirement is satisfied by the flexible notion of an abstract region. When we do not know anything about the location of a particular label, we attach it to some abstract region that is not separated to any other abstract region. This way, the abstract region could stand for a unique concrete region, or it may also stand for any concrete region also represented by other abstract regions in the state graph. When two labels are attached to the same abstract region, then we know that in the concrete state, they must point to locations on the same concrete region. However, we do not have to know *which* concrete region this is. Since we are interested in only a fixed number of labels, we can express every possible concrete state with a finite amount of abstract regions and labels.

The second requirement is fulfilled as well, since we are able to specify a *least upper bound operation* on any two state graphs. We will give this definition in section 4.4.1 of the next chapter.

The last requirement cannot be shown as clearly as the first two. There are just arguments for and against it. The abstract regions' flags play the central role in classifying routine calls – one of the ultimate goals of the whole analysis. Recall the main questions about separate routine calls the analysis tries to answer:

- Does a separate routine call trigger lock-passing?
- Is a separate routine call synchronous or asynchronous?

The analysis cannot give precise answers in every situation, since it is a static analysis run on the source code in a finite and reasonable amount of time. The goal is to be as precise as possible without sacrificing too much performance. We first analyze what properties of the concrete state are relevant for answering the above questions. Note that in the concrete state, it is indeed possible to answer them precisely, but at the expense of very high performance costs that are not much lower than actually executing the program.

Lock-passing occurs while calling a routine on an object that is located on a different concrete region than the calling routine's current object, and the calling processor holds *some* lock associated with the processor region of *any* of the actual routine arguments passed on to the callee routine.
The important kind of information therefore is, what locks the processor executing the target routine currently holds. This information is captured by the *locked* flag of some abstract region.

In order to answer the second question, we have to determine what questions influence the asynchrony of a separate routine call. There are three observations:

- The region of the call target has to be separated from the region of the currently executing processor, in order for the call to potentially be asynchronous.
- Lock-passing causes a call to be synchronous, not matter what.
- Function calls are always synchronous, since the caller necessarily needs to wait for the result.

Telling if a routine call is a function call is trivially given by the called routine's definition. Lock-passing has been described already, and telling regions apart is done via separating edges in the state graph.

A Word about the Heap

The choice of just including one level of attributes, and the reason why they have to be attached to controlled labels, are motivated by the SCOOP model itself.

SCOOP guarantees that inside a given routine body, the executing processor has exclusive access to all of the routine's formal argument's regions (called *formal regions* in the following). No other processor can influence the state as seen and modified by this routine, although it is possible that separate call messages (denoted by *foreign messages* in the following) are enqueued on the formal regions by other processors. But these messages can

Listing 3.1: Eiffel*: asynchronous calls synchronously interpreted*

```
class APPLICATION
feature
  ...
  two_async_calls (a_worker, another_worker: separate WORKER)
    local
      l_worker: separate WORKER
    do
      a_worker.work
      another_worker.work
      l_worker := a_worker
    end
  ..
end

class WORKER
feature
  ...
  progress: PROGRESS
  ...
  work
    do
      -- Assign something to 'progress'
      progress := ...
    end
  ...
end
```

only be enqueued on the private request queues corresponding to the issuing processors. Because of the strict ordering of private request queues on each formal region, we can be sure that either all foreign calls are applied before all the calls issued by the target routine, or after them, but not in-between. See the description about the queue of queue model in section 2.2.6 for more details.

This explains why only attributes of controlled labels are captured in a state graph. Everything else can be interrupted by some foreign processor at any time, and therefore, it is not safe to assume anything about it.

The region analysis interprets the source code in a completely synchronous fashion, i.e. whenever a routine call is encountered, it is followed and the results of the interpreted callee are used in order to continue the interpretation right after the call. This behavior is strictly respected, even in the case where the routine call may be asynchronous at runtime. In chapter 4, we describe this scheme in more detail.

Assume that the two formal arguments in the SCOOP routine `two_async_calls` of figure 3.1 are actually located on separate regions. Then, the first two statements will call two asynchronously executed routines, and the last statement – which will *not wait* for the `work` calls to finish – will assign one of the formal arguments to a local variable. Let us further assume that each worker has exactly one attribute (here: `progress`) that gets assigned to by `work`.

The state graph at the end of the interpretation of `two_async_calls` may reflect a situation that does not exist at a single point in time, but rather as a combination/blend of several points in time in the future. In particular, the state graph contains the `progress` attributes of both workers, just as if their `work` routine already finished, and at the same time, it contains the local label `l_worker`. In reality, this situation may never occur, because both `work` routines may start executing long after `two_async_calls` has returned.

However, this seemingly strange situation is still valid, at least from the viewpoint of the `two_async_calls` routine: The possible real-time sequence of actions does not matter to this routine, as long as it is executed sequentially consistent to the calls as issued (in the program text) to each formal argument in isolation. This kind of reasoning is exactly what was intended by SCOOP to begin with. And it is perfectly respected by the region analysis.

3.2.2 Formalization of State Graphs

Just as in the concrete state, a central element in a state graph is a *label*. Here, they are classified a bit differently. Labels represent variable names, i.e. formals and `Current` (together referred to as *controlled labels*), locals and `Result` (together referred to as *locals*), qualified and unqualified attributes. Therefore, a legal label must again obey the Eiffel rules of legal entity names, which are described by using regular expression syntax:

$$\begin{aligned} \mathbf{Lab} &:= \text{[A-Za-z][A-Za-z0-9\_]}^* \\ (\mathit{target}, \mathit{attribute}) \in \mathbf{QLab} &:= \mathbf{Lab} \times \mathbf{Lab} \end{aligned}$$

Note that **QLab** denotes a *qualified label*, that is, a 2-tuple of labels, which corresponds to a qualified attribute in Eiffel.

The *evaluation stack* can contain two kinds of elements: Controlled labels and anonymous regions. This difference is important, since controlled labels do not only specify the region they they are attached to, but also the actual object they refer to – which is important for keeping track of attributes –, whereas an anonymous region is just the abstract region an intermediate evaluation result is attached to.

$$(\mathit{idx}, \mathit{eval}) \in \mathbf{Eval} := \mathbb{N} \times (\mathbf{Lab} \cup \epsilon)$$

Eval contains the *elements* of the evaluation stack. The stack itself is modeled implicitly by idx, which denotes an element's position on the evaluation stack. The top element of the stack has the highest index of all stack elements present in a state graph. The special value ϵ is used for anonymous regions.

Abstract regions are the main component of a state graph. An abstract region's identity is defined entirely by the labels and/or evaluation stack elements that are attached to it. This is in contrast to a region's identity in the implementation, as described in section 5.4.1.

$$(\mathit{ctrls}, \mathit{attrs}, \mathit{locals}, \mathit{evals}) \in \mathbf{AReg} := \mathcal{P}(\mathbf{Lab}) \times \mathcal{P}(\mathbf{QLab}) \times \mathcal{P}(\mathbf{Lab}) \times \mathcal{P}(\mathbf{Eval})$$

Regions have internal properties as well. They can be *locked*, *fresh* or *leaked*.

$$\mathbf{Props} := \mathbf{AReg} \to \mathcal{P}(\{\mathcal{K}, \mathcal{F}, \mathcal{L}\})$$

The three constants on the right stand for the three properties in the order they are written above.

Invariant 3.2.1 (only fresh regions can leak)**.**

$$\forall p \in \mathbf{Props} \colon \forall e \in \mathrm{im}(p) \colon \mathcal{L} \in e \implies \mathcal{F} \in e$$

Separating edges are undirected and connect two abstract regions.

$$\mathbf{Sep} := \{\{r_1, r_2\} \mid r_1, r_2 \in \mathbf{AReg} \wedge r_1 \neq r_2\}$$

Regions can also be *representatives* of other regions.

$$\mathbf{Rep} := \mathbf{AReg} \to \mathcal{P}(\mathbf{AReg})$$

The regions in the map's domain are considered representatives of the regions in the map's image. We explain the motivation behind representatives in section 4.5.2.

Finally, we define the abstract domain A, the set of state graphs.

$$(\mathit{regs}, \mathit{sep}, \mathit{props}, \mathit{rep}) \in A := \mathcal{P}(\mathbf{AReg}) \times \mathcal{P}(\mathbf{Sep}) \times \mathbf{Props} \times \mathbf{Rep}$$

Queries over State Graphs

The set A, as defined so far, still contains illegal elements, much as was the case with the concrete domain as well. Thus, we formulate several invariants that restrict A to only the legal state graphs.

In order to simplify the formulation of these invariants, we define some auxiliary functions. All of these functions have a subscript a, which denotes the state graph they operate on.

Definition 3.2.1 (gather all controlled labels, local labels, and attributes labels).

$$\begin{aligned}
\mathit{controlled}\colon & A \to \mathcal{P}(\mathbf{Lab}) \\
\mathit{locals}\colon & A \to \mathcal{P}(\mathbf{Lab}) \\
\mathit{attributesOf}\colon & A \times \mathbf{Lab} \to \mathcal{P}(\mathbf{QLab}) \\
\mathit{allAttributes}\colon & A \to \mathcal{P}(\mathbf{QLab}) \\
\mathit{labels}\colon & A \to \mathcal{P}(\mathbf{Lab} \cup \mathbf{QLab})
\end{aligned}$$

$$\begin{aligned}
\mathit{controlled}_a &:= \bigcup_{r \in a_{\text{regs}}} r_{\text{ctrls}} \\
\mathit{locals}_a &:= \bigcup_{r \in a_{\text{regs}}} r_{\text{locals}} \\
\mathit{attributesOf}_a(c) &:= \bigcup_{r \in a_{\text{regs}}} \{l \mid l \in r_{\text{attrs}} \land l_{\text{target}} = c\} \\
\mathit{allAttributes}_a &:= \bigcup_{r \in a_{\text{regs}}} r_{\text{attrs}} \\
\mathit{labels}_a &:= \mathit{controlled}_a \cup \mathit{locals}_a \cup \mathit{allAttributes}_a
\end{aligned}$$

Definition 3.2.2 (query properties of regions).

$$\begin{aligned}
\mathit{locked}\colon & A \times \mathbf{AReg} \to \mathbb{B} \\
\mathit{fresh}\colon & A \times \mathbf{AReg} \to \mathbb{B} \\
\mathit{leaked}\colon & A \times \mathbf{AReg} \to \mathbb{B} \\
\mathit{separate}\colon & A \times \mathbf{AReg} \times \mathbf{AReg} \to \mathbb{B}
\end{aligned}$$

$$\begin{aligned}
\mathit{locked}_a(r) &:= \mathcal{K} \in a_{\text{props}}(r) \\
\mathit{fresh}_a(r) &:= \mathcal{F} \in a_{\text{props}}(r) \\
\mathit{leaked}_a(r) &:= \mathcal{L} \in a_{\text{props}}(r) \\
\mathit{separate}_a(r_1, r_2) &:= \{r_1, r_2\} \in a_{\text{sep}}
\end{aligned}$$

Definition 3.2.3 (get representatives, and check for representation).

$$\begin{aligned}
\mathit{reps}\colon & A \times \mathbf{AReg} \to \mathcal{P}(\mathbf{AReg}) \\
\mathit{isRepBy}\colon & A \times \mathbf{AReg} \times \mathbf{AReg} \to \mathbb{B}
\end{aligned}$$

$$\begin{aligned}
\mathit{reps}_a(r) &:= a_{\text{rep}}(r) \\
\mathit{isRepBy}_a(r_1, r_2) &:= r_1 \in a_{\text{rep}}(r_2)
\end{aligned}$$

Definition 3.2.4 (query a region of a given label).

$$\begin{aligned}
\mathit{regOf}\colon & A \times \mathbf{Lab} \cup \mathbf{QLab} \to \mathbf{AReg} \\
\mathit{regOf}_a(l) &:= \{r \mid r \in a_{\text{regs}} \land l \in r_{\text{locals}} \cup r_{\text{ctrls}} \cup r_{\text{attr}}\}
\end{aligned}$$

Invariant 3.2.2 (exactly one region per label).

$$\forall l \in \mathit{labels}_a \colon \left|\mathit{regOf}_a(l)\right| = 1$$

Definition 3.2.5 (extract the evaluation stack as a sequence).

$$\begin{aligned} \mathit{evalStack}\colon A &\to (\mathbf{Eval})^* \\ \mathit{evalStack}_a &\mapsto (e)_i, \quad \text{where} \quad (i,e) \in \bigcup_{r \in a_{\text{regs}}} r_{\text{evals}} \end{aligned}$$

The above definition is only legal if the indexes of all evaluation elements in the state graph form an ascending chain starting at 1, stated by the next invariant.

Invariant 3.2.3 (valid evaluation stack).

$$\forall a \in A, s = \{i \mid (i,e) \in \cup_{r \in a_{\text{regs}}} r_{\text{evals}}\}\colon |s| = \max(s)$$

Definition 3.2.6 (query the top of the evaluation stack).

$$\begin{aligned} \mathit{top}\colon A &\to \mathbf{Eval} \\ \mathit{top}_a &\mapsto (\mathit{evalStack}_a)_{|\mathit{evalStack}_a|} \end{aligned}$$

Note that top, like other queries as well, can be *undefined*. In the case where the evaluation stack is empty, this query is undefined.

Legal State Graphs

While introducing the definitions above, we have already stated some *invariants* that must be fulfilled in order for a state graph to be *legal*. In the following, more invariants will be stated. Every such invariant acts as a boolean function on some $a \in A$. Together, we denote the set of all invariants with I_{abstract}.

Invariant 3.2.4 (no local label is identical to any controlled label).

$$\mathit{controlled}_a \cap \mathit{locals}_a = \emptyset$$

Invariant 3.2.5 (attributes are only defined on controlled labels, not on local labels).

$$\forall (c,t) \in \mathit{allAttributes}_a\colon c \in \mathit{controlled}_a$$

Invariant 3.2.6 (`Current` is always a controlled label).

$$\texttt{Current} \in \mathit{controlled}_a$$

Invariant 3.2.7 (regions that have a controlled label attached are always locked).

$$\forall c \in \mathit{controlled}_a\colon \mathit{locked}_a(\mathit{regOf}_a(c))$$

Invariant 3.2.8 (the labels on the evaluation stack must be controlled).

$$\forall e \in \mathit{evalStack}_a\colon e_{\text{eval}} \in \mathbf{Lab} \implies e_{\text{eval}} \in \mathit{controlled}_a$$

Invariant 3.2.9 (legal separation graph).

$$\forall s \in a_{\text{sep}}\colon s \subseteq a_{\text{regs}}$$

Invariant 3.2.10 (regions only represent regions that are identified with labels from the state graph).

$$\forall r \in \bigcup_{r' \in a_{\text{regs}}} \mathit{reps}_a(r') \colon r_{\text{ctrls}} \subseteq \mathit{controlled}_a \\ \wedge r_{\text{locals}} \cup r_{\text{evals}} = \emptyset \wedge r_{\text{attrs}} \subseteq \mathit{allAttributes}_a$$

Since our model does not incorporate full static type information, we have to state the final invariant informally: Similar to the concrete domain, attribute labels of controlled labels must exist in the controlled label's associated class, and the attribute labels of `Current` cannot have the same name as the controlled labels or local labels.

Using these invariants, we define A_{legal} to be the set of all legal abstract states, thus:

$$A_{\text{legal}} \subset A \qquad \text{and} \qquad \forall a \in A_{\text{legal}} \colon (\forall i \in I_{\text{abstract}} \colon i(a))$$

3.2.3 Viewpoint-Adapted Abstract Domains

Just as we have divided the concrete domain into subsets of routine-conforming elements, we do the same with the abstract domain: A_{rout} denotes only the subset of the abstract domain whose elements conform to routine rout. We call these domains *viewpoint-adapted abstract domains*.

Definition 3.2.7 (viewpoint-adapted abstract domains).

$$\forall \mathit{rout} \in \mathbf{Routine} \colon \\ A_{\text{legal}} \supset A_{\text{rout}} := \{a \in A_{\text{legal}} \mid \mathit{controlled}_a = \{l \mid \exists t \colon (l,t) \in (\mathit{rout}_{\text{signature}})_{\text{formals}}\}\}$$

3.3 Representation Function

The abstract domain has a notion of region that is not equivalent to that of the concrete domain. In the latter, *different* concrete regions are also different in the real program state. However, in the abstract domain, different abstract regions may represent the *same* concrete region, unless they are pairwise separated via a separating edge in the state graph.

In the abstract domain, local labels and attribute labels point to regions, whereas in the concrete domain, they point to locations (objects) that are part of a region. However, one exception of this rule are the controlled labels of a state graph: They also point to actual objects. Otherwise, attribute labels would not be meaningful.

The following table outlines how individual aspects of a concrete state can be translated to aspects of an abstract state.

Aspect	**Translation from concrete to abstract domain**
Labels and *Regions*	In the concrete state, we can for each concrete region gather all locations that are directly known to formals, locals or elements on the stack, or indirectly (i.e. via one attribute lookup in the heap) to formals. Once we have identified all of these labels respectively stack indexes, we have a unique description of the corresponding abstract region in the abstract domain.
Freshness	Given a concrete state, it is not possible to determine whether or not a concrete region is fresh, since freshness is a concept that requires more information than what is contained in a concrete state. Therefore, no abstract region gets marked as fresh by the representation function.
Locks	In the concrete state, locks correspond to concrete regions. In the abstract state, we translate these concrete regions to their abstract counterparts. If this translation succeeds (i.e. we actually know about the abstract region), we mark the abstract region as locked.

The representation function takes an element of the concrete domain, and returns the best matching element in the abstract domain, that is, the state graph that *most precisely* describes the given concrete state.

The representation function, β_{rout}, is declared as follows:

$$\beta_{rout} : V_{rout} \rightarrow A_{rout}$$

where V_{rout} denotes a *viewpoint-adapted concrete domain* and A_{rout} a *viewpoint-adapted abstract domain*, for some $rout \in \mathbf{Routine}$. The representation function therefore only makes sense between domains associated to the same routine.

Our next goal is to define $a \in A_{rout}$ in terms of $v \in V_{rout}$, one aspect at a time. But before that, we introduce some convenience queries that allow us to formalize the representation function much more concisely.

In some situations, we have singleton sets (i.e. sets containing only one element) as a result of some queries. In order to extract the element from such a set, we use a designated function:

Definition 3.3.1 (extracting an element from a singleton set)**.**

$$\begin{aligned} \mathit{extract}\colon \mathcal{P}_1(\Omega) &\to \Omega \\ \mathit{extract}(\{x\}) &:= x \end{aligned}$$

The set Ω denotes an arbitrary set.

Definition 3.3.2 (reachable concrete regions)**.**

$$\begin{aligned} \mathit{regOfLoc}\colon V \times \mathbf{Loc} &\to \mathbf{Reg} \\ \mathit{regOfLoc}_v(l) &\mapsto \mathit{extract}(\{r \mid r \in v_{\text{heap}} \land l \in \operatorname{dom}(r)\}) \end{aligned}$$

$$\begin{aligned} \mathit{reachableRegions}\colon V &\to \mathcal{P}(\mathbf{Reg}) \\ \mathit{reachableRegions}_v &\mapsto \\ &\{r \mid r \in v_{\text{heap}} \land \exists l \in \operatorname{im}(v_{\text{formals}}) \cup \operatorname{im}(v_{\text{locals}}) \cup v_{\text{stack}}\colon l \in \operatorname{dom}(r)\} \\ &\cup \\ &\left\{ \mathit{regOfLoc}_v(l') \mid l' \neq \texttt{Void} \land l' \in \bigcup_{l \in \operatorname{im}(v_{\text{formals}})} \operatorname{im}(\mathit{regOfLoc}_v(l)(l)) \right\} \end{aligned}$$

$$\begin{aligned} \mathit{buildEvalOf}\colon V \times \mathbf{Loc} &\to \mathbf{Lab} \cup \epsilon \\ \mathit{buildEvalOf}_v(l) &:= \begin{cases} \min(\{c \mid v_{\text{formals}}(c) = l\}) & \text{if } \exists c\colon v_{\text{formals}}(c) = l \\ \epsilon & \text{otherwise} \end{cases} \end{aligned}$$

Reachable concrete regions are the ones the target routine knows about with the *finite horizon* constraint that it can at most follow one attribute access. Note that in $\mathit{buildEvalOf}$, we use the minimal controlled label (in the lexical order sense), in case there is more than one formal label pointing to the given location. This way, the function remains deterministic.

Definition 3.3.3 (define abstract regions from concrete ones)**.**

$$\begin{aligned} \mathit{abstractRegion}\colon V \times \mathbf{Reg} &\to \mathbf{AReg} \\ \mathit{abstractRegion}_v(r) &\mapsto (\\ &\{l \mid v_{\text{formals}}(l) \in \operatorname{dom}(r)\}, \\ &\{(l_t, l_a) \mid \mathit{regOfLoc}_v\left(\left(\left(\mathit{regOfLoc}_v(v_{\text{formals}}(l_t))\right)\left(v_{\text{formals}}(l_t)\right)\right)(l_a)\right) = r\}, \\ &\{l \mid v_{\text{locals}}(l) \in \operatorname{dom}(r)\}, \\ &\{(i, e) \mid e = \mathit{buildEvalOf}_v((v_{\text{stack}})_i)\}) \end{aligned}$$

The representation function is defined via these queries.

Definition 3.3.4 (definition of the representation function)**.**

$$\beta_{rout}(v) \mapsto (\mathit{regs}, \mathit{sep}, \mathit{props}, \mathit{rep}),$$

$$\begin{aligned}
&\text{where}\\
&\quad \mathit{regs} = \{\mathit{abstractRegion}_v(r) \mid r \in \mathit{reachableRegions}_v\}\\
&\quad \mathit{sep} = \{\{r_1, r_2\} \mid r_1 \neq r_2\}\\
&\quad \mathit{props} = \{(\mathit{abstractRegion}_v(r), \{\mathcal{K}\}) \mid r \in v_{\text{locks}} \cap \mathit{reachableRegions}_v\}\\
&\quad \mathit{rep} = \emptyset
\end{aligned}$$

Using the representation function, it is easy to infer the concretization and abstraction functions.

Definition 3.3.5 (definition of the concretization and abstraction function)**.**

$$\begin{aligned}
\gamma_{rout} &: A_{rout} \to \mathcal{P}(V_{rout})\\
\gamma_{rout}(a) &\mapsto \{v \mid \beta_{rout}(v) \sqsubseteq a\}
\end{aligned}$$

$$\begin{aligned}
\alpha_{rout} &: \mathcal{P}(V_{rout}) \to A_{rout}\\
\alpha_{rout}(V) &\mapsto \{\beta_{rout}(v) \mid v \in V\}
\end{aligned}$$

Note that the concretization function is not computable, because there are infinitely many concrete states represented by an abstract state. However, both α_{rout} and γ_{rout} form a *Galois insertion*[1], which we are not going to proof in this thesis.

[1]The Galois insertion fixes the shortcoming of a Galois connection, where there may be several elements of A_{rout} that *best* describe the same element in V_{rout}.

Chapter 4

Abstract Interpretation Framework

In this chapter, we describe the inter-procedural abstract interpretation framework, both formally and informally.

4.1 Overview

The concrete and abstract domains introduced in chapter 3 are both view-point-adapted to a given target routine. The representation function establishes the link between concrete states and abstract states. However, the overall framework operates inter-procedurally, so it is not enough to limit our viewpoint to only a single routine. Instead, we need to find ways of transitioning between different viewpoints.

The region analysis is referred to by the term *context-sensitive analysis* in the literature [6]. This means that we include a *context*. Without contexts, we would not be able to distinguish different calls to the same routine, and as a consequence, we would have to analyze a given routine in a more generic way. The addition of contextual information allows us to differentiate between calls, and therefore tremendously improve the precision, at the expense of additional iterations in the solving process.

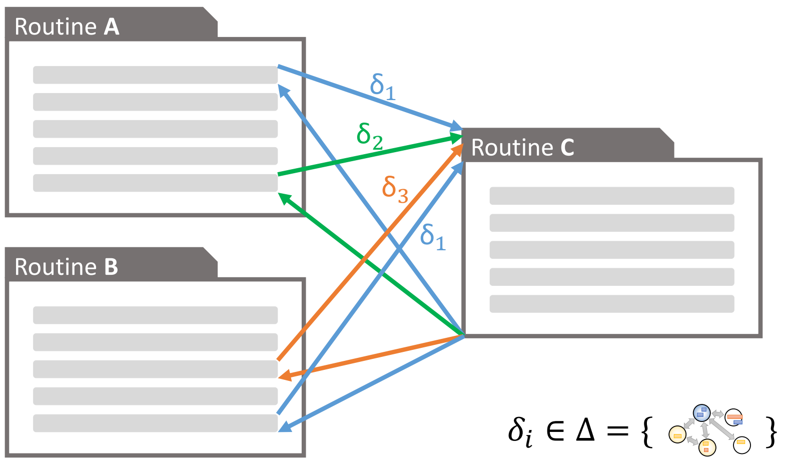


Figure 4.1: *different calling contexts*

Figure 4.1 schematically demonstrates what a context is. The colored arrows stand for different calling contexts, which are special kinds of state graphs carrying information about the state that was established by the calling routine.
In the figure, routine **A** has two calls to routine **C** in its body, but both calls are issued under different conditions – or contexts. The analysis therefore interprets routine **C** once for every context. However, routine **B** issues a call that has the same context as the call from routine **A**, so in this case, the same result can be used by routine **B**, and no additional interpretations of routine **C** are necessary.

4.2 Default Contexts

Since the region analysis is usually started on the root routine of the SCOOP system, no context is yet available. However, the type system already provides enough information to construct a default context for each routine in the system, not just for the root routine.

The default context is obtained by examining the static declaration of a given routine's signature. We create new abstract regions for every formal argument that is declared as `separate`. Then, we distribute the controlled labels that correspond to formals over them. Non-separate controlled labels are attached to the same abstract region as the controlled label `Current`. However, no attribute labels are inserted, as the shape of the heap can not be derived from the type system.

4.3 Embellished Monotone Framework

We are going the specify the region analysis as an *embellished monotone framework* (the full theory can be found in [6]), which consists of the following:

- An abstract domain that forms a *complete lattice* which satisfies the ascending chain condition[1]
- A set of *transfer functions* which transfer an element of the abstract domain and a program fragment into another element of the abstract domain.
- A labeled *finite flow* (i.e. the flow-graph of the program text).
- *Extremal labels* for the beginning and end of the flow.
- An *extremal element* of the abstract domain to start the analysis with.
- A *mapping* that associates labels in the flow (i.e. program fragments) to the corresponding transfer functions.
- A *context* domain that is paired with the abstract domain to enable a context-sensitive analysis.

The ascending chain condition is trivially fulfilled by the fact that our abstract domain is finite. The construction of the flow is straightforward: We decompose the program text into its fragments, and connect them according to the source code structure. The fragments are fine-grained on the level of single expressions and statements, even if they are nested. The program labels denote the entry and exit points of a given program fragment like an

[1]Every strictly ascending sequence (in terms of the less-than operator) of elements eventually terminates.

assignment or a routine call. The extremal labels denote the entry and exit points of the root routine. The extremal elements of the abstract domain are the default context state graphs. What remains to be specified are the complete lattice properties and the transfer functions of the analysis. The mapping thereof to the corresponding program labels is also straightforwardly given by a transfer function's purpose.

4.4 Complete Lattice

The elements of the abstract domain – the state graphs – form a finite and complete lattice, but only relative to a given written routine, that is, number, order, and names of formals must match, and all existing attribute labels must indeed exist in the corresponding class definitions.

$$A_{rout} = (A_{rout}, \sqsubseteq, \sqcup, \sqcap, , \top) \qquad \forall\, rout \in \mathbf{Routine}$$

where $A_{rout} \subset A$ is a viewpoint-adapted abstract domain corresponding to $rout \in \mathbf{Routine}$, $\sqsubseteq$ is the less-or-equal operator, $\sqcup$ is the least upper bound operator, and $\sqcap$ is the greatest lower bound operator. $\bot = \sqcup\emptyset = \sqcap A_{rout}$ is the least element, and $\top = \sqcap\emptyset = \sqcup A_{rout}$ the greatest element.

We show that A_{rout} is indeed a complete lattice by defining a *binary* least upper bound $\sqcup$ operator. However, the operator can also be applied to any subset of A_{rout}, by successively reducing pairs of elements, until only one is left, which is then the least upper bound of the whole subset. Using $\sqcup$, we can infer $\sqcap$ and $\sqsubseteq$.

4.4.1 The Join Operator

The binary least upper bound operator – or *join* operator – takes two state graphs that are associated to the same routine, and returns a new state graph which is their best possible common representation.

$$\sqcup\colon A_{rout} \times A_{rout} \to A_{rout} \qquad \forall\, rout \in \mathbf{Routine}$$

Note that $\sqcup$ is only defined on pairs of state graphs belonging to the same viewpoint-adapted abstract domain and that its result also belongs to this domain.

In order to easily specify $\sqcup$, we introduce three types of binary relations between labels of a state graph.

Definition 4.4.1 (binary boolean relations on labels)**.**

$$\overset{\mathrm{R}}{=\!=\!=} \subseteq A_{rout} \times (\mathbf{Lab} \cup \mathbf{QLab}) \times (\mathbf{Lab} \cup \mathbf{QLab})$$
$$\overset{\mathrm{R}}{\Longleftrightarrow} \subseteq A_{rout} \times (\mathbf{Lab} \cup \mathbf{QLab}) \times (\mathbf{Lab} \cup \mathbf{QLab})$$
$$\xleftrightarrow{\mathrm{R}} \subseteq A_{rout} \times (\mathbf{Lab} \cup \mathbf{QLab}) \times (\mathbf{Lab} \cup \mathbf{QLab})$$

$\forall l, l' \in allLabels_a :$

$$\begin{aligned}
l \overset{\mathrm{R}}{=\!=\!=}_a l' \quad &\Longleftrightarrow \quad regOf_a(l) = regOf_a(l') \\
l \overset{\mathrm{R}}{\Longleftrightarrow}_a l' \quad &\Longleftrightarrow \quad separate_a(regOf_a(l), regOf_a(l')) \\
l \xleftrightarrow{\mathrm{R}}_a l' \quad &\Longleftrightarrow \quad \neg separate_a(regOf_a(l), regOf_a(l')) \wedge regOf_a(l) \neq regOf_a(l')
\end{aligned}$$

In a simplified form, table 4.1 shows how two labels relate in the joined state graph (entries inside the table), depending on their relation in the two operand state graphs (upper row and left column). This table's information is also present in the definition of the join operator, although in a more formal form.

	$\xleftrightarrow{\mathrm{R}}$	$\overset{\mathrm{R}}{\Longleftrightarrow}$	$\overset{\mathrm{R}}{=\!=\!=}$
$\xleftrightarrow{\mathrm{R}}$	$\xleftrightarrow{\mathrm{R}}$	$\xleftrightarrow{\mathrm{R}}$	$\xleftrightarrow{\mathrm{R}}$
$\overset{\mathrm{R}}{\Longleftrightarrow}$		$\overset{\mathrm{R}}{\Longleftrightarrow}$	$\xleftrightarrow{\mathrm{R}}$
$\overset{\mathrm{R}}{=\!=\!=}$			$\overset{\mathrm{R}}{=\!=\!=}$

Table 4.1: *Separation combination (symmetric)*

Using these binary relations, we define the *join* operator:

Definition 4.4.2 (the *join* operator)**.**

$$\forall a_l, a_r \in A_{rout}\colon \quad a_l \sqcup a_r := a_j \quad \text{such that}$$

$$a_j \in A_{rout} \wedge \forall l, l' \in \mathit{allLabels}_{a_l} \cup \mathit{allLabels}_{a_r}, l \neq l'\colon$$

$$\begin{aligned}
l \overset{\text{R}}{\longleftrightarrow}_{a_l} l' \quad &\wedge \quad l \overset{\text{R}}{\longleftrightarrow}_{a_r} l' \quad &\Longrightarrow \quad l \overset{\text{R}}{\longleftrightarrow}_{a_j} l' \\
l \overset{\text{R}}{\longleftrightarrow}_{a_l} l' \quad &\wedge \quad l \overset{\text{R}}{\Longleftrightarrow}_{a_r} l' \quad &\Longrightarrow \quad l \overset{\text{R}}{\longleftrightarrow}_{a_j} l' \\
l \overset{\text{R}}{\longleftrightarrow}_{a_l} l' \quad &\wedge \quad l \overset{\text{R}}{=\!=}_{a_r} l' \quad &\Longrightarrow \quad l \overset{\text{R}}{\longleftrightarrow}_{a_j} l' \\
l \overset{\text{R}}{\Longleftrightarrow}_{a_l} l' \quad &\wedge \quad l \overset{\text{R}}{\Longleftrightarrow}_{a_r} l' \quad &\Longrightarrow \quad l \overset{\text{R}}{\Longleftrightarrow}_{a_j} l' \\
l \overset{\text{R}}{\Longleftrightarrow}_{a_l} l' \quad &\wedge \quad l \overset{\text{R}}{=\!=}_{a_r} l' \quad &\Longrightarrow \quad l \overset{\text{R}}{\longleftrightarrow}_{a_j} l' \\
l \overset{\text{R}}{=\!=}_{a_l} l' \quad &\wedge \quad l \overset{\text{R}}{=\!=}_{a_r} l' \quad &\Longrightarrow \quad l \overset{\text{R}}{=\!=}_{a_j} l'
\end{aligned}$$

$$\begin{aligned}
\mathit{locked}_{a_l}(\mathit{regOf}_{a_l}(l)) \wedge \mathit{locked}_{a_r}(\mathit{regOf}_{a_r}(l)) \quad &\Longleftrightarrow \quad \mathit{locked}_{a_j}(\mathit{regOf}_{a_j}(l)) \\
\mathit{fresh}_{a_l}(\mathit{regOf}_{a_l}(l)) \wedge \mathit{fresh}_{a_r}(\mathit{regOf}_{a_r}(l)) \quad &\Longleftrightarrow \quad \mathit{fresh}_{a_j}(\mathit{regOf}_{a_j}(l)) \\
\mathit{leaked}_{a_l}(\mathit{regOf}_{a_l}(l)) \vee \mathit{leaked}_{a_r}(\mathit{regOf}_{a_r}(l)) \quad &\Longleftrightarrow \quad \mathit{leaked}_{a_j}(\mathit{regOf}_{a_j}(l))
\end{aligned}$$

$$\mathit{reps}_{a_j}(\mathit{regOf}_{a_j}(l)) = \begin{cases} \mathit{reps}_{a_l}(\mathit{regOf}_{a_l}(l)) \cup \mathit{reps}_{a_r}(\mathit{regOf}_{a_r}(l)) \\ \qquad \text{if } \mathit{reps}_{a_l}(\mathit{regOf}_{a_l}(l)) \neq \emptyset \neq \mathit{reps}_{a_r}(\mathit{regOf}_{a_r}(l)) \\ \emptyset \\ \qquad \text{otherwise} \end{cases}$$

Note that The following rules are obeyed:

$$\begin{aligned}
&\forall a, b, c \in A\colon \\
&\qquad a \sqcup b = a \sqcup b && \text{(commutativity)} \\
&\qquad a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c && \text{(associativity)} \\
&\qquad a \sqcup a = a && \text{(idempotency)}
\end{aligned}$$

This is necessary for $\sqcup$ to be a legal join operator. We are not going to proof these properties formally, but we give the general idea as to why they hold:

- The definition of $\sqcup$ makes assertions about pairs of labels. However, it does not restrict the order of the labels. The definition is the same no matter how the two labels get ordered. Therefore, the operator is *commutative.*

- *Associativity* follows from the associativity of all the logical operations present in the definition, and the fact that the separation relations are associative as well. This last claim can be verified easily by looking at all possible combinations of the separation relations. We do not do this for all combinations, but we show that the claim holds for a single combination:

$$\underbrace{\underbrace{(\overset{R}{\Longleftrightarrow}, \overset{R}{\Longleftrightarrow})}_{\overset{R}{\Longleftrightarrow}}, \overset{R}{=\!=}}_{\overset{R}{\longleftrightarrow}} \quad \text{is the same as} \quad \underbrace{\overset{R}{\Longleftrightarrow}, \underbrace{(\overset{R}{\Longleftrightarrow}, \overset{R}{=\!=})}_{\overset{R}{\longleftrightarrow}}}_{\overset{R}{\longleftrightarrow}}$$

- *Idempotency* holds since all mathematical operations present in the definition are idempotent as well.

4.4.2 Completing the Complete Lattice

We define the missing operators that belong to the complete lattice in terms of the join operator.

Definition 4.4.3 (the partial order operator $\sqsubseteq$).

$$\forall a, b \in A_{rout} : \quad a = a \sqcup b \implies b \sqsubseteq a$$

Definition 4.4.4 (the greatest lower bound – or *meet* – operator $\sqcap$).

$$\forall Y \subseteq A_{rout} : \quad \sqcap Y = \sqcup \{a \in A_{rout} \mid \forall a' \in Y : a \sqsubseteq a'\}$$

Having a complete lattice as our abstract domain is of importance to the region analysis. It allows us to always find a safe approximation of the state in finite time.

4.5 Treatment of Routine Calls

The general scheme of all transfer functions is to mimic the actual computation on every piece of the program text, in the same order the program would be executed at runtime. Just like a real interpreter[2], the region analysis examines a program fragment (statement or expression), and replicates its effect as precisely as possible on the state graph that acts as the input to this program fragment. For example, every expression that gets evaluated will result in a new entry on the evaluation stack of the state graph.

A bit more involved are routine calls. We first describe a solution for approximating the effects of routine calls without analyzing the called routine's body. Then, we show how the interpretation of the routine's body can greatly increase the precision.

A given routine's type signature tells us if a result is returned, and – if so – if it is statically declared as `separate` or not. In the former case, the only save thing to assume in the caller is that the result is on a new abstract region that is not separated from any other abstract region in the caller's state graph. This way, we do not commit ourselves to a concrete location of the returned object reference and leave every possibility as an option.

Besides returning a result, the called routine may have an arbitrary effect on the heap. In fact, the routine may be called on the same object as the one the calling routine is executed on, that is, if the static types conform or if the call is unqualified. But even if it is not directly called on the same object, it may reach it nonetheless by calling other routines recursively. So from the viewpoint of the calling routine – and without inspecting the called routine's

[2]Interpreters are programs that read and execute the program text piece after piece. This is contrasted with compilers, which first translate the whole program text into machine code, before execution.

body – nothing about the state of the heap can be assumed after the call returns, and all we can do is remove any attribute label present in the target routine's state graph. That is, unless the analysis can *assure* that the call is asynchronous. In this case, we know that the call will be scheduled for later execution, and that it cannot possibly mess with the heap as seen by the target routine, at least not until after the target routine itself returned.

We can do better than this, at the expense of additional computations that need to be performed for every routine under consideration. The approach is straightforward: We inspect the body of the called routine and continue the analysis there. This is why the region analysis is called a context-sensitive analysis. So we expand from a purely intra-procedural to an inter-procedural abstract interpretation. This expansion gives rise to three main challenges:

- How to come up with an initial state graph for the called routine?
- What to do with the resulting state graph of the called routine?
- How efficient is this approach? Does it terminate?

We give answers to each of these questions in the following sections.

4.5.1 Callsite Transfer Function

The *callsite transfer function* takes care of the construction of the initial state graph passed on to a called routine. As always, the most important criterion is to be safe while maximizing the precision of the analysis.

Before the analysis can change its context from the target routine to the callee, it must go over all actual arguments and the target on which to call the routine, if any. Because a target/actual argument has to be an expression, we can be sure that the topmost elements on the evaluation stack correspond to the actual arguments. This information can thus be used to set up the abstract regions of the called routine's *controlled labels*. Remember that en element of the evaluation stack is not necessarily only an abstract region. It may also be a *controlled label* of the state graph. If so, we can not only take over its abstract region, but also the associated attribute labels and their regions. The callee will have exclusive access to the local heaps of its controlled arguments, since in this case, we know that lock-passing will be triggered and the caller will have to wait before reclaiming its locks.

We can also take over all separating edges between abstract regions as present in the state graph of the target routine, as long as both abstract regions are taken over as well.

The initial state graph passed to the called routine may vary depending on the calling context, i.e. the state of the caller while issuing the call. But we can be sure that the number of possible initial state graphs for some given routine is finite, since the abstract domain is finite as well. This property is important when it comes to the termination criteria of the region analysis.

4.5.2 Return Transfer Function and Representations

Once the region analysis completed the interpretation of the called routine, the calling routine's interpretation can continue. Recall that the region analysis interprets the chain of calls in a sequential fashion, no matter if some routine calls are actually asynchronous at runtime.

Listing 4.1: Eiffel*: reference code for the callsite and return transfer functions example*

```
...

-- In some class:
foo (x,y: separate EXAMPLE)
  do
    x.bar (y)
  end

...

-- In some other class, EXAMPLE:
a: separate ANY

bar (z: separate EXAMPLE)
  do
    if some_condition then
      a := z
    else
      a := Current
    end
  end

...
```

The task of the *return transfer function* is to inject the state graph as obtained by the interpretation of the called routine body back into the state graph of the calling routine. The caller will just take over as much information from the callee as possible. Some information, like attribute labels attached to controlled labels not known to the caller, will get lost.

Representations

In order to further improve the precision of the return transfer function, we make use of an abstract region's *representation* set. We briefly introduced this concept when we were formalizing the state graph in section 3.2.2, but did not further elaborate on it.

Every region in the state graph may represent an arbitrary number of other regions that do not necessarily have to exist in the same state graph at all times. Their purpose is to maintain a correspondence between the regions in the callee and the regions in the caller. Right after the region analysis starts the interpretation of a called routine, the initial state graph's regions that are known to the caller are representatives of themselves. Then, whenever two abstract regions with non-empty representations get combined via a join operation, we take the union of their representations as the resulting region's representation set, because the latter may represent anything its two operands have. However, when we merge a region with an empty representation set with any other region, the resulting region will still have an empty representation set. In this case, the caller does not know the merged region, so it may be *any* region. The caller either knows everything, or nothing.

Once we return to the caller, we can inspect the representation sets of the returned state graph's abstract regions and *regain* the correspondence to the abstract regions in the caller.

Figure 4.2 shows an example of such a routine call. The underlying code can be seen

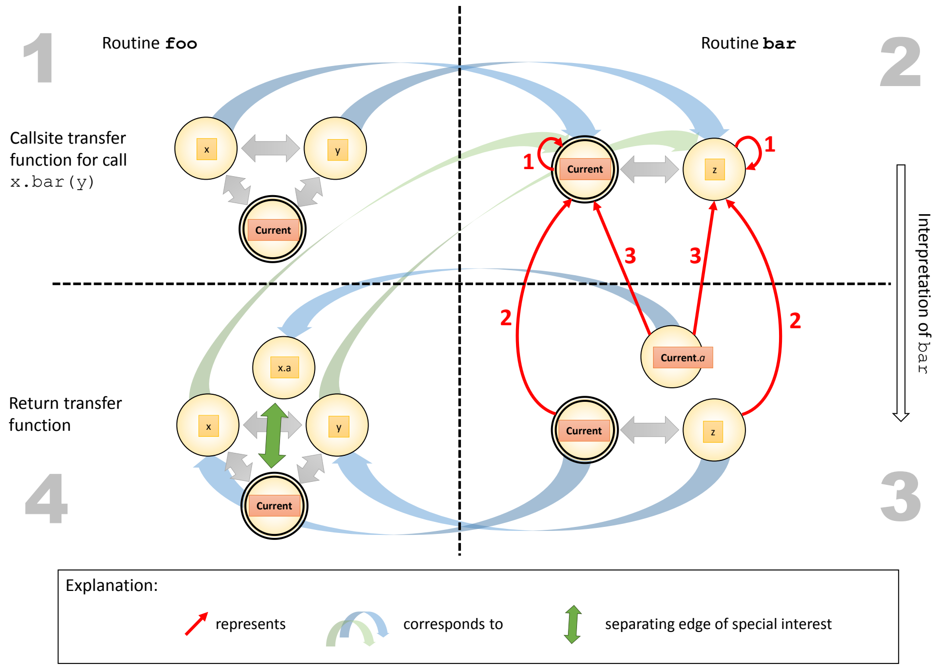


***Figure 4.2:** callsite and return transfer functions working with representation sets*

in listing 4.1. The two controlled formal arguments in the caller (`foo`), `x` and `y`, reside on separate abstract regions, both relative to each other, and relative to the region of `Current`. The caller will pass both controlled labels over to the callee, which in turn either assigns its attribute `a` to `z` or `Current`. The abstract interpretation can not know which choice will be taken, so it has to interpret both branches and combine the outcomes with the join operation, resulting in the state graph at the bottom right. This state graph also combines the representatives, drawn as red arrows. This means that the abstract region at the beginning of the arrow represents the abstract region at the end of the arrow. Notice that at the beginning of `bar`'s interpretation, both regions represent themselves, because they are both known to the caller, as indicates by the blue arrows.

Upon returning, the caller will see that the just modified attribute `a` (now correctly viewpoint-adapted to be attached to the caller's controlled label `x`) will point to some newly added abstract region. The reason why representation sets are useful is demonstrated by the additional separating edge (green) that we can add between the regions of `Current` and `x.a`. The following chain of reasoning explains why we can safely add this edge:

1. The return transfer function of `foo` still knows the association between its regions and the regions of `bar`'s initial state graph at the top right (green arrows).
2. It also knows that the region containing `x.a` comes from the region containing `Current .a` (topmost blue arrow from right to left) in the final state graph of `bar` (bottom right).
3. This region also represents the initial two regions (red arrows, labeled with 3).
4. Therefore, the caller can conclude (in the state graph at the bottom left) that the region of `x.a` must represent the regions both the region of `x` and the one of `y`.
5. Since both of these are separated from the region containing `Current` (still in the state

graph at the bottom left), it follows that also the region of `x.a` must be separate from the one of `Current`.

4.5.3 Efficiency and Termination

Termination is guaranteed by the fact that the abstract domain is finite. So there could be many different contexts for a single routine, but eventually, every possible context must be covered. Special care must be taken when handling recursive calls under the same context. We have a closer look at this problem when discussing implementation-specific issues in section 5.1.3.

The region analysis is designed to give programmers useful insights into the region topology of the program. Therefore, not using contexts and instead computing summaries for every routine that hold under all circumstances would have been too imprecise. In fact, it may not have been much more than what the type system tells us anyway. Using different contexts on the other hand improves the precision, but also introduces quite a bit of extra work. However, this overhead is not that big of a deal, since the viewpoint-adapted abstract spaces are manageable, even more so if the routines have only few formal arguments, which is recommended as a good programming style anyway.[3]

We use only contexts that are independent of any of the routines in the call-hierarchy, which further reduces the number of possibilities. They only contain elements known to the particular routine, but at the same time, concepts like the representation sets and freshness flags let the callers of a routine (and – transitively – even their callers) make precise adoptions to their own state graphs as well.

4.6 The Inter-Procedural Context

The context remains constant during the analysis of a given routine, and for each context, a routine gets analyzed individually.

$$\delta = (rout, a) \in \Delta := \textbf{Routine} \times A \qquad \text{(context information)}$$

The context is just another state graph that reflects the current situation of the abstract state in which the routine gets called. Naturally, different starting conditions may yield different analysis results.

For a context to be legal, certain rules have to be obeyed. These are (a denotes any input state graph, and $rout$ denotes the routine):

Invariant 4.6.1 (The state graph must be an element of the corresponding viewpoint-adapted abstract space.)**.**

$$\forall (rout, a) \in \Delta\colon\ a \in A_{rout}$$

Invariant 4.6.2 (The state graph has no locals, and nothing on the evaluation stack.)**.**

$$\forall (rout, a) \in \Delta\colon\ \mathrm{dom}(a_{\mathrm{locals}}) = \emptyset \wedge \mathrm{dom}(a_{\mathrm{evals}}) = \emptyset$$

[3]Too many formal arguments bloat the API and make it difficult for the programmer to understand the meaning of actual arguments involved in a routine call.

Invariant 4.6.3 (Every region in the state graph represents itself or nothing.).

$$\forall rout \in allRegions_a\colon\ represents_a(rout) = rout \cup undefined$$

4.7 Call Classification

One of the goals of the region analysis is to provide classifications for every reachable routine call in the system's source code. These classifications should provide information about the concurrent nature of the call. In this section, we informally describe and formally define a routine call classification.

A classification is fully determined by the state graph (a) in the calling routine, right before the call is issued, but *after* all actual arguments have been evaluated. We are interested in two core properties: Lock-passing and asynchrony.

Lock-passing Lock-passing is the act of the calling processor handing all currently held locks over to the processor that handles the call target's region. We say that lock-passing also occurs if the call is issued on a target located on the same region as the calling processor's region (i.e. a non-separate call). Because in this case, all locks are "passed" trivially.

- A routine call **involves lock-passing** if and only if any of the actual argument's regions is locked or the call is non-separate.
- A routine call **involves *no* lock-passing** if and only if the call has no arguments and the call target is on an abstract region that is separate to the caller's region.
- A routine call ***potentially* involves lock-passing** if and only if neither of the above holds.

Asynchrony The calling processor may either wait for the call to finish, or it may continue with the execution of successive instructions concurrently to the issued call's execution.

- A routine call is **synchronous** if and only if it involves lock-passing or it awaits a result (i.e. is a function call).
- A routine call is **asynchronous** if and only if it involves no lock-passing and does not await a result (i.e. is a procedure call).
- A routine call is **undecided** if and only if neither of the above holds.

Note that the static type system can never guarantee that a separately declared entity is on a different concrete region at runtime or not. Statically deciding that lock-passing happens is also only possible when at least one of the actual arguments is a separate formal of the calling routine, any non-separate entity, or any non-separate query applied to a separate formal.

The region analysis improves this situation with more accurate classifications than what the type system can provide. But it is *always at least as precise* as the type system.

The formal description of a classification is based on some state graph a and a called routine $rout$.

Definition 4.7.1 (call classification set)**.**

$$(\mathit{must\_lockpass}, \mathit{no\_lockpass}, \mathit{may\_lockpass}, \mathit{synchronous}, \mathit{asynchronous}, \mathit{undecided}) \in \\ \textbf{Classification} := \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}$$

Definition 4.7.2 (call classification query)**.**

$$\begin{aligned} \mathit{classification} &\colon A \times \textbf{Routine} \to \textbf{Classification}, \\ \mathit{classification}_a(\mathit{rout}) &\mapsto c \qquad \text{such that:} \end{aligned}$$

$$\begin{aligned} c_{\text{must\_lockpass}} &\iff \big(\exists r \in R_{\text{actuals}} \colon \mathit{locked}_a(r)\big) \lor \mathit{regOf}_a(\texttt{Current}) = r_t \\ c_{\text{no\_lockpass}} &\iff \big|\mathit{rout}_{\text{signature}}\big| = 0 \land \mathit{separate}_a(\mathit{regOf}_a(\texttt{Current}), r_t) \\ c_{\text{may\_lockpass}} &\iff \neg c_{\text{must\_lockpass}} \land \neg c_{\text{no\_lockpass}} \\ c_{\text{synchronous}} &\iff c_{\text{must\_lockpass}} \lor \mathit{isFunction}(\mathit{rout}) \\ c_{\text{asynchronous}} &\iff c_{\text{no\_lockpass}} \land \neg \mathit{isFunction}(\mathit{rout}) \\ c_{\text{undecided}} &\iff \neg c_{\text{synchronous}} \land \neg c_{\text{asynchronous}} \end{aligned}$$

where

$$\begin{aligned} R_{\text{actuals}} &= \{\mathit{regOf}_a(\mathit{eval}) \mid \mathit{eval} \in (\mathit{evalStack}_a)^{|\mathit{evalStack}_a|}_{|\mathit{evalStack}_a| - |\mathit{rout}_{\text{signature}}| + 1}\} \\ r_t &= (\mathit{evalStack}_a)_{|\mathit{evalStack}_a| - |\mathit{rout}_{\text{signature}}|} \\ \mathit{isFunction}(\mathit{rout}) &:= (\mathit{rout}_{\text{returntype}} \neq \text{NONE}) \end{aligned}$$

4.8 Formal Description of Transfer Functions

Transfer functions produce a new state graph from one or two given state graphs and a program fragment. There are two main types of transfer functions:

Intra-procedural These transfer functions always take *one* state graph and a program fragment, and compute an *updated* state graph. The functions belonging to this category operate entirely *inside* a given routine body.

Inter-procedural There are exactly two such transfer functions: The *callsite transfer function* and the *return transfer function*. Together they specify the semantics of a routine call in our abstract framework. Note that the return function is the only one that operates on two state graphs.

First, we have a look at the intra-procedural transfer functions. In general, they are defined as follows:

$$\begin{aligned} \phi^{\text{RA}}_{l,rout} &\colon (\Delta \to A_{rout}) \to (\Delta \to A_{rout}), \\ \phi^{\text{RA}}_{l,rout}((\delta, a_i)) &\mapsto (\delta, a_o) \end{aligned}$$

where *rout* $\in$ **Routine** is the currently analyzed routine. *l* denotes the *program fragment* on which the transfer function is applied. There are as many intra-procedural transfer functions as there are different l's. In the following, we refer to the input state graph as a_i and to the output state graph as a_o.

For our convenience, we define auxiliary functions that help us express the various effects of the intra-procedural transfer functions more readily.

The following operation *modifies* a given state graph by adding an additional label (of any kind but controlled) to one of its abstract regions.

Definition 4.8.1 (add a label to a region)**.**

$$\mathit{withLabel}\colon A \times \mathbf{AReg} \times \mathbf{Lab} \cup \mathbf{QLab} \cup \mathbf{Eval} \to A,$$

$$\begin{aligned}
\mathit{withLabel}(a, r, l) \mapsto a' \quad & \text{such that} \\
a'_{\text{regs}} &= (a_{\text{regs}} \setminus r) \cup (\mathit{ctrls}, \mathit{attrs}, \mathit{locals}, \mathit{evals}), \\
& \text{where} \\
& \mathit{ctrls} = r_{\text{ctrls}} \\
& \mathit{attrs} = \begin{cases} r_{\text{attrs}} \cup l & \text{if } l \in \mathbf{QLab} \\ r_{\text{attrs}} & \text{otherwise} \end{cases} \\
& \mathit{locals} = \begin{cases} r_{\text{locals}} \cup l & \text{if } l \in \mathbf{Lab} \\ r_{\text{locals}} & \text{otherwise} \end{cases} \\
& \mathit{evals} = \begin{cases} r_{\text{evals}} \cup l & \text{if } l \in \mathbf{Eval} \\ r_{\text{evals}} & \text{otherwise} \end{cases} \\
a'_{\text{sep}} &= \{s \in a_{\text{sep}} \mid r \notin s\} \cup \{\{r_t, \mathit{regOf}_{a'}(l)\} \mid (r_t, r) \in a_{\text{sep}}\} \\
a'_{\text{props}} &= a_{\text{props}} \\
a'_{\text{rep}} &= a_{\text{rep}}[r \mapsto \emptyset][\mathit{regOf}_{a'}(l) \mapsto \mathit{reps}_a(r)]
\end{aligned}$$

The notation $m[k \mapsto v]$ is only meaningful if m is an existing map that gets modified by changing (or adding if not yet present) the mapping from the key $k \in \mathrm{dom}(m)$ to the value $v \in \mathrm{im}(m)$.

The next operation will just do the opposite: Remove a non-controlled label from an existing state graph.

Definition 4.8.2 (remove a label from a state graph).

$$\mathit{withoutLabel}\colon A \times \mathbf{Lab} \cup \mathbf{QLab} \cup \mathbf{Eval} \to A,$$

$$\mathit{withoutLabel}(a, l) \mapsto a' \text{ such that}$$

$$a'_{\text{regs}} = \begin{cases} a_{\text{regs}} \setminus \mathit{regOf}_a(l) \\ \qquad \text{if } \forall l' \in \mathit{allLabels}_a, l \neq l'\colon \mathit{regOf}_a(l') \neq \mathit{regOf}_a(l) \\ a_{\text{regs}} \setminus \mathit{regOf}_a(l) \cup (\mathit{ctrls}, \mathit{attrs}, \mathit{locals}, \mathit{evals}), \\ \qquad \text{where} \\ \qquad \mathit{ctrls} = r_{\text{ctrls}} \\ \qquad \mathit{attrs} = \begin{cases} r_{\text{attrs}} \setminus l & \text{if } l \in \mathbf{QLab} \\ r_{\text{attrs}} & \text{otherwise} \end{cases} \\ \qquad \mathit{locals} = \begin{cases} r_{\text{locals}} \setminus l & \text{if } l \in \mathbf{Lab} \\ r_{\text{locals}} & \text{otherwise} \end{cases} \\ \qquad \mathit{evals} = \begin{cases} r_{\text{evals}} \setminus l & \text{if } l \in \mathbf{Eval} \\ r_{\text{evals}} & \text{otherwise} \end{cases} \\ \quad \text{otherwise} \end{cases}$$

$$a'_{\text{sep}} = \{s \in a_{\text{sep}} \mid \mathit{regOf}_a(l) \notin s\} \cup \{\{r_t, \mathit{regOf}_{a'}(l)\} \mid (r_t, r) \in a_{\text{sep}}\}$$

$$a'_{\text{props}} = a_{\text{props}}$$

$$a'_{\text{rep}} = a_{\text{rep}}[r \mapsto \emptyset][\mathit{regOf}_{a'}(l) \mapsto \mathit{reps}_a(r)]$$

With these operations at our disposal, we move on to the actual definitions of the transfer functions.

4.8.1 Expressions

An expression e in the simplified SCOOP language has to be *legal*, as defined in chapter 2. e can be one of the following:

Dynamic label $e = [\![\, l \,]\!]$ where $l \in \mathbf{Lab} \cup \mathbf{QLab}$ is the name of a local variable (including `Result`) or an attribute name of some existing controlled label of the target routine. The output state graph a_o can be specified in terms of the input state graph a_i:

$$\mathit{evalStack}_{a_o} = \mathit{evalStack}_{a_i} + \epsilon$$

$$a_o = \begin{cases} \mathit{withLabel}(a_i, \mathit{regOf}_{a_i}(l), \mathit{top}_{a_o}) & \text{if } l \in \mathit{allLabels}_{a_i} \\ ((a_i)_{\text{regs}} \cup (\{\}, \{\}, \{\}, \mathit{top}_{a_o}), (a_i)_{\text{sep}}, (a_i)_{\text{props}}, (a_i)_{\text{rep}}) & \text{otherwise} \end{cases}$$

Note that $+$ in the first line means *appending* an element to the evaluation stack (which is a sequence).
Informally, the evaluation of a local label puts a new element on the evaluation stack. If the label is already present in the state graph, we attach the evaluation element to the same abstract region as the one containing the local label. However, if the local label does not yet exist, we create an entirely new abstract region (not separate to any other region) and attach the evaluation element to it. This makes sense, since we have no information whatsoever about the region the unknown local label refers to.

Controlled label $e = [\![\, c \,]\!]$ where $c \in \mathit{controlled}_{a_i}$ is the name of a formal routine argument or `Current`. The output state graph a_o can be specified in terms of the input state graph a_i:

$$\begin{aligned} \mathit{evalStack}_{a_o} &= \mathit{evalStack}_{a_i} + c \\ a_o &= \mathit{withLabel}(a_i, \mathit{regOf}_{a_i}(c), \mathit{top}_{a_o}) \end{aligned}$$

This transfer function is similar to the one before. However, it is not possible that a controlled label does not yet exist, since it must be in the state graph since the beginning of the target routine's interpretation (we will see later how the initial state graph of some routine gets constructed). Also, note that the evaluation stack gets extended with the name of the controlled label, instead of the anonymous ϵ.

Unqualified function call $e = [\![\, m(a_1, \ldots, a_n) \,]\!]$ where m is the unqualified name of some function that has actual argument expressions a_1 to a_n for some $n \geq 0$.
First, the `Current` controlled label must be pushed onto the evaluation stack (to give the call a proper call target and treat it the same way as a qualified one). Then the corresponding transfer functions are applied to all actual argument expressions (in the order from left to right), if any. Finally, the transfer function for routine calls is applied, like so:

$$(\delta_t, a_t) = f^{\text{RA}}_{\mathit{rout}}((\delta_i, a_i))$$

Then, (δ_t, a_t) will be sent to the called routine, rout. Next, the result of the called routine's interpretation (denoted by (δ_f, a_f)) will be combined with a_i to get a_o, the desired result.

$$(\delta_i, a_o) = g^{\text{RA}}_{\mathit{rout}}((\delta_i, a_i), (\delta_f, a_f))$$

Notice that the context of the target routine, δ_i, will not have changed by the call. Also, g takes care of adding an element onto the evaluation stack, since rout denotes (here) a function that produces a result.
We have a closer look at the definitions of f (callsite function) and g (return function) in the next section.

Qualified function call $e = [\![\, e_t.m(a_1, \ldots, a_n) \,]\!]$ where e_t is the *target expression*, m is the name of some function that potentially has actual argument expressions a_1 to a_n for some $n \geq 0$.
The corresponding transfer functions are first applied to the target expression, then to all actual argument expressions (in the order from left to right), if any. Finally, the callsite and return transfer functions are applied, just as in the unqualified case above.

Boolean expression $e = [\![\, (e_l = e_r) \,]\!]$ where e_l and e_r are two expression whose result ought to be compared on equality.
Note that in our simplified SCOOP language, this is the only boolean expression allowed. The transfer function in this case is best explained informally: First, the transfer functions of e_l is applied, then the one of e_r. Finally, the evaluation stack gets emptied since we are not interested in the result of the expressions.

The literal value `Void` $e = [\![$ `Void` $]\!]$ This is the only literal value that is included in our simplified language. When evaluating it, a new abstract region will be added to the state graph, with only one label – an anonymous element on top of the stack – attached to it. Note that semantically, this is a bit of a hack, because there is no valid location on this abstract region. However, this solution is still safe, since if the program will try to access this location, it will fail anyway. And the job of the analysis is not to prohibit or detect any void-safety issues.

4.8.2 Statements

As opposed to expressions, statements do not alter the evaluation stack of a state graph. The output state graph will always have an empty evaluation stack. A statement s can be one of the following:

Assignment $s = [\![\ t\ \texttt{:=}\ e_s\ ]\!]$ where t is the *target* of the assignment, which can only be a local variable of the target routine, `Result`, or an unqualified attribute name of the target routine's enclosing class. e_s is the source expression of the assignment.
First, the transfer function for e_s gets applied. This ensures that a new element gets pushed onto the evaluation stack. The output state graph a_o can then be specified in terms of the input state graph a_i:

$$a_o := (\mathit{regs}, \mathit{sep}, \mathit{props}, \mathit{rep}) \quad \text{such that:}$$
$$a'_o = (\mathit{regs}, \mathit{sep}, \mathit{props}', \mathit{rep}) = \mathit{withoutLabel}(\mathit{withLabel}(a_i, \mathit{regOf}_{a_i}(\mathit{top}_{a_i}), t'), \mathit{top}_{a_i})$$
$$\wedge$$
$$\mathit{props} = \begin{cases} \mathit{props}'[r'_t \mapsto P' \mid & \\ \qquad \mathcal{F} \in P \implies \mathcal{L} \in P' \wedge & \\ \qquad \mathcal{F} \in P \iff \mathcal{F} \in P \wedge & \text{if } t' \in \mathit{allAttributes}_{a'_o} \\ \qquad \mathcal{L} \in P \iff \mathcal{L} \in P' & \\ ] & \\ \mathit{props}' & \text{otherwise} \end{cases}$$

where

$$t' = \begin{cases} (\texttt{Current}, t) & \text{if } t \text{ denotes an unqualified attribute} \\ t & \text{otherwise} \end{cases}$$
$$r'_t = \mathit{regOf}_{a'_o}(t')$$
$$P = \mathit{props}'[r'_t]$$

The effect is easily described: The target label gets attached to the region denoted by the top of the evaluation stack, while the top of the stack gets removed.
In case t denotes an unqualified attribute, we have to make sure that the region t was attached to becomes leaked if it was fresh before, because then it may be reachable via the heap.

Loop $s = [\![\,\texttt{until}\ e_c\ \texttt{loop}\ s_b\ \texttt{end}\,]\!]$ where e_c is a boolean expression and s_b is the statement in the body of the loop. a_o will be the *fixpoint* of the following function, if a_o gets initialized to a_i:

$$a_i \sqcup \phi^{\mathrm{RA}}_{e_c,rout}\left(\phi^{\mathrm{RA}}_{e_b,rout}\left(\phi^{\mathrm{RA}}_{e_c,rout}(a_o)\right)\right)$$

That is, we apply the transfer functions for the condition, the loop body, and again the condition, combine the result and a_i with the least upper bound operation, and repeat until convergence). This fixpoint does always exist, and is computable in finite time, because the transfer functions are monotone, and the complete lattice of state graphs is finite.

Branch $s = [\![\,\texttt{if}\ e_c\ \texttt{then}\ s_i\ \texttt{else}\ s_e\ \texttt{end}\,]\!]$ where e_c is a boolean expression and s_i and s_e are the bodies of the respective branches.
a_o can be described in terms of a_i:

$$a_o = \phi^{\mathrm{RA}}_{s_i,rout}(\phi^{\mathrm{RA}}_{e_c,rout}(a_i)) \sqcup \phi^{\mathrm{RA}}_{s_e,rout}(\phi^{\mathrm{RA}}_{e_c,rout}(a_i))$$

In other words, we build the least upper bound of the results of the transfer functions applied to the condition followed by the if-branch, and the condition followed by the else-branch.

Unqualified procedure call $s = [\![\ m(a_1, \ldots, a_n)\ ]\!]$ where the constituents are equal to the ones already encountered by *unqualified function calls*, with the exception that m here denotes the name of a procedure (not a function). Finally, we apply the callsite transfer function, just as we did when encountering unqualified or qualified function calls. The difference here will be that no element will be pushed onto the evaluation stack, since the called routine does not return anything.

Qualified procedure call $s = [\![\ e_t.m(a_1, \ldots, a_n)\ ]\!]$ where the constituents are equal to the ones already encountered by *qualified function calls*, with the exception that m here denotes the name of a procedure (not a function). Finally, we apply the callsite transfer function.

Creation procedure call $s = [\![\ \texttt{create}\ t.m(a_1, \ldots, a_n)\ ]\!]$ where t stands for either a local variable or an unqualified attribute name of the enclosing class of the target routine. The effect of the transfer function is as follows:

- If t is declared as a separate entity, a completely new abstract region gets added to the state graph. This region is fresh, non-leaked, and separate to any other region currently present in the state graph. If t is declared as a non-separate entity, the abstract region that contains the newly created object is the same as the one containing `Current`.
- Then, this region gets anonymously pushed onto the evaluation stack, and acts as the call target for the creation procedure call. Also, transfer functions for all actual arguments get applied.
- Finally, the transfer function for a regular procedure call gets applied.

4.8.3 Callsite Transfer Function

This callsite transfer function is declared like this:

$$\begin{aligned} f^{\mathrm{RA}}_{rout} &: (\Delta \to A) \to (\Delta_{rout} \to A_{rout}), \\ f^{\mathrm{RA}}_{rout}((\delta_c, a_c)) &\mapsto (\delta_i, a_i) \end{aligned}$$

For clarity, we used a different symbol for this function, f (as opposed to ϕ), because it is a transfer function that acts *across* different routines in the system, while most of the others act *within* some routine. But in order to stay compatible with the definitions of the intra-procedural transfer functions, we may regard f^{RA}_{rout} as an alias of $\phi^{\mathrm{RA}}_{call,rout}$. Furthermore, we refer to the current state (context and state graph) in the calling routine with subscripts c (for c̲urrent), and to the initial state in the called routine with subscripts i (for i̲nitial). $rout$ denotes the called routine, since we have to know which routine gets called (formal argument names, body the analysis should continue with, etc.). The calling routine on the other hand is of no importance to the callsite function.

We define f by describing δ_i and a_i in terms of δ_c and a_c. As a precondition, we require that there are at least $n + 1$ elements on the evaluation stack, if n is the number of formal arguments required by $rout$. The additional element is the target on which the routine gets called.

The callsite function takes the elements on the caller's evaluation stack, and translates them over to the callee by using the following rules:

- If the evaluation element corresponds to one of the actual arguments to the callee, we look up the corresponding formal argument name in the callee's signature.
- If the evaluation element corresponds to the call target, we know that in the callee, the corresponding controlled label must be called `Current`.

Definition 4.8.3 (translate an evaluation stack element to a controlled label in the callee routine)**.**

$$\begin{aligned}
&\mathit{translateEval}\colon A \times \mathbf{Eval} \times \mathbf{Routine} \to \mathbf{Lab}\\
&\mathit{translateEval}_a((i,e), \mathit{rout}) \mapsto \begin{cases}
((\mathit{rout}_{\text{signature}})_{\text{formals}})_{(\text{n}-\text{i}+1)} & \\
\qquad \text{if } 0 \le (n-i) < \left|(\mathit{rout}_{\text{signature}})_{\text{formals}}\right| & \\
\texttt{Current} & \\
\qquad \text{if } (n-i) = \left|(\mathit{rout}_{\text{signature}})_{\text{formals}}\right| & \\
\mathit{undefined} & \\
\qquad \text{otherwise} &
\end{cases}\\
&\text{where} \quad n = |\mathit{evalStack}_a|
\end{aligned}$$

An evaluation stack element can correspond to a controlled label in the caller, if it is not anonymous. If one of these gets passed to the callee, we have a one-to-one correspondence between a controlled label in the caller and a controlled label in the callee. The following function allows us to query the corresponding controlled label in the callee, if it exists. This correspondence is important, since the attributes of these controlled labels are also shared between the caller and the callee.

Definition 4.8.4 (translate a controlled label from the caller to the callee)**.**

$$\begin{aligned}
&\mathit{translateCtrl}\colon A \times \mathbf{Lab} \times \mathbf{Routine} \to \mathbf{Lab}\\
&\mathit{translateCtrl}_a(c, \mathit{rout}) \mapsto \begin{cases}
((r_{\text{signature}})_{\text{formals}})_i & \\
\qquad \text{if } \exists (i,e) \in \mathit{evalStack}_a\colon & \\
\qquad\qquad 0 \le (n-i) < \left|(\mathit{rout}_{\text{signature}})_{\text{formals}}\right| \wedge e_{\text{eval}} = c & \\
\texttt{Current} & \\
\qquad \text{if } \exists (i,e) \in \mathit{evalStack}_a\colon & \\
\qquad\qquad (n-i) = \left|(\mathit{rout}_{\text{signature}})_{\text{formals}}\right| \wedge e_{\text{eval}} = c & \\
\mathit{undefined} & \\
\qquad \text{otherwise} &
\end{cases}\\
&\text{where} \quad n = |\mathit{evalStack}_a|
\end{aligned}$$

The abstract regions that are associated to the evaluation elements in the caller get translated to the callee as well. In order to simplify their definitions in the callee, we introduce two functions that both take a region in the caller, and determine the set of controlled labels respectively attribute labels that are attached to the same region in the callee. This is important because regions are fully and exclusively identified by the labels attached to them.

Definition 4.8.5 (controlled labels and attribute labels in the callee attached to a region passed from the caller to the callee)**.**

$$\begin{aligned} \mathit{calleeCtrl}\colon A \times \mathbf{Routine} \times \mathbf{AReg} &\to \mathcal{P}(\mathbf{Lab}) \\ \mathit{calleeCtrl}_{a_c}(\mathit{rout}, r) &\mapsto \{c \mid \exists \mathit{eval} \in r_{\text{evals}}\colon \mathit{translateEval}_{a_c}(\mathit{eval}, \mathit{rout}) = c\} \end{aligned}$$

$$\begin{aligned} \mathit{calleeAttr}\colon A \times \mathbf{Routine} \times \mathbf{AReg} &\to \mathcal{P}(\mathbf{QLab}) \\ \mathit{calleeAttr}_{a_c}(\mathit{rout}, r) &\mapsto \begin{cases} \{(t, a) \mid \exists (t', a) \in r_{\text{attrs}}\colon \mathit{translateCtrl}_{a_c}(t', \mathit{rout}) = t\} \\ \qquad \text{if labels } t' \text{ and } \texttt{Current} \text{ may not be aliased} \\ \emptyset \\ \qquad \text{otherwise} \end{cases} \end{aligned}$$

Notice the case distinction of the *calleeAttr* function. We only take over attribute labels for controlled labels that may not be aliased (i.e. whose types do not conform, and/or whose regions are separated) by the `Current` controlled label in the caller's state graph. Because if they are indeed aliases of each other at runtime, the analysis would not be safe any more.

When specifying the callsite transfer function – and later on also the return function –, we have to be able to easily switch between corresponding entities in the caller and callee. The following two functions allow the querying of corresponding abstract regions.

Definition 4.8.6 (export/import an abstract region from the caller to/from the callee)**.**

$$\begin{aligned} \mathit{export}\colon A \times \mathbf{Routine} \times \mathbf{AReg} &\to \mathbf{AReg} \\ \mathit{export}_a(\mathit{rout}, r) &\mapsto (\mathit{calleeCtrl}_a(\mathit{rout}, r), \mathit{calleeAttr}_a(\mathit{rout}, r), \emptyset, \emptyset) \end{aligned}$$

$$\begin{aligned} \mathit{import}\colon A \times \mathbf{Routine} \times \mathbf{AReg} &\to \mathbf{AReg} \\ \mathit{import}_a(\mathit{rout}, r) &\mapsto \begin{cases} r' & \text{if } \mathit{export}_a(\mathit{rout}, r') = r \\ \mathit{undefined} & \text{otherwise} \end{cases} \end{aligned}$$

Using the just defined auxiliary functions, we are able to specify the callsite transfer function in a constructive way.

Definition 4.8.7 (callsite transfer function).

$$
\begin{aligned}
& f^{\mathrm{RA}}_{rout}((\delta_c, a_c)) \mapsto (\delta_i, a_i) \quad \text{such that:}\\
&\quad (a_i)_{\mathrm{reg}} = R_{\mathrm{exported}}\\
&\quad (a_i)_{\mathrm{sep}} = \{\{\hat{r_1}, \hat{r_2}\} \mid \hat{r_1} = \mathit{export}_{a_c}(\mathit{rout}, r_1) \wedge \hat{r_2} = \mathit{export}_{a_c}(\mathit{rout}, r_2)\\
&\qquad\qquad \wedge \hat{r_1} \neq \mathit{emp} \neq \hat{r_2} \wedge \{r_1, r_2\} \in (a_c)_{\mathrm{sep}}\}\\
&\quad (a_i)_{\mathrm{props}} = [r \mapsto P \mid r \in R_{\mathrm{exported}},\\
&\qquad \mathcal{K} \in P \iff (\mathit{locked}_{a_c}(\mathit{import}_{a_c}(r))\\
&\qquad\qquad \wedge (\mathit{classification}(a_c, \mathit{rout}))_{\mathrm{must\_lockpass}}) \vee r_{\mathrm{ctrls}} \neq \emptyset,\\
&\qquad \mathcal{F} \in P \iff \mathit{fresh}_{a_c}(r) \wedge \neg \mathit{leaked}_{a_c}(r),\\
&\qquad \mathcal{L} \notin P\\
&\quad]\\
&\quad (a_i)_{\mathrm{rep}} = [r \mapsto \{r\} \mid r \in R_{\mathrm{exported}}]\\
&\quad \delta_i = a_i\\
&\quad \text{where}\\
&\qquad R_{\mathrm{exported}} = \{\hat{r} \mid \hat{r} = \mathit{export}_{a_c}(\mathit{rout}, r') \wedge r' \in (a_c)_{\mathrm{reg}} \wedge \hat{r} \neq \mathit{emp}\}\\
&\qquad \mathit{emp} = (\emptyset, \emptyset, \emptyset, \emptyset)
\end{aligned}
$$

Notice that the context in the caller δ_c does not play any role in the transfer function's definition, and the context in the callee δ_i is just the initial state graph a_i. Concerning the properties of the regions in the callee, the following must hold:

- If at least one controlled label is attached to it, it has to be locked. If not, it can still be locked, namely when the call triggers lock-passing and the corresponding region in the caller was locked to begin with.
- Fresh regions that are passed to the callee remain fresh, but only if they are not leaked. Leaked fresh regions are of no use to the callee, because we can not be sure if an alias is encountered via the heap at a later point in the callee's analysis. But if the region is fresh and not leaked, we can be sure that it will remain a unique region during the analysis of the callee.
- Following this argumentation, it becomes clear the no region in the callee's initial state graph can be leaked.

The role of abstract regions' representations is explained in detail in section 4.5.2. All regions that are known to the caller represent themselves in the initial state graph of the callee.

4.8.4 Return Transfer Function

The return transfer function is declared like this:

$$
\begin{aligned}
g^{\mathrm{RA}}_{rout} &\colon (\Delta \to A) \times (\Delta \to A_{rout}) \to (\Delta \to A),\\
g^{\mathrm{RA}}_{rout}((\delta_c, a_c), (\delta_i, a_i)) &\mapsto (\delta_u, a_u)
\end{aligned}
$$

Just as with the callsite transfer function, we use a different symbol (g) to denote this inter-procedural function. The return function determines an updated version of the state graph in the calling routine (a_u, u for updated), based on both the state graph of the caller as it was before the routine call, and the state graph at the end of the callee's analysis, here still denoted by a_i (although the state graph is not *initial* any more).

Note that the return function is only defined for matching call/return pairs, that is, δ_i must be the context that was established by the callsite function beforehand.

In addition to the auxiliary functions $export$ and $import$, we define functions that allow us to translate *labels*.

Definition 4.8.8 (auxiliary functions that translate attribute labels between caller and callee)**.**

$$exportAttr \colon A \times \mathbf{Routine} \times \mathbf{QLab} \to \mathbf{QLab}$$

$$exportAttr(a, rout, (c,v)) := \begin{cases} (translateCtrl_a(c, rout), v) \\ \qquad \text{if } translateCtrl_a(c, rout) \neq undefined \\ undefined \\ \qquad \text{otherwise} \end{cases}$$

$$importAttr \colon A \times \mathbf{Routine} \times \mathbf{QLab} \to \mathbf{QLab}$$

$$importAttr(a, rout, (c',v)) := \begin{cases} (c,v) & \text{if } exportAttr(a, rout, (c,v)) = (c',v) \\ undefined & \text{otherwise} \end{cases}$$

The attribute labels' correspondence can only be judged based on the controlled label they are attached to and based on the evaluation stack at the moment of the call.

A special case needs to be considered when the called routine is a function. With the simplified SCOOP language, we made sure that every function assigns to a local variable called `Result` in its body. The effect of the return function then is to add another element to the evaluation stack that represents the result of the function. In order to determine this element, we have to examine the local label `Result` in the final state graph of the callee.

Definition 4.8.9 (constructs an evaluation stack element for function call results)**.**

$$fnRes \colon A \times \mathbf{Routine} \to \mathcal{P}(\mathbf{Eval})$$

$$fnRes_a(rout) \mapsto \begin{cases} \{(i,\sigma) \mid i = |evalStack_a| - |rout_{\text{signature}}|\} & \text{if } isFunction(rout) \\ \emptyset & \text{otherwise} \end{cases}$$

where

$$isFunction(rout) := (rout_{\text{returntype}} \neq \text{NONE})$$

Next, we define a function that can take an arbitrary label from the caller and translate it – if possible – to the callee.

Definition 4.8.10 (auxiliary function that translates arbitrary labels from the caller to the callee)**.**

$$exp\colon A \times \mathbf{Routine} \times \mathbf{QLab} \to (\mathbf{Lab} \cup \mathbf{QLab} \cup \mathbf{Eval})$$

$$exp_a(rout, l) := \begin{cases} l' & \text{if } translateCtrl_a(rout, l) = l' \neq undefined \\ l' & \text{if } exportAttr(a, rout, l) = l' \neq undefined \\ \texttt{Result} & \text{if } \{l\} = fnRes_a(rout) \\ undefined & \text{otherwise} \end{cases}$$

As a preparation step for the definition of the return function, we determine all the labels that will still be present in the updated state graph of the caller. This decision depends on the *classification* of the call under analysis:

- If the call *may* trigger lock-passing, everything that was established about the heap must be reset. This means in particular, that all attribute labels will be removed from the caller's state graph, such that the only remaining labels are the controlled labels, the local labels and the evaluation elements that are not used by the call as actual arguments and/or call target of the caller, in addition to all attribute labels established by the callee that have a correspondent in the caller.
- If the call *must* be asynchronous, the remaining labels are all the labels of the caller, except the attribute labels that may be aliases of the call target's attribute labels and the evaluation elements that were used by the call, in addition to all attribute labels established by the callee that have a correspondent in the caller.

Definition 4.8.11 (the set of labels that are present in the updated state graph of the caller)**.**

$$updatedLabels\colon A \times \mathbf{Routine} \times A \to \mathcal{P}(\mathbf{Lab} \cup \mathbf{QLab} \cup \mathbf{Eval})$$

$$updatedLabels(a, rout, a') := \begin{cases} controlled_a \cup locals_a \cup (evalStack_a)_1^{|evalStack_a|-|rout_{\text{signature}}|-1} \\ \cup\, fnRes_a(rout) \cup \{importAttr(a, rout, l) \mid l \in allAttributes_{a'}\} \\ \qquad \text{if } (classification(a, rout))_{\text{may\_lockpass}} \\ withoutAliases_a(callTarget_a(rout)) \\ \cup \{importAttr(a, rout, l) \mid l \in allAttributes_{a'}\} \\ \qquad \text{otherwise} \end{cases}$$

where

$$withoutAliases_a(t) = allLabels_a \setminus \{(c, a) \in allAttributes_a \mid$$
$$c \text{ conforms with } t \wedge t \in controlled_a \wedge \neg(c \overset{\text{R}}{\Longleftrightarrow}_a t)\}$$
$$callTarget_a(rout) = (evalStack_a)_{|evalStack_a|-|rout_{\text{signature}}|}$$

The potential aliases of the call target are those controlled labels whose static types conforms with the one of the call target's controlled label (*if* the target is a controlled label), and which are located on a non-separate abstract region relative to the region containing the call target.

Similar to the definition of the *join* operator in section 4.4.1, we specify the properties of the updated state graph of the caller in terms of the relationships of two distinct labels from our previously constructed set of labels. In addition to the invariants that apply for state graphs, these properties ensure that a_u is uniquely determined.

First, we specify those cases where two abstract regions (of two different labels) are *different* abstract regions without a separating edge between them.

- In the case where one label (l') is untouched by the callee and the other (l) is manipulated by the callee, the following must hold: l resides on an abstract region that is not known to the caller (i.e. one that does not represent anything), or it resides on some abstract region that represents at least one region that in the caller, is different than l' but not separate to it.
- In the case where both labels are untouched by the callee, their respective regions must be distinct but not separate in the caller.
- In the case where both labels are manipulated by the callee, their respective regions must be distinct but not separate in the callee.

$$\forall l, l' \in \mathit{updatedLabels}(a_c, \mathit{rout}, a_i), l \neq l' :$$

$$l \xleftrightarrow{\mathrm{R}}_{a_u} l' \Longleftarrow \begin{cases} (\mathit{reps}_{a_i}(\mathit{regOf}_{a_i}(\mathit{exp}_a(\mathit{rout}, l))) = \emptyset \wedge \neg \mathit{fresh}_{a_i}(\mathit{regOf}_{a_i}(\mathit{exp}_a(\mathit{rout}, l)))) \\ \vee\ \exists r \in \mathit{reps}_{a_i}(\mathit{regOf}_{a_i}(\mathit{exp}_a(\mathit{rout}, l))) : \\ \quad \mathit{import}_{a_c}(\mathit{rout}, r) \neq \mathit{regOf}_{a_c}(l) \wedge \\ \quad \neg \mathit{separate}_{a_c}(\mathit{import}_{a_c}(\mathit{rout}, r), \mathit{regOf}_{a_c}(l)) \\ \qquad \text{if } \mathit{exp}_{a_c}(\mathit{rout}, l) \neq \mathit{undefined} \wedge \mathit{exp}_{a_c}(\mathit{rout}, l') = \mathit{undefined} \\ l \xleftrightarrow{\mathrm{R}}_{a_c} l' \\ \qquad \text{if } \mathit{exp}_a(\mathit{rout}, l) = \mathit{undefined} = \mathit{exp}_a(\mathit{rout}, l') \\ \mathit{exp}_a(\mathit{rout}, l) \xleftrightarrow{\mathrm{R}}_{a_i} \mathit{exp}_a(\mathit{rout}, l') \\ \qquad \text{if } \mathit{exp}_a(\mathit{rout}, l) \neq \mathit{undefined} \neq \mathit{exp}_a(\mathit{rout}, l') \end{cases}$$

Next, we specify those cases where the regions of the two labels are separated.

- In the case where one label (l') is untouched by the callee and the other (l) is manipulated by the callee, one of the following must hold:
  - l resides on some region that represents only regions (but at least one) that in the caller, are separate to the region of l' in the caller.
  - l resides on some region that is fresh.
- In the case where both labels are untouched by the callee, their respective regions must be separate in the caller.
- In the case where both labels are manipulated by the callee, their respective regions must be separate in the callee.

$$\forall l, l' \in \mathit{updatedLabels}(a_c, \mathit{rout}, a_i), l \neq l' :$$

$$l \overset{\mathrm{R}}{\Longleftrightarrow}_{a_u} l' \Longleftarrow \begin{cases} \left(\left|\mathit{reps}_{a_i}(\mathit{regOf}_{a_i}(\mathit{exp}_a(\mathit{rout}, l)))\right| > 0 \wedge \forall r \in \mathit{reps}_{a_i}(\mathit{regOf}_{a_i}(\mathit{exp}_a(\mathit{rout}, l))) : \\ \quad \mathit{import}_{a_c}(\mathit{rout}, r) \overset{\mathrm{R}}{\Longleftrightarrow}_{a_c} \mathit{regOf}_{a_c}(l')\right) \\ \vee \\ (\mathit{fresh}_{a_i}(\mathit{regOf}_{a_i}(\mathit{exp}_a(\mathit{rout}, l)))\wedge \\ \qquad \text{if } \mathit{exp}_{a_c}(\mathit{rout}, l) \neq \mathit{undefined} \wedge \mathit{exp}_{a_c}(\mathit{rout}, l') = \mathit{undefined} \\ l \overset{\mathrm{R}}{\Longleftrightarrow}_{a_c} l' \\ \qquad \text{if } \mathit{exp}_a(\mathit{rout}, l) = \mathit{undefined} = \mathit{exp}_a(\mathit{rout}, l') \\ \mathit{exp}_a(\mathit{rout}, l) \overset{\mathrm{R}}{\Longleftrightarrow}_{a_i} \mathit{exp}_a(\mathit{rout}, l') \\ \qquad \text{if } \mathit{exp}_a(\mathit{rout}, l) \neq \mathit{undefined} \neq \mathit{exp}_a(\mathit{rout}, l') \end{cases}$$

Finally, we specify those cases where the regions of the two labels are equal.

- In the case where one label (l') is untouched by the callee and the other (l) is manipulated by the callee, the following must hold: l resides on some abstract region that represents exactly one region that in the caller, is the same as the region of l'.
- In the case where both labels are untouched by the callee, their respective regions must be equal in the caller.
- In the case where both labels are manipulated by the callee, their respective regions must be equal in the callee.

$$\forall l, l' \in \mathit{updatedLabels}(a_c, \mathit{rout}, a_i), l \neq l' :$$

$$l \overset{\mathrm{R}}{=\!=}_{a_u} l' \Longleftarrow \begin{cases} \exists r : \mathit{reps}_{a_i}(\mathit{regOf}_{a_i}(\mathit{exp}_a(\mathit{rout}, l)) = r \wedge \mathit{import}_{a_c}(\mathit{rout}, r) = \mathit{regOf}_{a_c}(l') \\ \qquad \text{if } \mathit{exp}_{a_c}(\mathit{rout}, l) \neq \mathit{undefined} \wedge \mathit{exp}_{a_c}(\mathit{rout}, l') = \mathit{undefined} \\ l \overset{\mathrm{R}}{=\!=}_{a_c} l' \\ \qquad \text{if } \mathit{exp}_a(\mathit{rout}, l) = \mathit{undefined} = \mathit{exp}_a(\mathit{rout}, l') \\ \mathit{exp}_a(\mathit{rout}, l) \overset{\mathrm{R}}{=\!=}_{a_i} \mathit{exp}_a(\mathit{rout}, l') \\ \qquad \text{if } \mathit{exp}_a(\mathit{rout}, l) \neq \mathit{undefined} \neq \mathit{exp}_a(\mathit{rout}, l') \end{cases}$$

Concerning the properties, the following can be said about an abstract region of some label in the caller:

- If the label is untouched by the callee, its region's properties remain untouched as well.
- If the label is manipulated by the callee, its region's properties are defined as follows:
  - If the region in the callee is fresh, the corresponding region in the caller is fresh as well.
  - If the region in the callee is leaked, the corresponding region in the caller is leaked as well.

$$
\begin{aligned}
&\forall l \in \mathit{updatedLabels}(a_c, \mathit{rout}, a_i):\\
\mathit{exp}_{a_c}(\mathit{rout}, l) = \mathit{undefined} &\implies (a_u)_{\mathrm{props}}[\mathit{regOf}_{a_c}(l)] = (a_c)_{\mathrm{props}}[\mathit{regOf}_{a_c}(l)]\\
\mathit{exp}_{a_c}(\mathit{rout}, l) \neq \mathit{undefined} &\implies (\\
&\quad \mathit{fresh}_{a_i}(\mathit{regOf}_{a_i}(\mathit{exp}_a(\mathit{rout}, l))) \implies \mathit{fresh}_{a_u}(\mathit{regOf}_{a_u}(l)) \wedge\\
&\quad \neg\mathit{fresh}_{a_i}(\mathit{regOf}_{a_i}(\mathit{exp}_a(\mathit{rout}, l))) \implies\\
&\qquad \mathit{fresh}_{a_u}(\mathit{regOf}_{a_u}(l)) = \mathit{fresh}_{a_c}(\mathit{regOf}_{a_c}(l)) \wedge\\
&\quad \mathit{leaked}_{a_i}(\mathit{regOf}_{a_i}(\mathit{exp}_a(\mathit{rout}, l))) \implies \mathit{leaked}_{a_u}(\mathit{regOf}_{a_u}(l)) \wedge\\
&\quad \neg\mathit{leaked}_{a_i}(\mathit{regOf}_{a_i}(\mathit{exp}_a(\mathit{rout}, l))) \implies\\
&\qquad \mathit{leaked}_{a_u}(\mathit{regOf}_{a_u}(l)) = \mathit{leaked}_{a_c}(\mathit{regOf}_{a_c}(l))\\
&)
\end{aligned}
$$

Finally, we need to specify the representations of all the regions in the caller defined so far. So for each label's region in the caller:

- If the label is untouched by the callee, the existing representations remain.
- If the label is manipulated by the callee, its region's representation consists of the union of all representations of all the regions in the caller that are represented by the label's region in the callee.

$$
\begin{aligned}
&\forall l \in \mathit{updatedLabels}(a_c, \mathit{rout}, a_i):\\
\mathit{exp}_a(\mathit{rout}, l) = \mathit{undefined} &\implies (a_u)_{\mathrm{rep}}[\mathit{regOf}_{a_c}(l)] = (a_c)_{\mathrm{rep}}[\mathit{regOf}_{a_c}(l)]\\
\mathit{exp}_a(\mathit{rout}, l) \neq \mathit{undefined} &\implies\\
\mathit{reps}_{a_u}(\mathit{regOf}_{a_u}(l)) = &\bigcup_{r' \in \mathit{reps}_{a_i}(\mathit{regOf}_{a_i}(\mathit{exp}_a(\mathit{rout}, l)))} \mathit{reps}_{a_c}(\mathit{import}_{a_c}(\mathit{rout}, r'))
\end{aligned}
$$

Chapter 5

Implementation

In this chapter, we are taking the theoretical basis developed so far as the groundwork for an actual implementation in the research branch of EiffelStudio (EVE in short). Since the implementation has to work for the full-blown Eiffel language, several features need to be supported in addition to what we used in our simplified SCOOP language before.

Besides the increased feature set, the implementation also supports a simple form of deadlock detection.

5.1 Abstract Interpretation Framework in Practice

When specifying the *embellished monotone framework* for the region analysis in chapter 4, we mentioned the program *flow*, which connects different program *labels* in the source code. We use transfer functions for every single statement and expression, even when they are nested. This means that we have a lot of program labels in our framework. The elements of our abstract domain – the state graphs – do not have a very economical representation in memory, no matter how much we are able to compress them.

In terms of the space complexity, it would be a rather bad choice to store state graphs at each program label of every routine (under all encountered contexts) in the system. Luckily, we do not have to either. The interesting kind of information are not the state graphs, but rather the *classifications* (see section 4.7) of routine calls, for which state graphs are used.

5.1.1 From a System of Equations to a Program

In principle, every monotone framework is nothing more than a huge system of equations. As such, it does not know a beginning or end. We are just interested in a solution that satisfies the system at every program label. However, when we are supposed to write a program that actually solves these equations, different techniques come into play. Among them are *chaotic iteration*[1] or the *worklist algorithm*[2]. But these approaches rely on the existence of a flow graph that allows the storage of state graphs (in our case) at every program label, which we have just ruled out due to performance concerns.

Instead, we directly traverse the *abstract syntax tree* (AST in short) of a routine's source code. The region analysis is an abstract interpretation that is considered a forward analysis,

[1]Apply transfer functions randomly at each program label, until no more changes can be observed.

[2]Maintaining a pool of program labels for which transfer functions need to be recomputed.

because information travels forward in the source code order. Thus, a natural choice for the traversal order is *post-order*. As an introductory example, we consider a simple `if-then-else` branch as shown in figure 5.1.

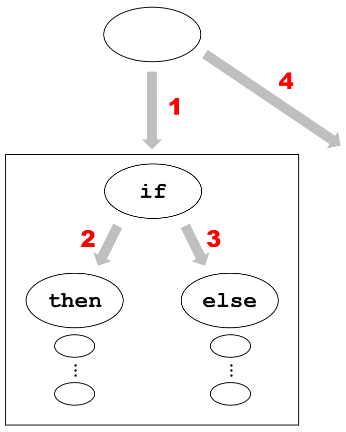


Figure 5.1: *AST of a simple branch instruction*

The ellipses represent AST nodes, and the boxed area is the subtree that represents the branch.

1. First, we send the current state graph along edge #1. At the `if` node, we interpret the condition expression on it.
2. Then, we create a copy of the state graph and remember it.
3. Next, we send the original state graph along edge #2, where all the statements in the `then` branch get interpreted.
4. The state graph that results in the left branch will be remembered at the `if` node as well.
5. Then, we send the previously remembered copy of the original state graph along edge #3, where all statements in the `else` branch get interpreted.
6. Finally, we combine the resulting state graph of the `else` branch with the remembered state graph of the `if` branch with the *least upper bound* operation, which yields a new state graph.
7. This new state graph represents the effects of the whole branch, and will be used in the continuation, i.e. it will be sent along edge #4, to whatever AST node follows the branch.

This procedure perfectly reflects the mathematical specification of the branch instruction, but it does not leave any traces in the memory at all. We may temporarily use several copies of state graphs, but we delete them as soon as they are no longer needed. To put it another way: We can always delete state graphs that are known to be part of the final solution of the underlying equation system. We have to remember them temporarily whenever we know that at some later point they get updated. This is all possible in one single AST traversal per routine and context.

Notice that we will not further explain how the other transfer functions are implemented, because they all follow the same principle as in this example.

5.1.2 Contexts

For each routine in the system, we store a map – called the *context map* – that associates context state graphs with the final state graph obtained by interpreting the routine under this context. In particular:

- Whenever we start the interpretation of a routine under a new context, we put the context as a key into the map, but without the associated value (that is, we store `Void`).
- Whenever the interpretation for a given context finishes, we add the resulting state graph as the value into the map.
- Whenever the interpretation tries to call a routine under some context for which a result was previously computed, it can directly use that result and continue with the return transfer function.

5.1.3 Recursive Calls

In section 4.5.3, we briefly hinted at a delicate problem regarding recursive calls under the same context. Figure 5.2 shows the simplest instance of a recursive call in our abstract interpretation framework.

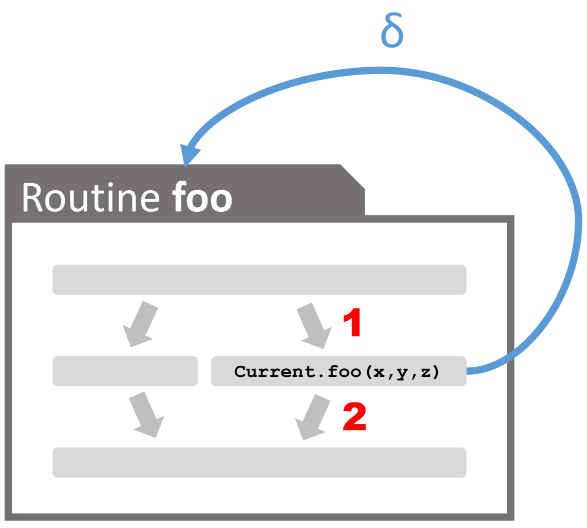


Figure 5.2: *a recursive self-call of routine* `foo`

Let us assume that we enter routine `foo` in context δ, and then perform the transfer function at #1. Then, we encounter a recursive call to routine `foo` under the very same context δ. It is possible to apply the callsite transfer function, but applying it will bring us into the exact same situation as before. The interpretation can only continue if it can apply the return transfer function before continuing with #2.

Remember that in routine `foo`, there has to be *some path* that bypasses the recursive call. Otherwise, the program would never terminate. One solution therefore would be to first identify this path in advance, compute an analysis result just for this path, and use it in the return function at the call. However, this would not yet yield a safe approximation. The analysis would have to further repeat this strategy until it reaches a fixpoint right after the return transfer function. But since all of our transfer functions are monotone, such a fixpoint exists (just like when interpreting a loop construct), and the result would indeed be the most precise safe approximation.

However, we did not implement this approach, due to a very simple reason. In this basic example that we have just seen, it is easy to find the path that does not involve the recursive call. However, in a real program, there may be arbitrary long indirect recursive calls, and even multiple of them in a single routine. It would require a special preprocessing phase just to identify all potentially recursive routines, i.e. routines that are part of a cycle in a directed routine call graph, then for each such cycle we would have to find some non-recursive paths on which we could start the fixpoint iteration scheme. This is not impossible, but it would break the simple forward interpretation algorithm and would further decrease performance. Nevertheless, it is an option for future improvements.

The present solution in our implementation makes use of default state graphs (see section 4.2) that are retrieved from the recursively called routine's signature. These graphs represent the most general solution for a routine (i.e. the top element of the complete lattice of the corresponding viewpoint-adapted abstract domain), and are safe approximations, although not very precise. Recursive calls are identified when the AST traversal tries to jump from a routine call to the called routine's body AST, and realizes that for this context, a computation was started but not finished, that is, there is a key for this context in the routine's *context map*, but the associated value is `Void`. Then it can conclude that the current call must be part of this computation, and in particular, that it must be recursive for this context.

5.2 Enhanced Region Analysis

The region analysis as described in the theoretical part of this thesis worked on a simplified SCOOP language, as defined in chapter 2. This let us formalize it much more concisely, and it was enough to explain the key workings of the analysis. However, for the implementation, the region analysis has to work on the full-blown Eiffel language. We therefore need to consider many more features like the following:

`once` routines Routines that are executed only once *per processor* the first time they are called, are treated like recursive calls by our implementation. This is because we can not determine what the first call is. This highly depends on the concrete state at runtime. Recursive calls directly return with a default state graph. And since default state graphs represent the most general state, they are the only safe option to use.

Type system based on multiple inheritance and generics In our simplified language, we do not allow any kind of inheritance. But the real Eiffel language has a sophisticated type system. Surprisingly enough, we do not have to make many changes to the analysis. The only moment where types actually play a role is in the callsite transfer function. There, we copy the attribute labels of some passed controlled label only if the passed controlled label cannot be an alias of any of the other controlled labels of the caller. Labels can only be aliased if their types conform to each other, one way or the other. Therefore, we only have to refine the notion of type conformance when deciding this question for real Eiffel programs.

Dynamic binding Dynamic binding is dealt with by first determining, for each call in the source code, all routines that may be dynamically called at runtime. This can be done for each routine in a preprocessing stage. Then, the callsite transfer function is modified in such a way, that it calls every possible routine in isolation, and then combines all the resulting state graphs with the least upper bound operation. This is the only way to do it, and the precision suffers quite a bit from it.

Renamed attribute names Another problem is posed by the renaming mechanism of Eiffel, in combination with dynamic binding. Since we are using attribute labels in state graphs, we have to uniquely name them. So when combining multiple state graphs that resulted from several dynamically callable routines, not all attributes that are actually equal may also be equally named. This problem is overcome by determining, for each attribute, the name it had when it was first introduced in the class hierarchy. In the case where it was introduced in multiple classes and later merged, we take the name of the first of these classes as dictated by the lexical ordering of their names.

Expanded types Expanded types in Eiffel cover simple types such as integers and characters. But it is also possible for programmers to define their own expanded types. Since objects of these types are always copied when assigned, care must be taken in the region analysis to reflect this behavior properly.

5.3 Integration into EVE

The region analysis is implemented as an Eiffel library project that itself depends on existing libraries of EVE, such as the compiler kernel, or parts of the general program analysis framework developed by Stefan Zurfluh in [14]. The data structures provided by the compiler exist on several levels of the compilation process, and contain all information that is required by the region analysis. The region analysis needs a fully and successfully compiled SCOOP project in order to work properly.

The *region analysis library* consists of these clusters:

Framework Contains classes related to the overall abstract interpretation framework, like light-weight representations of routines, contexts and classifications.

Interface Contains classes that heavily cooperate with existing data structures of the EVE compiler, like the AST visitor (including many of the routine-local transfer functions) and the overall controller of the region analysis.

State Graph Contains the classes like regions, labels and maps, that together comprise state graphs.

Transfer Functions Contains the callsite and return transfer functions and the least upper bound operation.

User Refinement Contains the operations required to refine state graphs.

The analysis can be started either on the system's root procedure, or on any user-specified routine. It starts with a default context graph, and runs until convergence. It stores routine classifications in a map that associates them with AST nodes, such that they are easily queried after the analysis completed. At any time after the initial run-trough, a new context state graph associated to any of the routines can be provided, and the analysis incrementally runs again until convergence. This is useful for the parallelism visualizer that enables user to manually refine the analysis.

Stopping Criteria The region analysis' is most useful on programs that include lots of separate entities and calls. However, there are many sequential libraries like *EiffelBase*[3]. Whenever a given program calls into one of these libraries, a potentially large chain of nested sequential calls may follow. One example is the call of the `print` feature, which is declared in class `ANY`. The region analysis may take a long time to examine all of these calls, and yet, the benefit is non-existing, since everything is sequential anyway.

There exist sophisticated approaches to a special treatment of large libraries in program analyses, like [10]. For our purposes, we opted for a simple and flexible solution: We integrated an upper bound on the number of completely sequential routine calls, i.e. static non-separate calls with only non-separately declared formal arguments. Once this upper bound is reached, the region analysis stops applying the callsite transfer function and instead directly uses the default return transfer function, which examines that static signature of the called routine only. There is always a chance that deeper in a sequential call chain there are again separate calls, but mostly, this is a harmless limitation.

As a final resort, the user will be able to pick any routine and restart the analysis with a default context graph anyway.

5.4 Data Structures of State Graphs

The core data type used by the region analysis' implementation is the state graph, together with core operations such as *identifier extraction* and *least upper bound*. In this section, we focus on these, although there are quite a few related operations such as the callsite and return transfer functions, and the classification generation.

Let us begin with a state graph's data structure. The most important goal in designing this data structure is to keep it's memory footprint as small as possible. Although we are already reusing a given state graph instance during the interpretation, there are still quite a few *copy and remember* operations involved. Another important goal is to identify the required operations and their expected frequency in the algorithms that use them. The following table lists the ones with the highest frequencies.

Operation	**Description**
Identifier generation	Whenever we are comparing state graphs e.g. during fixpoint computations or when we check for already existing contexts in the callsite transfer function, we have to efficiently determine if two given state graphs are equal or not. For this purpose, we shall be able to compute a unique identifier for each state graph.
CRUD[4] region separateness	Given a single abstract region of some state graph, we should be able to quickly determine all regions that are separated from it. Likewise, given two distinct abstract regions, we want to now if they are separated. Adding and deleting separating edges should also be fast.
CRUD labels and region access	Quickly changing the attachment of an attribute or local label to some abstract region must be a priority, as well as adding and deleting them on the fly. Furthermore, accessing a label's abstract region should be easy.

We chose to use the following sub structures to satisfy the established requirements.

Regions & Region IDs In the mathematical model, we identify regions implicitly via their associated labels. Since a region must at least have one label attached in order to exist,

[3]See `https://docs.eiffel.com/book/solutions/eiffelbase`, as retrieved on 2014/12/16

[4]**C**reate, **R**ead, **U**pdate, **D**elete

this approach is legitimate. This way, we also set the stage for cleaner mathematical descriptions of the transfer functions and the least upper bound operation. In the implementation however, such a representation is neither intuitive nor efficient. Instead, regions are represented by their own object type.
Since regions may be shared internally (by more than one label) and even among different state graphs (during the callsite function for example), they have to be easily identifiable. One possibility would be to just use and identify them as object references. However, this solution – although feasible – is quite cumbersome to work with (e.g. for debugging purposes), since objects are dynamically allocated on the heap, and might have different addresses at different times. A much more explicit solution is to use self-made identifiers for regions, like canonically ordered natural numbers. The association between these region identifiers and the region objects themselves is established via a hash map called the *region map*, which is the pivotal constituent of every state graph.

Region properties The region objects additionally have private properties like boolean flags for *locked*, *fresh* and *leaked*, a set (implemented as a sorted linked list) that contains the represented region IDs, and a sorted list of region IDs to which it is considered separate. Note that this separated list is symmetric, i.e. for every entry y in the list of region x, there has to be an entry x in the list of region y. Sorted lists are preferred since it is faster to query, and there is a fixed order that simplifies identifier generation (as described shorty).

Labels We implement controlled labels as an arrayed list, in order to be able to directly access a controlled label with its index in a routine's signature, where the first index is always occupied by the `Current` label. A controlled label itself is an object that stores a region ID (of the region it is attached to) and contains a private hash map that associates attribute names with region IDs as well. This way, we can reach attribute labels directly via their controlled labels, which is the only kind of access we require. Lastly, local variables are modeled just like controlled attributes, that is, by a single hash map that associates local variable names with region IDs.

Figure 5.3 shows an illustration of the main components of a state graph's representation in memory.

5.4.1 State Graph Identifiers

On a logical level, the only thing that distinguishes two distinct regions are still the labels that are associated with it. But in the implementation, we model regions with their own identifiers. So we may run into ambiguity problems, if we do not make sure the IDs we assign don't change a state graph's logical identity. [3] discusses the general problem of object identity and inspired our choices.

Note that region IDs may not even be densely seeded, since the transfer functions and the least upper bound operation may arbitrarily add new abstract regions or remove existing ones. Instead of keeping a pool of free IDs to choose from, we keep a global counter for each state graph that is fetched and incremented whenever an operation needs a new region ID. This approach has the advantage of being fast and simple.

The problem is addressed by computing an identifier (which is simply a string) for a state graph that internally re-identifies all the regions with a strictly and densely increasing set of region IDs starting from 1. It does so by following a deterministic order. Note that the computation of this identifier is completely side-effect free and leaves no traces on the

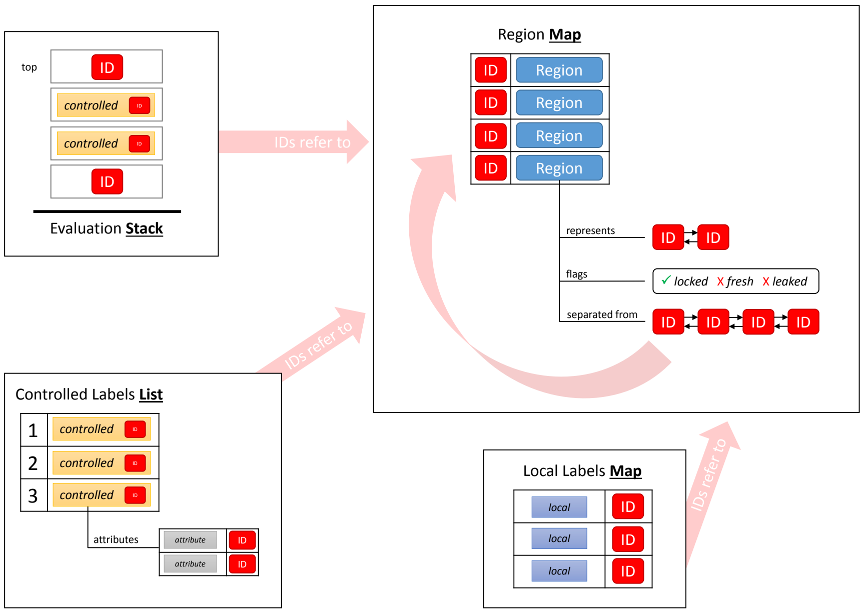


Figure 5.3: *qualitative representation of some state graph in memory*

state graph itself. This way, we compare state graphs whenever we need, by computing their identifiers and comparing them on string equality.

Algorithm 1 outlines the generation of the identifier in pseudo code. The function SIGNATURE takes as input a state graph a, and produces as output the signature string s. The central idea is to traverse all labels in the order given by their positions (controlled labels) or names (attributes and locals), and output a region's properties whenever we first encounter it. Regions are marked with a constantly increasing temporary ID $tmp\_id > 0$ that will also be used to identify the region in the identifier.

This is how the identifier of the state graph of figure 3.1 looks like:

'Current@1K;.attr_bar@2;.attr_foo@3;a_baz@4K|1;.attr_bar@4;a_fez@1;.attr_foo@2;l_bar@5F|1|2|3|4;l_foo@5;Result@3;'

Notice that the identifier ignores the evaluation stack entirely. This is because whenever we require a state graph comparison, the evaluation should be empty anyway.

5.4.2 Callsite and Return Transfer Functions

In our mathematical description, the callsite and return transfer functions are specified completely independently of each other. However, in the implementation, we use various intermediate data structures that are created and stored while executing the callsite function, and reused again by the following return transfer function (which will follow in any case). These data structures primarily consist of hash maps that associate labels and regions in the caller with labels and regions in the callee. This way, the return function can easily inject parts of the callee's resulting state graph back into the state graph of the caller at the right places.

Algorithm 1 Signature extraction

```
global $counter = 0$

function SIGNATURE(in: $a$, out: $s$)
    $s \leftarrow ''$
    for all $c \in a.controlledLabels$ do
        $s \leftarrow s + $ REGIONSIGNATURE($c, regionOf(c.region\_id)$)
        for all $attr \in sorted(c.attributes)$ do
            $s \leftarrow s + '.' + $ REGIONSIGNATURE($attr, regionOf(attr.region\_id)$)
    for all $l \in a.localLabels$ do
        $s \leftarrow s + $ REGIONSIGNATURE($regionOf(l)$)


function REGIONSIGNATURE(in: $label, r$, out: $s$)
    local $r'$

    $s \leftarrow label + '@'$
    if $r.tmp\_id = 0$ then
        $r.tmp\_id \leftarrow counter$
        $counter \leftarrow counter + 1$
        $s \leftarrow s + r.tmp\_id$
        if $r.locked$ then
            $s \leftarrow s + 'K'$
        if $r.fresh$ then
            $s \leftarrow s + 'F'$
        if $r.leaked$ then
            $s \leftarrow s + 'L'$
        for all $rid \in sorted(r.represents)$ do
            $r' \leftarrow regionOf(rid)$
            $s \leftarrow s + 'r' + rid.tmp\_id$
        for all $rid \in sorted(r.separated\_from)$ do
            $r' \leftarrow regionOf(rid)$
            if $r'.tmp\_id \neq 0$ then
                $s \leftarrow s + '|' + r'.tmp\_id$
    else
        $s \leftarrow s + r.tmp\_id$
    $s \leftarrow s + ';'$
```

5.5 State Graph Refinement

So far, both the theoretical description and the implementation-specific considerations of the region analysis do not assume any user interaction or guidance that may improve the precision. In the *parallelism visualizer* that we integrated into EVE, there is a mechanism that allows a user to *refine* a given state graph. However, a refinement can only be applied to an *initial* state graph of some routine, also known as a routine's context. There might be several of these contexts after the analysis completed. So the user may pick any of them, refine it, and re-execute the region analysis on this routine using the newly refined context. The analysis will then incrementally add new results to all affected routines, until it converges again.

Mathematically, a refinement is defined by traversing the complete lattice (see 4.4.2) of a given viewpoint-adapted abstract domain downwards (in terms of $\sqsubseteq$), starting from a state graph that ought to be refined. Conversely, we can define for a given state graph the set of all state graphs that are legal refinements for it.

Definition 5.5.1 (refinement set)**.**

$$\begin{aligned} \mathit{refinements}\colon A_{rout} &\to \mathcal{P}(A_{rout}) \\ \mathit{refinements}_a &\mapsto \{a_{\text{ref}} \mid a_{\text{ref}} \sqsubseteq a\} \end{aligned}$$

The mathematical description of a refinement set does not yet tell us how a refined state graph can be obtained in practice. Therefore, we have implemented a set of operations that can be applied to a given state graph such that after such an operation is completed, the modified state graph can only be a refinement of the original. The following table informally describes each of these operations.

Operation	**Description**
Merge regions	Given two abstract regions in the original state graph that are *not* connected with a separating edge, merge them into a new abstract region that contains the union of all labels and all outgoing separating edges of its constituents. Furthermore, the merged region is: • *locked* if and only if at least one of the constituents is, • *fresh* if and only if at least one of the constituents is, • *leaked* if and only if at least one of the constituents is.
Separate regions	Given two abstract regions in the original state graph that are *not* connected with a separating edge, connect them.
Lock region	Given an abstract region of the original state graph that is not locked, lock it.
Make fresh	Given an abstract region of the original state that is not fresh, make it fresh.
Remove leaked	Given an abstract region of the original state that is leaked, remove the leaked flag.
Add attribute label	Add a not yet existing attribute label of some existing controlled label to one of the state graph's abstract regions.

As we explain in chapter 6, not all of these operations are offered to the user, since they are not all equally intuitive and useful from a user's perspective.

5.6 Deadlock Hints

Although the region analysis is designed for an entirely different purpose, it is still possible to detect a certain kind of deadlock as an added bonus, by modifying the analysis only slightly. In general, deadlock detection is a rather involved task on its own. In [12], this topic is approached with a static solution that requires additional annotations made to the source code.

5.6.1 Blocked Regions

We enhance abstract regions with yet another flag called *blocked*. The meaning of a blocked region is, that there *must* exist some processor that is holding a lock on this region while waiting for the currently executing target routine to finish, and the lock has not been passed over. This situation is quite critical, because whenever the target routine issues a synchronous call to a blocked region, it will never make any progress due to a *circular waits-for relationship* between processors. Note that this is only one type of deadlock. There are others not covered by this strategy.

A blocked region cannot unblock during the interpretation of a routine.

Adapted callsite transfer function The callsite transfer function marks a region as blocked while it passes it to the callee, if and only if:

- The passed region is not the region the callee operates on (the region of `Current` in the callee), and:
  - The call is synchronous and the region is already blocked in the caller, or
  - the call does not involve lock-passing and the region is locked in the caller.

Detection of critical calls Whenever a synchronous call on a blocked region is encountered during the interpretation of the target routine's body, the analysis reports the detection of a deadlock.

Since this deadlock detection strategy produces *must* assertions, the programmer can easily identify the source of a deadlock, or even better, prevent it from occurring entirely. However, only a small amount of all potential deadlocks of this type are detected this way. The recall is not very high.

If we would slightly relax the above conditions for a region to get blocked, we could increase the number of deadlock alarms produced. For example, we could relax the condition that no lock-passing does happen while passing a region to the callee, to the condition that there *may* be no lock-passing involved. In this case, the analysis conservatively captures many situations where there is a possibility of a deadlock occurring. But at the same time it would become unsound (in addition to being incomplete anyway), because it may produce false alarms. This is why we persist on the sound version.

5.6.2 Advanced Approach

An advanced approach has been theoretically developed as part of this thesis. However, we did not implement it as it would introduce many new operations and data structures to our existing region analysis. Also, it was not the main task of the region analysis to detect deadlocks. A full specification as well as implementation may be part of future work.

Enhanced State Graphs We propose the addition of two new *directed* edges that connect abstract regions:

wait-now The presence of this edge means that the handler of the region at the start of the edge will at some point *synchronously* call a routine on the region at the end of the edge, and the handler of the target routine (i.e. the routine the state graph is associated with) holds locks for both of these regions. In this situation, we know that the handler of the region at the start already holds the lock it needs to issue a call.

wait-then The presence of this edge means that the handler of the region at the start of the edge will at some point *synchronously* call a routine on the region at the end of the edge, and the target routine does not hold a lock for the region at the end of the edge (so it may or may not hold a lock for the routine at the start). In this situation, we know that the handler of the region at the start must allocate a new private queue on the request queue of the region at the end, which will be granted at *some point in the future* with possible actions interleaving.

With these additions in place, we must adapt the callsite and return transfer functions.

Adapted Callsite Transfer Function If the issued call is synchronous, add a new *wait-now* edge from the abstract region with the `Current` label to the abstract region on which the target of the call resides, but only if both regions are different and separated.

Adapted Return Transfer Function Upon returning, we add all *wait-now* and *wait-then* edges from the callee's resulting state graph to the state graph of the caller, as long as both abstract regions are also known to the caller. For the *wait-now* edges, we additionally do: If the region at the end of the edge is not locked, we turn a *wait-now* edge into a *wait-then* edge. Then, for all newly added edges and the existing *wait-then* edges in the caller's state graph, we check if they form a cycle that includes at least one newly added and one existing edge. If so, we may have detected a deadlock and can report it.

The reason why this work is because the calls represented by the *wait-then* edges may be overtaken by many other calls, including the ones represented by other *wait-then* or *wait-now* edges. So depending on the actual scheduling of these calls, the detected cycle may indeed be a true interdependency that leads to the deadlock. The reason why we do not let the *wait-now* edges of the caller be part of a cycle is because we know that all of these calls must happen before the ones collected by the callee and therefore can not be part of a true interdependency. These edges are again meaningful when the caller itself returns to *its* own caller.

Since this is possible future work, we will not go into all the detail of these transfer functions or how the least upper bound operation must be adapted.

Chapter 6

Parallelism Visualizer

The implementation of the region analysis forms the basis of the *parallelism visualizer*, which is a visual tool integrated into the research branch of EiffelStudio, also known as EVE. This tool consists of a widget and a pop-up and is seamlessly integrated into the user experience of EiffelStudio.

6.1 Goals

The goal is four-fold:

1. Provide the user with feedback about the possible **region topologies** encountered at runtime for a given routine.
2. Visually **classify routine calls** inside a given routine according to lock-passing, asynchrony and deadlocks, to provide the user with deeper knowledge about the parallel nature of the program.
3. Let the user **navigate** through the program by following routine calls (referred to as *drilldown*) and switching different contexts for a given routine.
4. Let the user influence the analysis results by **refining** a chosen context state graph with a number of modifications.

6.2 Design

Navigation in EiffelStudio is a key element. There already exists a designated panel that offers an interactive view of a selected routine's body. Since our analysis offers results also on a routine-level, we used a similar panel that shows a routine's body with special formatting.

We chose to use differently colored formal argument names (including `Current`) to indicate the regions these entities reside on (see figure 6.1 for more details). If two entities have the same color, they must reside on the same concrete region. If they have different colors, they may be either on the same region, or on different ones. The user can open a pop-up that will show which regions (represented as colors) are known to be separated from each other.

We do not color attribute names or local variable names, because the regions they are assigned to may change over the course of the routine body, and as such, would force the region analysis to store state graphs at almost every program point, which is a drastic performance decrease. Also, too many colors on the screen may distract the user from the essential information.

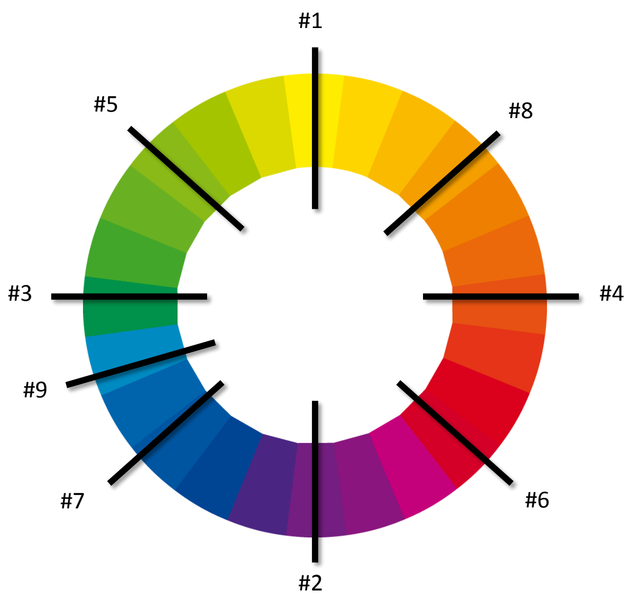


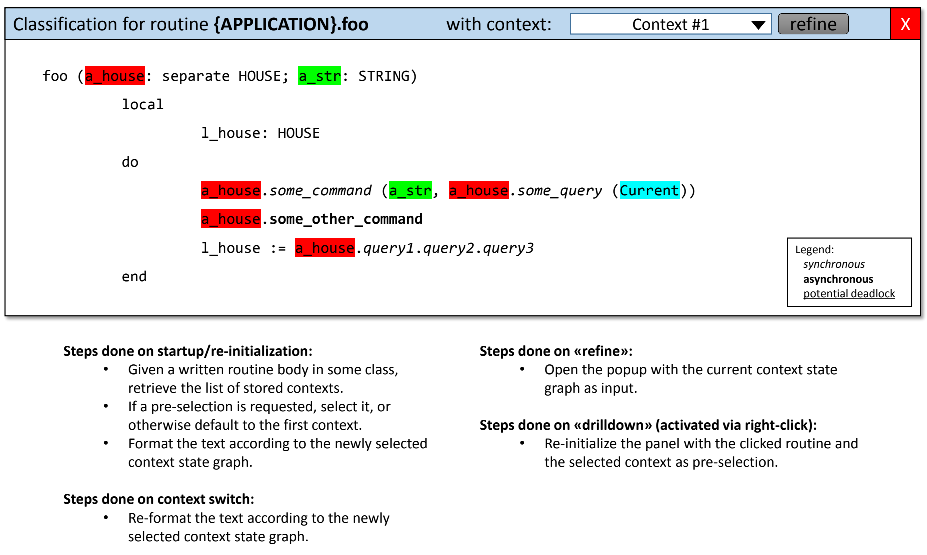
Figure 6.1: *Colors are generated via the color wheel*[1] *in such a way that a newly added color is as visually different to all the colors before. In addition to the hue of the color, we also varied the saturation and lightness. The algorithm works for an arbitrary number of colors. In practice however, not that many are simultaneously used.*

The routine calls inside the currently viewed routine's body are visually classified by using different font styles. However, only the information about asynchrony is displayed like this, because it is the most useful. Via a right-click on a routine name, the user is presented with the remaining information about lock-passing, and an option to drilldown, which switches the panel's currently viewed routine to the called one. Already assigned colors in the original routine are taken over to the new routine if they do indeed represent the same concrete region.

Figure 6.2 shows the conception of the panel and the actions that may be applied to it.

The user can either switch a selected context, display it as a real graph (see figure 6.9), or refine it. The last option opens a pop-up that shows the separateness of the regions and offers actions for manipulation. Note that not all actions as described in section 5.5 are offered, due to a lack of usefulness and intuition. Figure 6.3 illustrates the conception.

[1]Image retrieved from
`http://www.thoth-adan.com/de/beitrag/die-welt-der-farben.html`,
as per 2014/12/11



```
foo (a_house: separate HOUSE; a_str: STRING)
        local
                l_house: HOUSE
        do
                a_house.some_command (a_str, a_house.some_query (Current))
                a_house.some_other_command
                l_house := a_house.query1.query2.query3
        end
```

Steps done on startup/re-initialization:

- Given a written routine body in some class, retrieve the list of stored contexts.
- If a pre-selection is requested, select it, or otherwise default to the first context.
- Format the text according to the newly selected context state graph.

Steps done on context switch:

- Re-format the text according to the newly selected context state graph.

Steps done on «refine»:

- Open the popup with the current context state graph as input.

Steps done on «drilldown» (activated via right-click):

- Re-initialize the panel with the clicked routine and the selected context as pre-selection.

Figure 6.2: *conception of the parallelism visualizer's panel*

6.3 Implementation

Since the parallelism visualizer is an add-on for EVE, it is built on top of the existing architecture, which in turn makes heavy use of the *Eiffel Vision 2*[2] GUI library. Many components of the visualizer are directly programmed against the API of Eiffel Vision, which is well documented and easy to use.

The visualization of a state graph as a real graph makes use of the *DOT* program, which is part of the freely available *graphviz*[3] package. DOT turns a plain text graph description into a rendered image with optimized layout of nodes and edges. Figure 6.9 shows two examples.

6.4 Sample Usage

To demonstrate the parallelism visualizer's usefulness, we go over a small real-world example. Listings 6.1 and 6.2 contain the source code of a basic SCOOP program, and figure 6.4 shows the heap topology thereof.

In the parallel visualizer's panel, we can click on the "kick on" button, which identifies the root procedure – feature `{APPLICATION}.make` – and starts the region analysis on it using a default state graph as the context. Figure 6.5 shows the state of the panel right after the region analysis converged. By right clicking on the `setup` feature call name, the user can choose to "drilldown" in order to show `setup`'s body under the context given by the call. In the menu, it can also be seen that this call is synchronous and causes lock-passing, which is the case for any non-separate call.

In figure 6.6, we can see the routine `setup`. The formal argument `a_logger` is highlighted in green, indicating the region it is located on. Note that in figure 6.4, we used the same colors as the ones used by the tool.

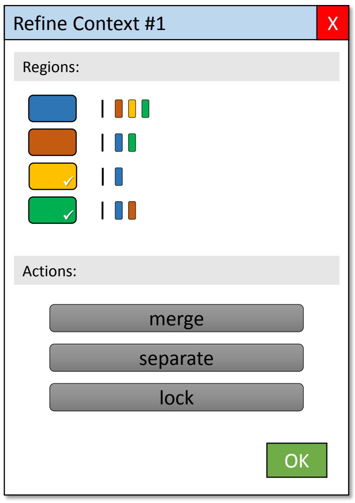
[2]See `https://docs.eiffel.com/book/solutions/eiffelvision-2`, as retrieved on 2014/12/16
[3]See `http://www.graphviz.org/`, as retrieved on 2014/12/16

Listing 6.1: Eiffel*: real-world SCOOP sample program / part 1*

```
class
  APPLICATION

inherit
  ARGUMENTS

create
  make

feature
  make
    do
      create logger
      setup (logger)
    end

  setup (a_logger: separate LOGGER)
      -- Initialize the logger and start the cooking.
    local
      l_cook: separate COOK
    do
      a_logger.init
      create l_cook
      run (l_cook)
    end

  run (a_cook: separate COOK)
      -- Prepare ingredients and let the cook use them.
    local
      l_spaghetti: SPAGHETTI
      l_sauce: SAUCE
    do
      create l_spaghetti
      create l_sauce
      a_cook.work (l_spaghetti, l_sauce, logger)
    end

feature
  logger: separate LOGGER

end
```

Listing 6.2: Eiffel: *real-world SCOOP sample program / part 2*

```
class
  LOGGER

feature
  init do
    -- Initialize
  end

  log do
    -- Log something.
  end
end

...

class
  COOK

feature
  work (a_spaghetti: separate SPAGHETTI; a_sauce: separate SAUCE;
      a_logger: separate LOGGER)
    do
      -- Cook the meal.
      a_logger.log
    end

end

...

class SAUCE end

...

class SPAGHETTI end
```



Steps done on startup:

- Make a fresh copy of the given context state graph.
- Initialize the «Regions» view.

Steps done on «X»:

- Close the popup and do nothing else.

Steps done on «OK»:

- Save the resulting state graph as new context, if not already present.
- If a new context was created, run the region analysis with it again (do not drop existing results).
- Close the popup, re-initialize the panel with the resulting context as pre-selection.

Steps done on «merge»:

- If exactly two regions are selected, merge them and re-initialize the «Regions» view.

Steps done on «separate»:

- If exactly two regions are selected, separate them and re-initialize the «Regions» view.

Steps done on «lock»:

- Mark each selected region locked.

Figure 6.3: *conception of the pop-up, where the user can select colored regions and an action to perform on them*

Drilling down into feature call `run`, the user sees the panel as shown in figure 6.7, where another color (blue) is used for the region `a_cook` is located on. The interesting part here is the procedure call `a_cook.work`, which, from a static point of view, is separate and does not involve any lock-passing. However, the tool gives the insight that the call is indeed synchronous because of some hidden lock-passing going on. The actual argument `logger` – which is an attribute of `APPLICATION` – was already locked by the processor executing routines `make` and `setup`. By viewing the body of `run` in isolation, this situation is not identifiable. The region analysis helps in this case, because it works inter-procedurally.

Finally, if the user drills down into `work`, he can see that both separate arguments are – in this particular context – located on the same region, which again is not statically determinable by just looking at the source code of routine `work`.

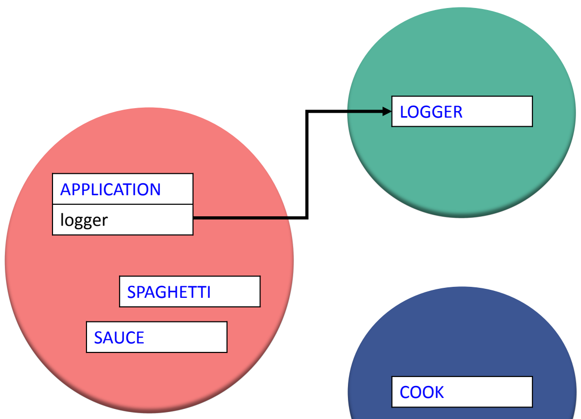
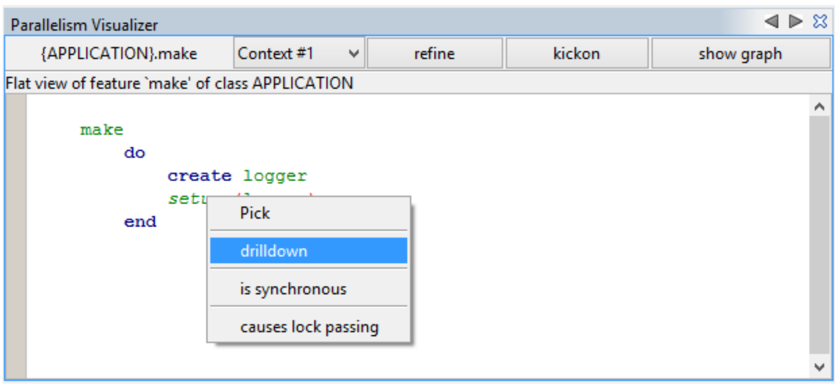


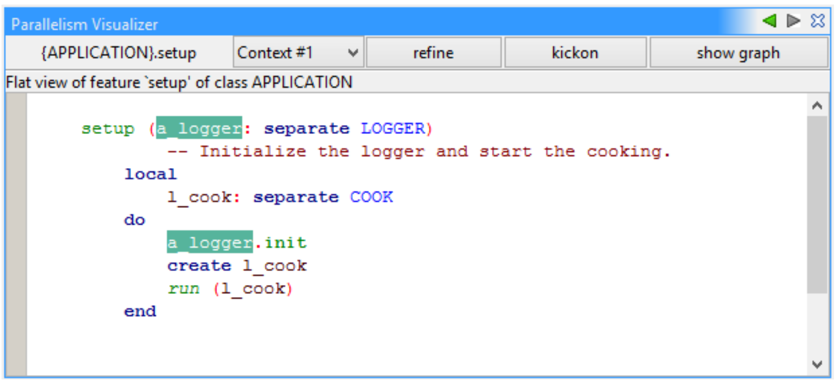
Figure 6.4: *runtime heap topology of a real-world SCOOP program*



Flat view of feature `make' of class APPLICATION

```
make
    do
        create logger
        setu
    end
```

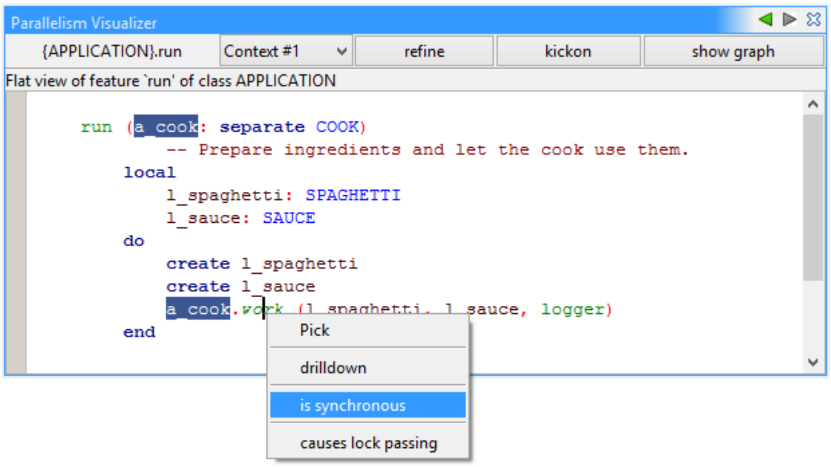
Figure 6.5: *The user can follow routine calls by choosing "drilldown" in the context-menu.*



Flat view of feature `setup' of class APPLICATION

```
setup (a_logger: separate LOGGER)
        -- Initialize the logger and start the cooking.
    local
        l_cook: separate COOK
    do
        a_logger.init
        create l_cook
        run (l_cook)
    end
```

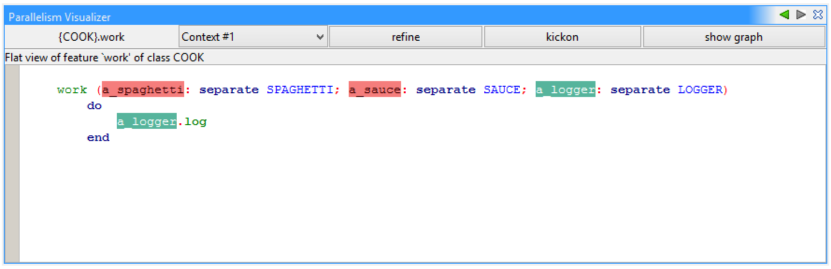
Figure 6.6: *Only formal argument names and* `Current` *are highlighted with a region color, mainly because they remain constant throughout the routine's body.*



Flat view of feature `run' of class APPLICATION

```
    run (a_cook: separate COOK)
            -- Prepare ingredients and let the cook use them.
        local
            l_spaghetti: SPAGHETTI
            l_sauce: SAUCE
        do
            create l_spaghetti
            create l_sauce
            a_cook.work (l_spaghetti, l_sauce, logger)
        end
```

Figure 6.7: The selected separate call, `work`*, is found the be synchronous, because it triggers lock-passing.*



Flat view of feature `work' of class COOK

```
    work (a_spaghetti: separate SPAGHETTI; a_sauce: separate SAUCE; a_logger: separate LOGGER)
        do
            a_logger.log
        end
```

Figure 6.8: The first two formal arguments are located on the same region (colored red) in this particular context (#1).

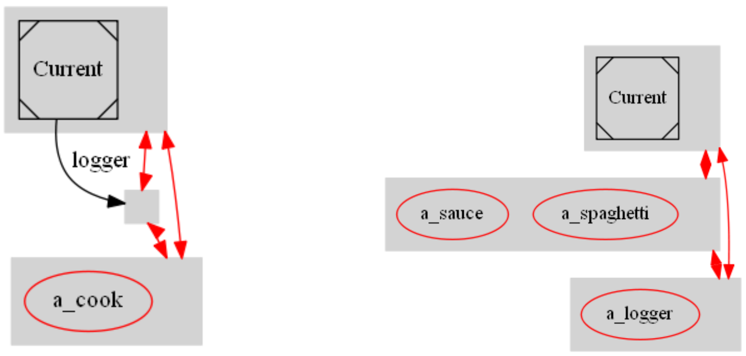


Figure 6.9: left: context state graph with which routine `run` *gets called; right: context state graph with which routine* `work` *gets called; These graphs show up in a pop-up when the user pressed the "show graph" button in the panel.*

Chapter 7

Conclusion

In this thesis, we have designed an original static analysis for SCOOP programs, which helps programmers get a better understanding of the parallel nature of their code. We specified the analysis formally and implemented it in EVE, the research branch of EiffelStudio. In addition, we developed a visual tool called *parallelism visualizer* in EVE that enables programmers to interact with the results of the analysis, and refine it if necessary.

7.1 Summary

SCOOP's type system alone is not powerful enough to provide a fine level of control over aspects like entity-to-processor mappings. Also, the runtime system does a rather obscure job of avoiding potential deadlocks via a mechanism known as lock-passing, which has the downside of reducing the degree of parallelism by turning asynchronous calls into synchronous ones. The goal of this thesis was to develop a solution to this problem in the form of an interactive visual tool that makes use of an underlying static code analysis.

As a first step, we specified a simplified version of the SCOOP programming language, in order to lay a simple but still realistic foundation for the formal description of the *region analysis*, an inter-procedural abstract interpretation following an embellished monotone framework.

We formalized the region analysis by first defining the concrete and abstract domains, the representation function, and the associated concretization and abstraction functions. We then introduced state graphs as contexts and developed the overall inter-procedural framework while carefully balancing performance and accuracy. We showed that the abstract domain is a complete lattice, and continued with the various transfer functions that specify what effects program expressions and statements have to the elements of the analysis.

Following the theoretical groundwork, we implemented the region analysis as a library in EVE, working not just on the simplified SCOOP language, but rather on the *real* language with many more features to cope with. In order to improve the overall performance, we designed the data type of state graphs by heavily deviating from the mathematical objects, in particular by giving abstract regions their own identity in the form of natural numbers that are used as a short-form representation in various related data structures. As an additional goal, we also integrated a simple deadlock detection scheme directly into the existing framework.

We made the region analysis' findings accessible via the *parallelism visualizer*, a visual tool integrated into the graphical user interface of EVE, that highlights entity names in the program text with colors associated to processor regions, and gives programmers options to navigate along routine calls. There is an option to show an abstract heap topology as a graph, and the ability to refine these topologies as desired.

7.2 Future Work

The following list contains areas of potential future work.

Support for agents Agents are delayed operations in Eiffel. They are essentially objects that store some routine to call, alongside some actual arguments (if any) to pass. An agent's creation and application are two distinct events. In SCOOP programs, special care must be taken, since agents can only be called when the processor region of the associated target object is locked (otherwise, we have a *traitor*[1]). The region analysis does not yet support agents explicitly, because it treats them the same way as any other object. In particular, whenever `call` is invoked on an agent, some external code gets executed, which is not tracked by the region analysis. However, since agents are dynamic objects, it would require some other form of analysis to determine which routine gets actually called at runtime.

Recursive calls There is still room for improvements when it comes to direct or indirect recursive calls. The current solution fits well into the analysis' overall framework, but is less accurate than possible. Improving the transfer functions for these calls may require the specification and implementation of preprocessing phases. For the sake of accuracy, this may be worth investigating.

Improved alias analysis The region analysis is conservative with respect to potential object aliases. It resets a state graph's attribute labels whenever there is a possibility that two labels may refer to the same object at runtime. Also, only one level of attributes is considered for every controlled label. This limited horizon could be extended, but the benefits decrease drastically with every additional level of attributes introduced, since the potential for aliases increases exponentially. The problem of handling side effects is an entirely separate research area. Some approaches focus on extending the program specification, like [2]. Other approaches are concerned with the use of dedicated static analyses, such as [4], which might be a candidate for the region analysis to cooperate with.

Advanced deadlock detection As described in section 5.6.2, we partly specified a more advanced approach for detecting some forms of deadlocks. These ideas can be investigated more thoroughly, and integrated into the region analysis, if they are not worth an analysis on their own.

Tool improvements The parallelism visualizer is not yet tightly integrated into EVE. There are several opportunities to make the look-and-feel more closely resemble the rest of EVE's graphical user interface.

[1]Agents that are callable on a non-locked processor region.

List of Figures

Listings

Bibliography

[1] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[2] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Proceedings of the 14th International Conference on Formal Methods*, FM'06, pages 268–283, Berlin, Heidelberg, 2006. Springer-Verlag.

[3] Setrag N. Khoshafian and George P. Copeland. Object identity. *SIGPLAN Not.*, 21(11):406–416, June 1986.

[4] Bertrand Meyer. The theory and calculus of aliasing. *CoRR*, abs/1001.1610, 2010.

[5] Benjamin Morandi. *Prototyping a Concurrency Model*. PhD thesis, ETH Zurich, 2014.

[6] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[7] Piotr Nienaltowski. Flexible locking in SCOOP. In *First International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE)*, York, United Kingdom, July 2006.

[8] Piotr Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, ETH Zurich, 2007.

[9] Piotr Nienaltowski, Bertrand Meyer, and Jonathan S. Ostroff. Contracts for concurrency. *Form. Asp. Comput.*, 21(4):305–318, July 2009.

[10] Atanas Rountev, Scott Kagan, and Thomas Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In *Proceedings of the 15th International Conference on Compiler Construction*, CC'06, pages 2–16, Berlin, Heidelberg, 2006. Springer-Verlag.

[11] Scott West. *Correctness and Execution of Concurrent Object-Oriented Programs*. PhD thesis, ETH Zurich, July 2014.

[12] Scott West, Sebastian Nanz, and Bertrand Meyer. A modular scheme for deadlock prevention in an object-oriented programming model. In *Proceedings of the 12th International Conference on Formal Engineering Methods and Software Engineering*, ICFEM'10, pages 597–612, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] Scott West, Sebastian Nanz, and Bertrand Meyer. Efficient and reasonable object-oriented concurrency. *CoRR*, abs/1405.7153, 2014.

[14] Stefan Zurfluh. Rule-based code analysis. Master's thesis, ETH Zurich, 2014.