A comprehensive operational semantics of the SCOOP programming model

Benjamin Morandi, Sebastian Nanz, and Bertrand Meyer

Chair of Software Engineering, ETH Zurich, Switzerland

Abstract. Operational semantics has established itself as a flexible but rigorous means to describe the meaning of programming languages. Oftentimes, it is felt necessary to keep a semantics small, for example to facilitate its use for model checking by avoiding state space explosion. However, omitting many details in a semantics typically makes results valid for a limited core language only, leaving a wide gap towards any real implementation. In this paper we present a full-fledged semantics of the concurrent object-oriented programming language SCOOP (Simple Concurrent Object-Oriented Programming). The semantics has been found detailed enough to guide an implementation of the SCOOP compiler and runtime system, and to detect and correct a variety of errors and ambiguities in the original informal specification and prototype implementation. In our formal specification, we use abstract data types with preconditions and axioms to describe the state, and introduce a number of special run-time operations to model the runtime system with our inference rules. This approach allows us to make our large formal specification manageable, providing a first step towards reference documents for specifying object-oriented languages based on operational semantics.

Keywords: concurrent programming, operational semantics

1. Introduction

Concurrent programming has become an important part of mainstream software development, caused by the widespread use of multicore processors. The notorious difficulty of writing concurrent programs correctly has on the other hand spawned work into novel language abstractions to express concurrency and synchronization. One such language is SCOOP [19, 22, 21], an object-oriented programming model for concurrency based on the idea of contracts.

The main idea of SCOOP is to simplify the writing of correct concurrent programs for developers, who can use familiar concepts from object-oriented programming but are protected by the model from common concurrency errors such as data races. This is achieved by a runtime system that automatically takes care of operations such as obtaining and releasing of necessary locks, without the need of explicit program

Correspondence and offprint requests to: Benjamin Morandi, e-mail: benjamin.morandi@inf.ethz.ch statements. While being based on conceptually simple ideas, the semantics of the language concepts and runtime system turns out to be very complex.

The question is therefore how the semantics can be properly documented. The initial version of SCOOP has been defined in [19], where all the main concepts are outlined but implementation aspects are neglected for the most part. A first prototype implementation was then introduced in [22], where the semantics was described only informally, with the exception of a formalization of the type system. In this paper we provide a full formalization of the operational behavior of SCOOP, specified by a structural operational semantics. The main contributions of the paper are:

- a formal specification of SCOOP that treats all important language elements;
- clarification and correction of the informal specification in [22];
- an approach to writing large formal language specifications, using abstract data types with preconditions and axioms for the state formalization, and special operations (run-time syntax) to model the behavior of the runtime within an inference system.

The paper is structured as follows. In the remainder of this introduction, we give a brief overview of the main ideas of the SCOOP model to provide a basic intuition for the main part of the paper. In Section 2 we give an overview of related work. In Section 3 we describe the syntax of SCOOP. The two following chapters contain the main parts of our model: in Section 4 we describe the state formalization and in Section 5 the formalization of computations. We conclude and discuss future applications of our model in Section 6.

An informal overview of SCOOP

The central idea of SCOOP is that every object is associated for its lifetime with a *processor*, an abstract notion denoting a site for computation: just as threads may be assigned to cores on a multicore system, processors may be assigned to cores, or even to remote processing units. References can point to local objects (on the same processor) or to objects on other processors; the latter ones are called *separate* references. Calls within a single processor remain synchronous, while calls to objects on other processors are dispatched asynchronously to those processors for execution, thus giving rise to concurrent execution.

The SCOOP version of the producer/consumer problem serves as a simple illustration of these main ideas. In a root class, the main entities *producer* and *consumer* are defined. The keyword **separate** denotes that these entities may be associated with a processor different from the current one.

producer: **separate** *PRODUCER*
consumer: **separate** *CONSUMER*

Creation of a separate object such as *producer* results in the creation of a new processor and of a new object of type *PRODUCER* that is associated with this processor. Hence in this example, calls to *producer* and *consumer* will be executed concurrently, as they will be associated with two different new processors.

Both *producer* and *consumer* access an unbounded buffer

buffer: **separate** *BUFFER* [*INTEGER*]

and thus their access attempts need be synchronized to avoid data races, by mutual exclusion, and to avoid that an empty buffer is accessed, by condition synchronization. In the following we explain how these two mechanisms are expressed in SCOOP.

To ensure mutual exclusion, the model prescribes that separate objects needed by a routine be *controlled*, i.e. passed as arguments to the routine. The runtime system will automatically lock the processors corresponding to such objects, hence preventing data races on the group of controlled objects for the duration of the routine. For example, in a call *consume* (*buffer*), the separate object *buffer* is controlled and thus the processor associated with *buffer* gets locked.

For condition synchronization, the condition to be waited upon can be explicitly stated as a precondition, indicated by the keyword **require**. The evaluation of the condition uses wait semantics: the runtime system automatically delays the routine execution until the condition is true. For example, the implementation of the routine *consume*, defined in the consumer, ensures that an item from *a_buffer* is only removed if *a_buffer* is not empty.

```
consume (a_buffer: separate BUFFER[INTEGER])
  require
    not (a_buffer.count = 0)
  local
    value: INTEGER
  do
    value := a_buffer.get
  end
```

The runtime system further ensures that the result of the call *a_buffer.get* is properly assigned to *value* using a mechanism called *wait by necessity*: while the client usually does not have to wait for an asynchronous call to finish, it will do so if it needs the result of this call.

As the buffer is unbounded, the corresponding producer routine does not need a condition to be waited upon, however mutual exclusion will be ensured as before:

```
produce (a_buffer: separate BUFFER[INTEGER])
  local
    value: INTEGER
  do
    value := new_value
    a_buffer.put (value)
  end
```

In the main part of the paper, we define formally the implementation that gives rise to the behavior outlined above. We will also introduce advanced concepts and additional language elements, which cannot be covered in a brief overview, and show how these give rise to a complexity which can only be dealt with satisfactorily by formal specification.

2. Related Work

We divide the presentation in this section into related work on SCOOP and related work on other concurrent programming languages.

2.1. SCOOP

In his dissertation, Nienaltowski [22] worked out the details of an implementation of SCOOP as suggested by Meyer [19], and provided a prototype implementation. The language semantics is described informally only, with the exception of the type system which is defined using an inference system. The informal description and the prototype contain various ambiguities and omissions, which we are able to clarify with our semantics.

Torshizi et al. [30] have defined and implemented JSCOOP, a version of the SCOOP model for the Java language. Only the most important language elements are considered, and no attempt at formalization is made. In contrast, our specification and implementation [29] on top of Eiffel considers all language elements. We believe that our specification could help to extend JSCOOP to a full treatment of the language concepts as well.

Brooke, Paige and Jacob [5] have used CSP [11] to give a semantics to SCOOP as described by Meyer [19]. Their initial hope was to use tools for analyzing CSP specifications, such as FDR, to automatically check for deadlock in SCOOP programs, but found that the available tools could not handle their specification. A benefit of their approach is that CSP provides the machinery needed to express concurrency and synchronization, leading to a relatively concise model. Our goal is to provide formal descriptions close to an actual implementation, and therefore prefer to design an own operational semantics, rather than going through the indirection of another process algebra.

Structural operational semantics, introduced by Plotkin [26], is a flavor of operational semantics that has been used with great success to define various concurrent systems. Our specification uses this style of semantics as well. To model SCOOP we also make use of established modeling concepts from process algebra, such as the notion of channels, which is present in most calculi such as CSP [11] or the π-calculus [20]. We use the theory of abstract data types (ADT) [15] to model the elements of a program text and to model the state of a SCOOP program.

Ostroff et al. [25] describe a structural operational semantics for SCOOP in the refined version by Nienaltowski [22]. This operational semantics inspired our work, and we have attempted to stay close to their modeling ideas where possible, so that [25] can be viewed as a reduced version of the semantics we describe in this paper. While [25] covers some of the most significant aspects of SCOOP, it falls short of describing a number of other critical language concepts: in their reduced model, a query routine handled by some processor p must not make calls to a processor other than p; lock passing, expanded objects and the import mechanism, once routines, evaluation of (asynchronous) postconditions and invariants, and explicit processor tags are not considered. We clarify these aspects in this paper. Furthermore, [25] have pursued the goal to check temporal logic properties of SCOOP programs using their semantics and the SPIN model checker, but were limited to small programs by state space explosion. We have the different goal of providing a reference document for SCOOP, and thus don't have to sacrifice coverage of the language for keeping the size of the specification small.

2.2. Other concurrent programming languages

Axum [3] is a concurrent programming language. The language is based on the actor model. In Axum, actors are called agents. An agent is an isolated runtime component that executes in parallel with other agents. The agents communicate with each other by sending messages through channels. Each channel has input ports, output ports, and a protocol. The ports are queues of messages. The protocol is a state machine that defines how the channel behaves. Schemas define the structure of messages. Besides message passing, Axum also provides domains - shared state between groups of actors. SCOOP does not use the actor model. Instead, SCOOP defines processors that execute in parallel. Each processor can only operate on its assigned set of objects. Processor do not exchange messages as such, but they do exchange feature requests where a client processor asks a supplier processor to execute a feature. SCOOP does not define shared space. However, it would be interesting to explore this concept for SCOOP. SCOOP takes a different approach to concurrency than Axum. Axum is a domain-specific language for concurrency. SCOOP considers a concurrent program as a generalization of a sequential program. For this reason, all object-oriented concepts such as inheritance have been generalized for the concurrent case. As a consequence of this effort, the synchronization mechanisms have been defined in such a way that inheritance anomalies [17] do not occur.

Cω [4] is an extension of C#. It integrates elements of the Join Calculus [10]. Cω allows computations to be spawned off into different threads using asynchronous methods: while for synchronous methods the caller must wait until a routine completes, asynchronous methods return immediately while their body is scheduled for execution in another thread. Cω supports so-called chords, which associate the body of a routine with more than one method; only if all methods have been called will the body be executed.

X10 [7], Fortress [2], and Chapel [12] are based on the Partitioned Global Address Space (PGAS) model. PGAS uses a global shared memory. It defines portions on the global shared memory and associates them to specific processors to improve performance and scalability. X10 provides important abstractions such as places, asynchronous methods, future invocations, and barriers. However, it places a considerable burden on programmers. Fortress offers implicit parallelization of loops and operations on data structures. Chapel provides a higher-level multithreaded parallel programming model with abstractions for data parallelism, task parallelism, and nested parallelism.

For the related languages mentioned above, we are not aware of rigorous behavioral specifications, with the exception of Cω, which uses the Join Calculus as the underlying model. For multi-threaded Java however, such formalizations have been attempted.

Ábrahám, de Boer, de Roever, and Steffen [1] present an operational semantics for a subset of multi-threaded Java. They focus on the most important multi-threaded aspects, i.e. dynamic thread creation, thread termination, and re-entrant monitors. The semantics consists of two components: the semantics for isolated objects and the semantics for interacting objects. The authors want to use the semantics to develop a proof system that is based on an existing proof-system for isolated objects. A configuration is a set of instance configurations. An instance configuration contains the attribute values of one object. It also contains the local environment and the expression of each thread that is concurrently executing within the object. In modeling the state of a program, our semantics strictly separates the actions to be executed from the data. This makes it easier to derive implementations from the semantics because an implementation is likely to keep the program text and data separate. Ábrahám et al. use transition labels to synchronize inference rules. The labels allow an external observer to follow the transitions. Our semantics is a pure reduction semantics without labels because we do not require observable transitions.

Cenciarelli, Knapp, Reus, and Wirsing [6] also describe an operational semantics for a larger subset of multi-threaded Java. They cover a larger number of multi-threaded aspect than [1]. In particular they formalize Java's notification mechanism and the working memory. A configuration consists of a function that maps each thread to its expression and its local environments. The configuration also has a container with the objects and the static typing information. Lastly, the configuration consist of an event space. The event space is a partially ordered set of events that have been executed by the threads. The ordering reflects the order in which the events took place. An event space serves two purposes. First, it contain certain aspects of the state. For example, the lock and unlock actions tell us which thread owns which lock. Second, it records the history. A number of constraints state when an event space is valid. Hence, the event space indicates which further actions can take place. The authors use two different validity constraints for both Java's non-prescient semantics and its prescient semantics. Using this, they show that any prescient execution of a properly synchronized program can be simulated by a non-prescient execution. Compared to our semantics, there is no clean division between program text and the state and there is no clean division between the state and the typing information.

Lochbihler [16] suggest a different operational semantics for a large subset of multi-threaded Java. Just like [6], he covers the notification mechanism, but he does not formalize the working memory. He defines an instantiating semantics based on an extension of Jinja [14]. Jinja is an operational semantics for a subset of single-threaded Java. The instantiating semantics is used for the sequential case. Lochbihler defines a generic formal framework to lift the instantiating semantics to the concurrent case. The configuration of the instantiating semantics consists of the expression, a container with the objects, and the local environments. The state of the framework semantics consists of the lock status, the thread information with the thread's expression along with the thread's local environments, a container with the objects, and the wait sets. Lochbihler formalizes the notion of deadlocks, where deadlocks are either based on locks or on wait sets. He then proves that every program that satisfies certain criteria either produces a final value, throws an exception, or deadlocks. He also shows that every such program preserves type safety. Every notion and every theorem is proved with the theorem prover Isabelle/HOL [23]. In future work we want to prove properties of the SCOOP model.

3. Language overview

SCOOP is a programming language based on Eiffel, an object-oriented programming language, defined in the Eiffel ECMA standard [9]. The programming language SCOOP defines the SCOOP model, which can be applied to other programming languages as well. For this reason, we do not focus on Eiffel, but on the SCOOP model. We define a subset of the SCOOP programming language that only covers the parts that are relevant for the SCOOP model. This excludes aspects of Eiffel that are not relevant for the SCOOP model. In this section we describe this subset. In the following we first define the syntax of the considered language. We then discuss the simplifications that have been taken with respect to Eiffel. We then discuss the intermediate representation of a program, which is based on the notion of abstract data types (ADT). For this purpose we introduce a number of basic ADTs.

3.1. Syntax

The following EBNF grammar defines the set of all considered programs:

program = *class_declaration*∗ *root_procedure_declaration* ;

```
root_procedure_declaration = {class_name}.routine_name ;
class_declaration =
  [”expanded”] ”class” class_name
    ”inherit” class_name
    [”create” routine_name {”,” routine_name}]
    ”feature” [”{” class_name {”,” class_name} ”}”] {feature_declaration}
    [”invariant” expression]
  ”end” ;

feature_declaration = routine_declaration | attribute_declaration ;
routine_declaration =
  routine_name [”(” entity_declaration {”,” entity_declaration} ”)”] [”:” type]
    [”require” expression]
    [”local” entity_declaration {entity_declaration}]
    (”do” | ”once”)
      instruction {”;” instruction}
    [”ensure” expression]
    ”end” ;
attribute_declaration = entity_declaration ;
entity_declaration = entity_name ”:” type ;

instruction =
  entity_name ”:=” expression |
  expression ”.” feature_name [”(” expression {”,” expression} ”)”] |
  ”create” entity_name ”.” routine_name [”(” expression {, expression} ”)”] |
  ”if” expression ”then” instruction {”;” instruction} ”else” instruction {”;” instruction} ”end” |
  ”until” expression ”loop” instruction {”;” instruction} ”end” ;

expression =
  literal |
  entity_name |
  expression ”.” feature_name [”(” expression {, expression} ”)”] ;
literal = boolean_literal | integer_literal | character_literal | void_literal ;
boolean_literal = ”True” | ”False” ;
integer_literal = [”–”](”0” | ... | ”9”) {”0” | ... | ”9”} ;
character_literal = ” ’ ” ”a” | ... | ”z” | ”A” | ... | ”Z” | ”0” | ... | ”9” ” ’ ” ;
void_literal = ”Void” ;

type =
  [detachable_tag]
  [”separate”] [explicit_processor_specification]
  class_name [actual_generics] ;
detachable_tag =
  ”attached” | ”detachable” ;
explicit_processor_specification =
  qualified_explicit_processor_specification |
  unqualified_explicit_processor_specification ;
qualified_explicit_processor_specification =
  ”<” entity_name ”.” ”handler” ”>” ;
unqualified_explicit_processor_specification =
  ”<” entity_name ”>” ;

class_name = name ;
feature_name = routine_name | entity_name ;
routine_name = name ;
entity_name = name | ”Result” | ”Current” ;
```

$name = (\textbf{"a"} \mid \ldots \mid \textbf{"z"} \mid \textbf{"A"} \mid \ldots \mid \textbf{"Z"})\ \{\textbf{"a"} \mid \ldots \mid \textbf{"z"} \mid \textbf{"A"} \mid \ldots \mid \textbf{"Z"}\};$

The absence of both the **attached** and **detachable** keyword is treated as if there was an **attached** keyword. Note that the grammar anticipates a change in the syntax of the detachable tag which is not yet part of the Eiffel ECMA standard [9]. In the following we assume that each program follows the above grammar and that each program satisfies the SCOOP type rules. A formalization of these rules can be found in Nienaltowski's dissertation [22].

3.2. Simplifications

In the following we point out the most important simplifications with respect to Eiffel.

- We do not consider unqualified feature calls. We expect all feature calls to be in the qualified form. This includes accesses to attributes of the current object in expressions.
- We do not consider infix feature calls. We expect all feature calls in the non-infix form. For example, an expression $x > y$ must be transformed into the equivalent form $x.is\_greater(y)$.
- We simplify the automatic initialization of entities. All entities, except for the current object entity, are initialized with the void reference.
- We neglect exception handling. The exception handling mechanism for SCOOP is still under development.
- We do not consider garbage collection because garbage collection is not refined in the SCOOP model.

3.3. Intermediate representation

For the purpose of the formalization we assume that a program is given to us in an enriched intermediate representation, where the syntactical elements are replaced with instances of abstract data types. In particular we assume ADTs for class types, features, expressions, and instructions. We briefly summarize these ADTs in figure 1.

The instances of the ADT **CLASS_TYPE** are all possible class types, i.e. the types directly defined by all non-generic classes and all possible generic derivations of all possible generic classes. We omit the creation of such instances because it is not relevant for our formalization. We discuss in Section 4.1.3 how we can get these instances. The class type ADT defines a name query *name*. Each class type can either be a reference class type or an expanded class type. The queries *is_ref* and *is_exp* provide this information. Each class type defines a number of features. These features can be divided into attributes, functions, and procedures. An attribute of an object stores a value. A function performs a computation and returns the result. This computation must not modify the state. A procedure performs a computation that modifies the state. Functions and procedures are also known as routines. For each of these categories, the class type ADT defines a query that returns a tuple of features. The query *attributes* returns a tuple of attributes, the query *functions* returns a tuple of functions, and the query *procedures* returns a tuple of procedures. If the name of a feature is known, then the query *feature_by_name* can be used to get the feature with that name. Every class type can have an invariant. The query *inv_exists* indicates whether such an invariant exists. In case an invariant exists, it can accessed with the query *inv* as an expression. As one of the instances of the **CLASS_TYPE** ADT we assume an instance *BOOLEAN*. This class type is expanded and it has an attribute with name *item*. The value of this attribute is the represented boolean value, i.e. an instance of the **BOOLEAN** ADT.

In our formalization, a feature is an instance of the ADT **FEATURE**. We assume that these instances are given to us in the intermediate representation. The name of the feature can be retrieved with the query *name* and the formal arguments are given by the query *formals* that returns a tuple with the formal arguments as entities. Whether or not the feature is a once feature can be determined using the query *is_once*. The queries *pre* and *post* return an expression for the precondition respectively the postcondition, provided that the queries *pre_exists* and *post_exists* indicate that the assertions exist. Next, we have the query *locals* that gives us the locals of the feature as entities. The query *body* returns the body of the feature as a tuple of instructions. Each feature is either exported or not. A non-exported feature is only available in calls on the current object within the class that declared the feature. An exported feature can be called by other clients

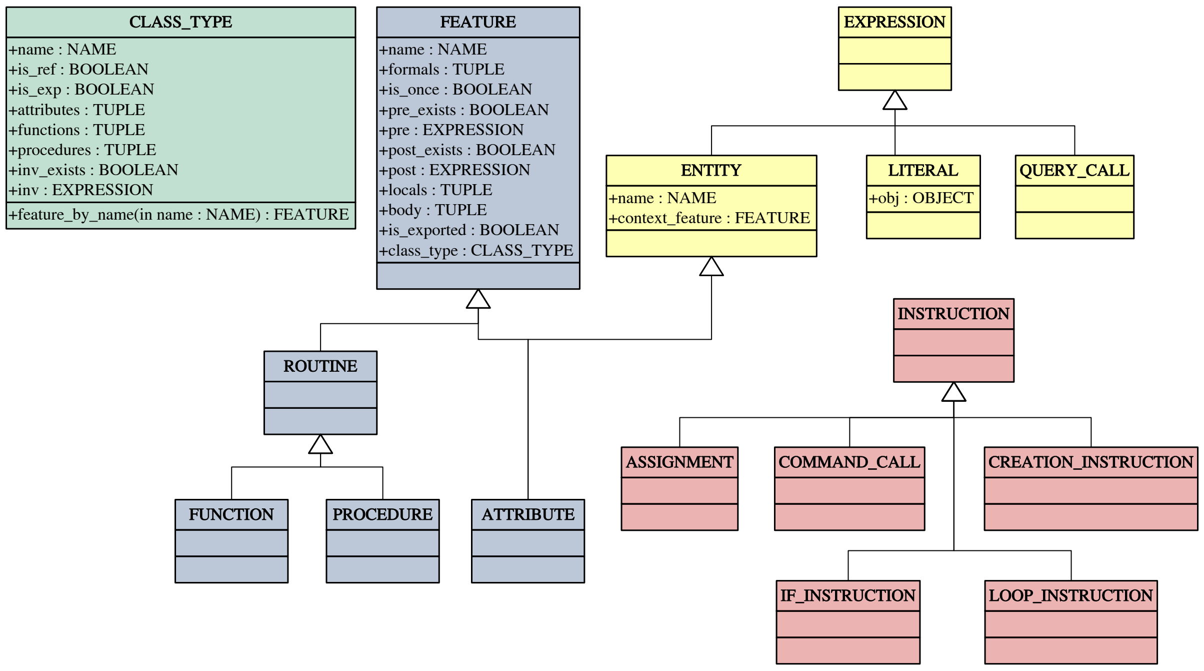

Figure 1. ADTS for the intermediate representation

as well. The query *is_exported* returns whether a feature is exported or not. Lastly, ever feature has a link to the class it belongs to. This is given by the query *class_type*. This can be used for example to retrieve the invariant that must be preserved by a feature. For each feature category we define an ADT that inherits from the **FEATURE** ADT.

Expressions can either be entities, literals, or query calls. We introduce an ADT **EXPRESSION** whose instances are all expressions. For each form of expression, we introduce an ADT that inherits from the **EXPRESSION** ADT. For entities we introduce an ADT with a query *name* that returns the name of an entity. A query *context_feature* links an entity to the feature in which the entity is declared. A literal is a character sequence that represents a constant value. As such, literals count as manifest expressions - programming constructs whose values can be deduced by the compiler statically. We model literals as instances of an ADT **LITERAL**. This ADT has instances for booleans, integers, characters, and the void literal. Each literal except the void literal can be translated into an object with the query *obj*. This object matches the literal in both type and value. We look at the object ADT later on in this article. The following notation is used to describe instances of the **EXPRESSION** ADT:

$$e \triangleq w \mid b \mid e.f(e, \dots, e)$$

Here, w is an element of the **LITERAL** ADT, b is an instance of the **ENTITY** ADT, and f is an instance of the **FEATURE** ADT. Note once again, that we reserve entity expressions for internal purposes.

For instructions we also introduce an ADT **INSTRUCTION** with ADTs for each kind of instruction. The following notation is used to describe such instances:

$h \triangleq$
$b := e \mid$
$e.f(e, \ldots, e) \mid$
create $b.f(e, \ldots, e) \mid$
if e **then** $[s\{;s\}*]$ **else** $[s\{;s\}*]$ **end** $\mid$
until e **loop** $[s\{;s\}*]$ **end**

Here, s stands either for an instance of the **INSTRUCTION** ADT or an operation. Instructions are actions that are explicit in the intermediate representation (user syntax). Operations are actions that do not explicitly occur in the intermediate representation (run-time syntax).

Up until now we omitted the types. While it is possible to model this information as queries of the ADTs, we build on an existing type formalization. As we will explain later, we assume the existence of a typing environment that can be queried for type information.

3.4. Basic ADTs

We assume an ADT **BOOLEAN** for the two boolean values *true* and *false* along with the typical boolean operators. We assume an ADT **NAME** for names and an ADT **ID** for identifiers.

Based on these ADTs, we have a generic ADT **SET**[**G**] whose instances are sets of elements with type **G**. A set can be created with a call to the constructor *make* and it can be populated with a call to the command *add* that takes an element and returns a new set containing the element. However, we prefer to define an instance of this ADT with set inference or with a set expression $\{x_1, \ldots, x_n\}$. A set can be inspected with the query *has* that returns whether an element is part of the set or not. The $\in$ notation is used as an equivalent alias to the *has* query. A query *is_empty* tells us whether the set is empty or not, and the query *count* returns the number of elements in the set.

A tuple is an ordered list of elements. We assume a generic ADT **TUPLE** that takes zero or more generic parameters for the types of each tuple element. Each generic parameter **G** restricts the set of possible tuples to the ones who have an element of type **G** in the respective position. For example, the instances of the ADT **TUPLE** without any generic parameters are all possible tuples. The instances of the ADT **TUPLE**[**A**] are all tuples whose first elements are of type A. Note that this also includes tuples with more than one element, as long as the first element is of type A. A tuple can be constructed with a call to the constructor *make*. Elements can be added to the end of the tuple with a call to the command *add*. We prefer to construct tuples with tuple expression $(x_1, \ldots, x_n)$. The query *has* checks whether an element is part of the tuple or not and the query *is_empty* checks whether the tuple is empty or not. The query *count* returns the number of elements in the tuple.

Furthermore, we assume a generic ADT **STACK**[**G**] for stacks with elements of type **G**. A stack gets constructed with a call to *make*. An element gets pushed with a call to *push* and is then available through the query *top*. The element can then be popped with a call to *pop*. A query *is_empty* returns whether the stack is empty or not. We add a query *flat* that returns a set of type **SET**[**G**]. This set contains all the elements in the stack.

Next, we assume an ADT **MAP**[**K**, **G**] for maps with keys of type **K** and values of type **G**. As the name suggests, a map associates keys to values. A map gets constructed with a call to the constructor *make*. An association can be added with a call to *add* where the first argument is the key and the second argument is the value. The set of all possible keys can be retrieved with a call to the query *keys*. The value for a specific key in this key set is returned by the query *val*.

4. State formalization

In this section we provide a formalization of the state of a SCOOP program. This is necessary to describe the effect of SCOOP constructs on the state, as it is done later on. We start with some comments on the general approach. Then we dive into the description of the state.

4.1. General approach

We consider the state of a SCOOP program to be a data structure that can be created, modified, and queried through features. We are mainly interested in the interface of the state data structure and we defer the implementation of the data structure to other projects. For the specification of the state we use Liskov's ADT theory [15]. We begin with a justification of this choice. We finish the general comments with our solution on identity management and an explanation of how to get types for elements in the intermediate representation.

4.1.1. Abstract data types

Meyer's work on a three-level approach to the description of data structure [18] defines three levels on which a data structure can be described: functional, constructive, and physical. The functional specification is an algebraic approach that uses an implicit characterization of the data structure. The constructive specification provides a means to construct instances of the data structure. The instances constructed like this are mathematical entities. A physical description describes the layout of instances in memory. The constructive specification can be derived from the functional specification and the physical description can be derived from the constructive description.

We describe the type of the state as an ADT on the functional level in the hierarchy described above. This has several reasons. First of all, ADT theory allows us to describe the state on an abstract level without dealing with aspects of the implementation. The constructive and the physical level can be derived from our ADTs on the functional level. Second, ADT theory allows us to modularize the state. Different concerns of the state can be modeled as individual ADTs, while a single ADT can be used to consolidate the individual ADTs. This improves understandability and maintainability of the state description. Lastly, ADT theory is well established and suitable for what we have to do.

An ADT t consists of queries, commands, and constructors. A query of t provides information about an instance of t. The query takes as a first argument the target of type t, which is the instance to be queried. Next to the target, the query can take further arguments with types $t_1, \dots, t_n$. Finally, the query returns a result of a type t_{n+1}. We write the declaration of this query as $\mathit{query}\colon t \to t_1 \to \dots \to t_n \to t_{n+1}$. Note that for flexibility reasons we use the curried form as used in Haskell instead of the equivalent Cartesian form $\mathit{query}\colon t \times t_1 \times \dots \times t_n \to t_{n+1}$. A command is used to modify an instance of t. However, instances of t are mathematical objects and as such they do not get modified. A modified instance is a different instance. This is reflected in the way a command gets declared. The declaration looks much like the one of a query. However, the result of the command is an instance of t. The result is the updated target according to the semantics of the command. A constructor of t creates a new instance of t. In contrast to query and commands, a constructor does not take the target as the first argument because its purpose is to create a new instance. However, a constructor can take a number of arguments.

To describe an instance of an ADT we can build an expression that starts with a constructor call. This expression can then be used as the first actual argument of a command call. The resulting expression can then be used as the first actual argument of the next command. This leads to a nested expression, in which the first feature call is in the root of the expression and the last feature call is on the outside of the expression. The instance described in such a way can then be queried. We find this functional notation hard to read. Therefore we use an equivalent object-oriented notation in which the first feature call is on the left and the last feature call is on the right. The main idea is to not to write targets as arguments, but to write a target in front of the feature name and to use a dot to separate the two parts from each other. This leads to the following translation between the functional notation and the object-oriented notation:

- The query expression $\mathit{query}(e_0, e_1, \dots, e_n)$ written in functional notation is equivalent to the expression $e_0.\mathit{query}(e_1, \dots, e_n)$ written in object-oriented notation.
- The command expression $\mathit{command}(e_0, e_1, \dots, e_n)$ written in functional notation is equivalent to the expression $e_0.\mathit{command}(e_1, \dots, e_n)$ written in object-oriented notation.
- The creation expression $\mathit{constructor}(e_1, \dots, e_n)$ for an instance of an ADT t written in functional notation is equivalent to the expression $\mathit{new}\, t.\mathit{constructor}(e_1, \dots, e_n)$ written in object-oriented notation.

Example 1 (Functional notation versus object-oriented notation) The expression in functional notation $\mathit{is\_empty}(\mathit{pop}(\mathit{push}(\mathit{new}\,\mathbf{STACK}[\mathbf{PROC}].\mathit{make}, p)))$ can be written in object-oriented notation as $\mathit{new}\,\mathbf{STACK}[\mathbf{PROC}].\mathit{make}.\mathit{push}(p).\mathit{pop}.\mathit{is\_empty}$. $\square$

Each feature can have a precondition that must be satisfied before the feature gets called. A precondition is expressed as a number of assertions on the target and the arguments. A feature with a precondition is a partial feature. A partial feature is a feature whose domain is restricted. We indicate such a partial feature with a crossed arrow $\nrightarrow$ after the type of each formal argument that got restricted by the feature's precondition. For non-restricted formal arguments we use the normal arrow $\rightarrow$. The effect of an ADT command is described in a number of axioms. We deviate from the practice of bundling all axioms for a specific ADT. Instead, we write all the axioms for a specific feature in a single feature declaration. Note that we do not aspire a sufficiently complete ADT because it would lead to rule explosion. An ADT is sufficiently complete if its axioms make it possible to reduce any query expression to a form that does not involve an instance of the ADT. This requires that the axioms describe the effect of each command on each query. We follow the practice to describe the effect of each command of an ADT on all the queries of the ADT that have been changed by the command. Unmentioned queries are unchanged.

Example 2 (Command declaration) The following declaration shows a command to set the value of an attribute f of an object o to the value v. The value can either be a reference or a processor. The command takes the object as the target and returns an updated object whose attribute value is set.

set_att_val: $\mathbf{OBJ} \rightarrow \mathbf{FEATURE} \nrightarrow \mathbf{REF} \cup \mathbf{PROC} \rightarrow \mathbf{OBJ}$

$o.set\_att\_val(f, v)$ **require**

$o.class\_type.attributes.has(f)$

axioms

$o.set\_att\_val(f, v).att\_val(f) = v$

The command states in its precondition that the class type of the target object o must have an attribute f. This is expressed as an assertion after the require keyword. The part in front of the require keyword is used to give names to the target and to the arguments. Note that the precondition makes the command partial. The updated object has the value of its attribute f set to v. This is stated as an axiom after the ensure keyword. □

So far we looked at queries, commands, and constructors for ADTs. We extend the ADT theory with the notion of auxiliary features. Auxiliary features are convenience features that are not essential for the definition of the ADT, but nevertheless useful. In the remaining of this article we will declare various ADTs to model the state of a SCOOP program. Unless it would create confusion, we will use the same name for an instance of an ADT and for the corresponding domain element. For example, we will call the instance of the **OBJ** ADT an object.

4.1.2. Identifier management

Our formalization models objects, references, and processors. All of these domain elements have an identity. These identities are automatically managed by the runtime system. The work by Khoshafian and Copeland [13] on different levels of object identity provides good reasons for this decision. They introduce a scale that starts with identities given by the value, goes on with user-supplied identities, and ends with built-in identities. Built-in identities have the advantage that the identities are preserved in case of user modifications. For example, if an object gets updated with a new attribute value then the identity of the object is preserved. According to this hierarchy, our domain elements have a built-in identity. We must model this in our formalization. One straightforward way is to model each domain element as an instance of an ADT. However, with this direct approach we do not properly capture the identities of our domain elements because the identity of an ADT instance is not built-in but based on its value. A user modification of an ADT instance leads to an ADT instance with a new identity. For example, if an ADT instance that models an object gets updated with a new attribute value then we get an ADT instance with a new identity. In this section we describe a way to introduce built-in identities for ADT instances.

In short, explicit identifiers are used to model domain elements with an identity. To model a single domain element, we use a number of ADT instances of the same ADT. This ADT defines an identity query. Each of these ADT instances has the same value for the identity query at all times. A modification of the domain element can then be modeled as a new ADT instance based on an existing ADT instance where the value of the identity query is preserved and all other queries modulo the modification are preserved.

For this to work, we must somehow ensure that no two ADT instances that model different domain elements have the same identity. We ensure this with a fresh identifier for each ADT instance that is used to model a new domain element. We then preserve the built-in identity of the ADT instance in every modification. For this, we make use of a universal stateful query *new_id* that returns a fresh identifier.

4.1.3. Typing Environment

Nienaltowski [22] presented a formalization of the SCOOP type system for a core of SCOOP called SCOOP$_\mathrm{C}$. We base our work on this formalization because it only requires minor extensions to be reused for our purposes. We want to get the types of features and expressions. We are also interested in whether an expression is controlled or not. All of this information can be retrieved from Nienaltowski's typing environment Γ and his type rules. The *typing environment* contains the class hierarchy of a SCOOP program along with all the type definitions of all features and entities. The type rules are based on the typing environment. They allow us to derive conclusion based on the typing environment.

We use the notation $\Gamma \vdash e : t$ to denote that we can derive from Γ that expression e is of type t. Based on this derivation we introduce the function $\mathit{type\_of}(\Gamma, e)$ to denote the type of expression e in the typing environment Γ. A type t is a triple (d, p, c). The component d is the *detachable tag*, p is the *processor tag*, and c is the *class type*. The detachable tag d captures detachability. An entity of attached type, written as $d = !$, is statically guaranteed to be non-void. An entity of detachable type, written as $d = ?$, can be void. As we discuss later, the detachable tag is also used for the selective locking mechanism to prevent a request queue from being locked. The processor tag p captures the locality of objects accessed by an entity of type t. The processor tag p can be separate, written as $p = \top$. The object attached to the entity of type t is potentially handled by a different processor than the current processor. The processor tag p can be explicit, written as $p = \alpha$. The object attached to the entity of type t is handled by the processor specified by α. The processor tag p can be non-separate, written as $p = \bullet$. The object attached to the entity of type t is handled by the current processor. The processor tag p can denote no processor, written as $p = \perp$. It is used to type the void reference. A processor can be specified more explicitly. This can be done either in an unqualified or a qualified way. An *unqualified explicit processor specification* is based on a processor attribute p. We write it as $< p >$. The processor attribute p must have the type $(!, \bullet, \mathit{PROCESSOR})$ and it must be declared in the same class as the explicit processor specification or in one of the ancestors. The processor denoted by this explicit processor specification is the processor stored in p. A *qualified explicit processor specification* relies on an entity e occurring in the same class as the explicit processor specification or in one of the ancestors. We write it as $< e.\mathit{handler} >$. The entity e must be a non-writable entity of attached type and the type of e must not have a qualified explicit processor tag. The processor denoted by this explicit processor specification is the same processor as the one of the object referenced by e. Explicit processor tags support precise reasoning about object locality. Entities declared with the same processor tag represent objects handled by the same processor. Last but not least, the class type c is an instance of **CLASS_TYPE**.

The type rules can be used to check whether an expression is controlled or not. In a SCOOP program, each processor p that wants to apply a feature f must make sure that all the processors $(q_1, \ldots, q_n)$ of all attached actual arguments of f are exclusively available on behalf of processor p. This guarantees exclusive access on all objects handled by processors $\{p, q_1, \ldots, q_n\}$. Note that processor p is in this set too because p can exclusively access its objects during a feature execution. For safety, the type system only allows feature calls in f on expressions, where the type system can derive that the value of the expression is a reference to an object and this object is handled by one of the processors $\{p, q_1, \ldots, q_n\}$. Such an expression is called *controlled*. Whether or not an expression is controlled can be determined through the context in which the expression appears and the type of the expression. The context can either be the enclosing class, in case of expressions in invariants, or it can be the enclosing feature, in case of all other expressions. To be more precise, an expression e of type $t = (d, p, c)$ is controlled if and only if t is attached, i.e. $d = !$, and t satisfies at least one of the following conditions:

- The expression e is non-separate, i.e. $p = \bullet$.
- The expression e appears in a routine f that has an attached formal argument w with the same handler as e, i.e. $p = w.\mathit{handler}$.

The second condition is satisfied if and only if at least one of the following conditions is true:

- The expression e appears as an attached formal argument of f.

- The expression e has a qualified explicit processor specification $w.handler$ and w is an attached formal argument of f.
- The expression e has an unqualified explicit processor specification p, and some attached formal argument of f has p as its unqualified explicit processor specification.

We write $\Gamma \vdash \mathit{is\_controlled}(t)$ to say that we can derive from Γ that an expression e of type t is controlled.

To establish the derivation $\Gamma \vdash \mathit{is\_controlled}(t)$ one has to find an attached formal argument w in the enclosing routine such that the types suggest that w and e are handled by the same processor or one has the establish that the type t is non-separate. We can therefore be sure that whenever an expression e is controlled, either a matching formal argument exists or its type is non-separate. For the first case, we call this formal argument the *controlling entity* for e. For the second case, we call the current entity the controlling entity. Although not present in Nienaltowski's formalization of the type system, we introduce a new derivation $\Gamma \vdash y = \mathit{controlling\_entity}(e)$ that returns the controlling entity y for an expression e as an instance of **ENTITY**. This notion is essential to determine the handler of any controlled expression without evaluating the expression. We can simply determine the controlling entity and then determine the handler of the controlling entity. This is important to formalize the conditions for the asynchronous postcondition evaluation.

4.2. Components of the state

We divide the state into three parts: the regions, the heap, and the store. The main purpose of the heap is to keep track of objects and to maintain the mapping of references to objects. It also maintains the once status of once routines, i.e. whether a once routine is fresh on a processor. The regions manage the association between objects and processors. Objects that are handled by the same processor form a region. The regions are also concerned with locking. The store is a map of names to references. This is used to map names of formal argument, names of local variables, the name of the current object entity, and the name of the result entity to references.

We model the state with a state ADT that has one query for each of the three parts:

$$\mathit{regions}\colon \mathbf{STATE} \to \mathbf{REGIONS}$$

$$\mathit{heap}\colon \mathbf{STATE} \to \mathbf{HEAP}$$

$$\mathit{store}\colon \mathbf{STATE} \to \mathbf{STORE}$$

Before we give further details of the state ADT we will introduce one ADT for each of the parts. We then declare auxiliary features in the state ADT to construct a facade. The facade provides an abstraction on top of the three parts. This abstraction makes the execution formalization easier.

4.3. Heap ADT

The *heap* keeps track of the objects and the references associated to them. It also keeps track of the status of once routines. We first define an ADT for objects and references and then we introduce a heap ADT.

4.3.1. Objects and references

There are two kinds of class types in the SCOOP type system: reference class types and expanded class types. The main difference lies in the semantics of using an instance of the types as the source of an attachment, such as assignment or argument passing. If an object of *reference class type* is the source of an attachment then the reference to the object gets copied over to the destination of the attachment. The object is then accessible both through the source of the attachment as well as through the destination of the attachment. If an object of *expanded class type* is the source of an attachment then a copy of the object gets attached to the destination of the attachment. The details can be found in Section 7.4 of the Eiffel ECMA standard [9].

In our formalization we take a unified view on objects and references that is compatible with the semantics described in the Eiffel ECMA standard. We do not consider objects of expanded class type as sub-objects in other objects or in an environment. Instead we locate expanded objects on the heap, just like objects of reference class type. For each object we have exactly one reference. Assigning references to objects of expanded type has one major advantage for the formalization. If an ADT instance x that models an object y gets updated then we get a new ADT instance z. If we would model expanded objects as sub-objects stored in other objects or in environments then such an update might trigger a cascade of ADT instance updates. Each ADT instance that has x as a query value has to be updated with y, and so on. A consequent use of references avoids this issue. An update can be handled by redirecting the reference to x to y.

In the following we show an ADT for references. We define an ADT for references with an identity query and a constructor. The empty reference *void* is an instance of this ADT.

$id\colon \mathbf{REF} \to \mathbf{ID}$

$make\colon \mathbf{REF}$

axioms

$make.id = new\_id$

Next, we show an ADT for objects. We start with a number of queries. Each object has an identifier, a class type, and attribute values for each attribute. An object can only have attribute values for attributes that are defined in its class type.

$id\colon \mathbf{OBJ} \to \mathbf{ID}$

$class\_type\colon \mathbf{OBJ} \to \mathbf{CLASS\_TYPE}$

$att\_val\colon \mathbf{OBJ} \to \mathbf{FEATURE} \nrightarrow \mathbf{REF} \cup \mathbf{PROC}$

$o.att\_val(f)$ **require**

$o.class\_type.attributes.has(f)$

An object can only be modified with the command *set_att_val*. Only the attribute values for attributes that are defined in the class type can be modified. The result is an updated object where the attribute value of f is set to v. Note that the value can either be a reference or a processor. Processor values are necessary to support processor attributes. We introduce the **PROC** ADT later on.

$set\_att\_val\colon \mathbf{OBJ} \to \mathbf{FEATURE} \nrightarrow \mathbf{REF} \cup \mathbf{PROC} \to \mathbf{OBJ}$

$o.set\_att\_val(f, v)$ **require**

$o.class\_type.attributes.has(f)$

axioms

$o.set\_att\_val(f, v).att\_val(f) = v$

The constructor *make* can be used to create a new object. It creates a new object with the given class type. The new object has a new identifier that is given by the query *new_id*. The constructor initializes all the attribute values of the new object with the void reference *void*.

$\mathit{make}\colon \mathbf{CLASS\_TYPE} \to \mathbf{OBJ}$

axioms

$\mathit{make}(c).\mathit{id} = \mathit{new\_id}$

$\mathit{make}(c).\mathit{class\_type} = c$

$\forall i \in \{1, \ldots, n\}\colon \mathit{make}(c).\mathit{att\_val}(a_i) = \mathit{void}$

where

$\{a_1, \ldots, a_n\} \overset{def}{=} c.\mathit{attributes}$

An object can also be copied with the auxiliary query *copy*. This is relevant for expanded objects with a copy semantics. The copied object has the same class type and the same attribute values as the original object, but it has a new identity. The new identity comes from the call to the constructor *make*.

$\mathit{copy}\colon \mathbf{OBJ} \to \mathbf{OBJ}$

axioms

$o.\mathit{copy} = \mathit{make}(o.\mathit{class\_type})$

$.\mathit{set\_att\_val}(a_1, o.\mathit{att\_val}(a_1))$

$. \ldots$

$.\mathit{set\_att\_val}(a_n, o.\mathit{att\_val}(a_n))$

where

$n \overset{def}{=} o.\mathit{class\_type}.\mathit{attributes}.\mathit{count}$

$\{a_1, \ldots, a_n\} \overset{def}{=} o.\mathit{class\_type}.\mathit{attributes}$

4.3.2. Mapping from references to objects

We introduce a heap ADT that makes use of the ADTs for objects and references to model the mapping from references to objects. For this purpose we declare the query *objs* to store all the objects on the heap and we declare the query *refs* to get all the references to these objects. The reference *void* is not part of the reference set. We also declare the query *last_added_obj* to keep track of the last object that has been added to the heap. We use this query to define the effect of adding an object to the heap. The query *ref_obj* defines the actual mapping. For each reference in *refs* an object in *objs* gets returned.

$\mathit{objs}\colon \mathbf{HEAP} \to \mathbf{SET}[\mathbf{OBJ}]$

$\mathit{refs}\colon \mathbf{HEAP} \to \mathbf{SET}[\mathbf{REF}]$

$\mathit{ref\_obj}\colon \mathbf{HEAP} \to \mathbf{REF} \nrightarrow \mathbf{OBJ}$

$h.\mathit{ref\_obj}(r)$ **require**

$h.\mathit{refs}.\mathit{has}(r)$

$\mathit{last\_added\_obj}\colon \mathbf{HEAP} \to \mathbf{OBJ}$

$h.\mathit{last\_added\_obj}$ **require**

$\neg h.\mathit{objs}.\mathit{is\_empty}$

We define a number of commands to add objects and to alter the mapping of references to objects. The command *add_obj* takes an object o and adds it to the heap. The result of the command is a new heap with the object o and a new reference that points to o. The newly added object is indicated in the query *last_added_obj*. Note that this command does not create a new object. It simply adds an object that has been provided as an argument. The command requires that the object is not yet part of the heap.

$add\_obj\colon \mathbf{HEAP} \rightarrow \mathbf{OBJ} \nrightarrow \mathbf{HEAP}$

$\quad h.add\_obj(o)$ **require**

$\quad\quad \forall u \in h.objs\colon u.id \neq o.id$

$\quad\quad \forall a \in o.class\_type.attributes\colon o.att\_val(a) \in \mathbf{REF} \rightarrow (o.att\_val(a) = void \vee h.refs.has(o.att\_val(a)))$

$\quad$ **axioms**

$\quad\quad h.add\_obj(o).objs = h.objs \cup \{o\}$

$\quad\quad h.add\_obj(o).refs = h.refs \cup \{r\}$

$\quad\quad h.add\_obj(o).ref\_obj(r) = o$

$\quad\quad h.add\_obj(o).last\_added\_obj = o$

$\quad\quad$ **where**

$\quad\quad\quad r \overset{def}{=} new\, \mathbf{REF}.make$

If an object that is already part of the heap gets updated then it is necessary to update the mapping from the reference to the object on the heap. This can be done with the command *update_ref* that takes a reference r and an updated object o and returns a heap where the reference r points to o. The command requires that r is a valid reference and that o is an updated version of the original object. Because the remaining part of the state only deals with references rather than objects directly, a reference update does not require an update of these parts.

$update\_ref\colon \mathbf{HEAP} \rightarrow \mathbf{REF} \nrightarrow \mathbf{OBJ} \nrightarrow \mathbf{HEAP}$

$\quad h.update\_ref(r, o)$ **require**

$\quad\quad h.refs.has(r)$

$\quad\quad o.id = h.ref\_obj(r).id$

$\quad\quad \forall a \in o.class\_type.attributes\colon o.att\_val(a) \in \mathbf{REF} \rightarrow (o.att\_val(a) = void \vee h.refs.has(o.att\_val(a)))$

$\quad$ **axioms**

$\quad\quad h.update\_ref(r, o).objs.has(o)$

$\quad\quad o \neq h.ref\_obj(r) \rightarrow \neg h.update\_ref(r, o).objs.has(h.ref\_obj(r))$

$\quad\quad h.update\_ref(r, o).ref\_obj(r) = o$

$\quad\quad h.last\_added\_obj = h.ref\_obj(r) \rightarrow h.update\_ref(r, o).last\_added\_obj = o$

So far we have defined the mapping from references to objects. Occasionally it is necessary to have the inverse mapping. The commands *add_obj* and *update_ref* ensure that there is a exactly one reference for every object on the heap. Thus it is possible to define the inverse query *ref* as an auxiliary query.

$ref\colon \mathbf{HEAP} \rightarrow \mathbf{OBJ} \nrightarrow \mathbf{REF}$

$\quad h.ref(o)$ **require**

$\quad\quad h.objs.has(o)$

$\quad$ **axioms**

$\quad\quad h.ref\_obj(h.ref(o)) = o$

4.3.3. Once routines

A *once routine* can either be a once function or a once procedure. A once routine gets executed at most once in a certain context. If a once routine has been executed in a the context then it is called *non-fresh* in the context. Otherwise it is called *fresh* in the context. The context is either the set of all processors in the system or a single processor. We use the heap to remember which once routines are fresh. For this purpose, we refine the heap ADT and declare the queries *is_fresh* and *once_result*. For any processor p and any once routine f, the query *is_fresh* states whether f is fresh on p or not. For a once function f that is not fresh on a processor p the query *once_result* returns the result of f on p.

$\mathit{is\_fresh}\colon \mathbf{HEAP} \rightarrow \mathbf{PROC} \rightarrow \mathbf{FEATURE} \nrightarrow \mathbf{BOOLEAN}$

$\quad h.\mathit{is\_fresh}(p, f)$ **require**

$\qquad f.\mathit{is\_once}$

$\mathit{once\_result}\colon \mathbf{HEAP} \rightarrow \mathbf{PROC} \rightarrow \mathbf{FEATURE} \nrightarrow \mathbf{REF}$

$\quad h.\mathit{once\_result}(p, f)$ **require**

$\qquad f \in \mathbf{FUNCTION} \wedge f.\mathit{is\_once}$

$\qquad \neg h.\mathit{is\_fresh}(p, f)$

We declare two commands to change the once status of a fresh once routine to non-fresh. One version is used for once functions and the other one for once procedures. Both commands take the once routine f and the processor p. The version for once functions also takes a once result r. The two commands implement the semantics for once routines: A once routine has either a once per system or a once per processor semantics. Once functions declared as separate with or without an explicit processor specification have the once per system semantics. In this case, the command *set_once_func_not_fresh* defines f as non-fresh on all processors. Once functions with a non-separate result type have the once per processor semantics. In this case, the command *set_once_func_not_fresh* sets f as non-fresh on p with the once result r. Once procedures have the once per processor semantics. In this case, the command *set_once_proc_not_fresh* sets f as non-fresh on p. We use the typing environment Γ to get the type of once routines.

$\mathit{set\_once\_func\_not\_fresh}\colon \mathbf{HEAP} \rightarrow \mathbf{PROC} \rightarrow \mathbf{FEATURE} \nrightarrow \mathbf{REF} \nrightarrow \mathbf{HEAP}$

$\quad h.\mathit{set\_once\_func\_not\_fresh}(p, f, r)$ **require**

$\qquad f \in \mathbf{FUNCTION} \wedge f.\mathit{is\_once}$

$\qquad r \neq \mathit{void} \rightarrow h.\mathit{refs}.\mathit{has}(r)$

$\quad$ **axioms**

$\qquad (\exists d, c\colon \Gamma \vdash f : (d, \bullet, c)) \rightarrow$

$\qquad\quad h.\mathit{set\_once\_func\_not\_fresh}(p, f, r).\mathit{is\_fresh}(p, f) = \mathit{false} \wedge$

$\qquad\quad h.\mathit{set\_once\_func\_not\_fresh}(p, f, r).\mathit{once\_result}(p, f) = r$

$\qquad (\exists d, c\colon \Gamma \vdash f : (d, p, c) \wedge p \neq \bullet) \rightarrow \forall q \in \mathbf{PROC}\colon$

$\qquad\quad h.\mathit{set\_once\_func\_not\_fresh}(p, f, r).\mathit{is\_fresh}(q, f) = \mathit{false} \wedge$

$\qquad\quad h.\mathit{set\_once\_func\_not\_fresh}(p, f, r).\mathit{once\_result}(q, f) = r$

$\mathit{set\_once\_proc\_not\_fresh}\colon \mathbf{HEAP} \rightarrow \mathbf{PROC} \rightarrow \mathbf{FEATURE} \nrightarrow \mathbf{HEAP}$

$\quad h.\mathit{set\_once\_proc\_not\_fresh}(p, f)$ **require**

$\qquad f \in \mathbf{PROCEDURE} \wedge f.\mathit{is\_once}$

$\quad$ **axioms**

$\qquad h.\mathit{set\_once\_proc\_not\_fresh}(p, f).\mathit{is\_fresh}(p, f) = \mathit{false}$

4.3.4. Creation

A new heap can be created with the constructor *make*. A new heap has no objects and no references. All once routines are marked as fresh on all processors. Note that we do not make any assumptions on processors, as the heap does not know about the processors in the system. Later on, we take a look at how to guarantee consistency between the heap and the regions.

$make$: **HEAP**

axioms

$make.objs.is\_empty$

$make.refs.is\_empty$

$\forall p \in \mathbf{PROC}, f \in \mathbf{FEATURE}\colon f.is\_once \rightarrow make.is\_fresh(p, f) = true$

4.4. Regions ADT

We partition the heap into disjunct *regions* and assign each region to exactly one processor. This concept relates to the concept of a ken in Schmidt's work on reasoning about concurrent objects [28]. The processor of a region is the handler of all the objects in the region. Regions are also used to maintain locks. In the following we first describe an ADT for processor and then use it to describe an ADT for regions.

4.4.1. Processors

A *processor* is an autonomous thread of control capable of executing features on objects. Every processor is responsible for a set of objects. As such a processor is called the handler of its associated objects. Every object is assigned to exactly one processor that is the authority of feature executions on this object. If a processor q wants to call a feature on an object handled by a different processor p then q needs to send a feature request to processor p. This is where the request queue of processor p comes into place. The *request queue* keeps track of features to be executed on behalf of other processors. Processor q can add a request to this queue and processor p will execute the request as soon as it executed all previous requests in the request queue. Processor p uses its *call stack* to execute the feature request at the beginning of the request queue. The call stack is responsible for the order of feature executions on the same processor. In a situation of a non-separate call, the call stack ensures that the calling feature execution resumes once the called feature execution terminated. The interaction between the call stack and the request queue is best described with the following loop through which every processor goes:

1. Idle wait: If both the call stack and the request queue are empty then wait for new requests to be enqueued.
2. Request scheduling: If the call stack is empty but the request queue is not empty then dequeue an item and push it onto the call stack.
3. Request processing: If there is an item on the call stack then pop the item from the call stack and process it. If the item is a feature request then apply the feature. If the items is an operation then execute the operation.

For every processor there is a *request queue lock* and a *call stack lock*. A lock on the request queue grants permission to add a feature request to the end of the request queue. A lock on the call stack grants permission to add a feature request to the top of the call stack. Later on we formalize the call stack and the request queue as we introduce action queues. Before processor q can add a request to p's request queue, it must have a lock on this request queue. Otherwise another processor could intervene. Once processor q is done with the request queue of processor p it can add an unlock operation to the end of the request queue. This makes sure that the request queue lock of p will be released after all the previous feature requests have been executed. Similarly, processor p must have a lock on its call stack to add features to its call stack. Initially, every processor has a lock on its own call stack and its request queue is not locked.

Processor q could also make a synchronous call to p. However q might be in possession of some locks that are necessary for the execution the resulting feature request on p. In such a situation, q is waiting for the synchronous call to terminate and p is waiting for locks to be available. According to the conditions given by Coffman et al [8] a deadlock occurred. This can be avoided if q temporarily passes its locks to the p. This allows p to finish the execution and hence q can continue. This operation is called lock passing, which we discuss in more details later on.

Clarification 1 (Request queue locks and call stack locks) The notion of request queue locks and call stack locks was not present in Nienaltowski's [22] definition of SCOOP. He defines one lock for each processor. A lock on a processor means exclusive access to the whole processor. This lock model is not sufficient to describe SCOOP. [1] In particular, this lock model creates a contradiction with respect to separate callbacks. A separate callback is a feature call in which processor q made a direct or indirect call to processor p and now p is calling back processor q. The separate callback is only possible, if p has a lock on q. However, p does not necessarily have this lock because the lock might be in possession of the processor that locked q in the first place. Request queue locks and call stack locks allow us to clarify the situation. Thus we propose a new lock model with request queue locks and call stack locks.

We consider the lock model used in Nienaltowski's work [22] as an abstraction of our lock model. The abstraction works under the assumption that no processor passes its locks. Under this assumption every processor keeps its call stack lock. In this abstraction we call the request queue lock on a processor p simply the lock on p. As long as the call stack lock on a processor p is in possession of p, a request queue lock on p in possession of a processor q means that processor p will be executing new feature requests in the request queue exclusively on behalf of q. This means that a request queue lock grants exclusive access to all the objects handled by p. Transferring this insight to our abstractions, a lock on processor p denotes exclusive access to the objects handled by p. $\square$

After this informal explanation of processors, we introduce an ADT for processors. A processor only has an identifier stored in the query *id*.

$id\colon \mathbf{PROC} \to \mathbf{ID}$

The constructor *make* returns a new processor with a fresh identifier. The fresh identifier is defined through the query *new_id*.

$make\colon \mathbf{PROC}$

axioms

$make.id = new\_id$

The ADT for processors is very simple. It neither takes care of the mapping from processors to the handled objects nor does it take care of the locks. These aspects are taken care of by the ADT for regions. The reason for this design decision comes once again from the fact that updating an ADT instance creates a new ADT instance that could cause a cascade of updates. Unfortunately, we do not have the notion of references for processors and therefore we cannot apply the technique that helped us for objects. Our solution is to make processors immutable and to deal with the mentioned aspects in the regions ADT where an update is simple because regions are stored in a single query in the state.

4.4.2. Mapping of processors to objects and locking

In this section, we introduce an ADT for regions. The regions ADT declares a query *procs* that keeps track of all the processors in the system. The query *handled_objs* defines a set of handled objects for each processor in *procs*. Finally, the query *last_added_proc* denotes the last processor that has been added to *procs*.

$procs\colon \mathbf{REGIONS} \to \mathbf{SET}[\mathbf{PROC}]$

$handled\_objs\colon \mathbf{REGIONS} \to \mathbf{PROC} \nrightarrow \mathbf{SET}[\mathbf{OBJ}]$

$k.handled\_objs(p)$ **require**

$k.procs.has(p)$

[1] We thank Stephan van Staden for interesting discussions on this topic.

$last\_added\_proc$: **REGIONS** $\nrightarrow$ **PROC**

  $k.last\_added\_proc$ **require**

    $\neg k.procs.is\_empty$

Next to the queries that are concerned with the mapping from processors to objects, there are a number of queries that deal with locking. The feature *is_rq_locked* states whether the request queue of a processor in *procs* is locked or not. Similarly, the feature *is_cs_locked* states whether the call stack is locked.

The remaining queries are used to specify the owners of the locks. For this, we distinguish between *obtained* and *retrieved* locks. Obtained locks are locks that got acquired by a processor. Retrieved locks are locks that got passed from another processor.

The query *obtained_rq_locks* returns a stack of obtained processor sets for a processor. We use a stack of sets to model the way processors acquire locks: They go through a nested series of feature applications and each feature application requires a set of locks before the feature can be executed. For each feature application we add a new set on top of the stack and as soon as the feature application finished, the top set gets removed from the stack. The query *obtained_cs_lock* returns the acquired call stack lock of a processor. Initially every processor starts with a lock on its own call stack and this call stack lock never changes. Thus this query is only declared for reasons of completeness. Note that we use a processor itself to denote either its request queue lock or its call stack lock. If a processor appears in a set of request queue locks then the processor denotes its request queue lock. If a processor appears in a set of call stack locks then the processor denotes its call stack lock. We follow this scheme because we do not want to introduce an ADT for locks.

A processors can pass its locks to another processor. There are several queries to formalize this aspects. The features *retrieved_rq_locks* and *retrieved_cs_locks* return the retrieved locks of a processor. Both of these queries return a stack of sets. The stack is used to keep track of the set of retrieved locks for each feature application. These two stacks grow and shrink in parallel to the stack *obtained_rq_locks*. Once a processor passed its locks, it cannot use them anymore until the locks are revoked. The query *are_locks_passed* returns whether a processor passed some or all of its locks or not.

is_rq_locked: **REGIONS** $\rightarrow$ **PROC** $\nrightarrow$ **BOOLEAN**

  $k.is\_rq\_locked(p)$ **require**

    $k.procs.has(p)$

is_cs_locked: **REGIONS** $\rightarrow$ **PROC** $\nrightarrow$ **BOOLEAN**

  $k.is\_cs\_locked(p)$ **require**

    $k.procs.has(p)$

$obtained\_rq\_locks$: **REGIONS** $\rightarrow$ **PROC** $\nrightarrow$ **STACK**[**SET**[**PROC**]]

  $k.obtained\_rq\_locks(p)$ **require**

    $k.procs.has(p)$

$obtained\_cs\_lock$: **REGIONS** $\rightarrow$ **PROC** $\nrightarrow$ **PROC**

  $k.obtained\_cs\_lock(p)$ **require**

    $k.procs.has(p)$

$retrieved\_rq\_locks$: **REGIONS** $\rightarrow$ **PROC** $\nrightarrow$ **STACK**[**SET**[**PROC**]]

  $k.retrieved\_rq\_locks(p)$ **require**

    $k.procs.has(p)$

$retrieved\_cs\_locks$: **REGIONS** $\rightarrow$ **PROC** $\nrightarrow$ **STACK**[**SET**[**PROC**]]

  $k.retrieved\_cs\_locks(p)$ **require**

    $k.procs.has(p)$

are_locks_passed: **REGIONS** $\rightarrow$ **PROC** $\nrightarrow$ **BOOLEAN**

$k.are\_locks\_passed(p)$ **require**

$k.procs.has(p)$

We study these queries in more details as we discuss the commands that operate on them. In the following we first go through the list of commands that add processors and commands that change the association of processors to objects. We then proceed with the commands that handle locks.

The command *add_proc* updates the regions with a new processor. Note that the processor must have been created beforehand. The axioms state that the new processor will be included in *procs* and that it will be stored in *last_added_proc*. The axioms also state how the new processor is initialized. The new processor's request queue is unlocked and its call stack is locked. Apart from the initial lock on the call stack there are no obtained or retrieved locks and hence the processor did not pass its locks.

add_proc: **REGIONS** $\rightarrow$ **PROC** $\nrightarrow$ **REGIONS**

$k.add\_proc(p)$ **require**

$\neg k.procs.has(p)$

axioms

$k.add\_proc(p).procs.has(p)$

$k.add\_proc(p).last\_added\_proc = p$

$k.add\_proc(p).handled\_objs(p).is\_empty$

$k.add\_proc(p).is\_rq\_locked(p) = false$

$k.add\_proc(p).is\_cs\_locked(p) = true$

$k.add\_proc(p).obtained\_rq\_locks(p).is\_empty$

$k.add\_proc(p).obtained\_cs\_lock(p) = p$

$k.add\_proc(p).retrieved\_rq\_locks(p).is\_empty$

$k.add\_proc(p).retrieved\_cs\_locks(p).is\_empty$

$k.add\_proc(p).are\_locks\_passed(p) = false$

The command *add_obj* takes a processor p in *procs* and an object o that is not handled by a processor in *procs* yet. It returns the updated regions in which o is handled by p.

add_obj: **REGIONS** $\rightarrow$ **PROC** $\nrightarrow$ **OBJ** $\nrightarrow$ **REGIONS**

$k.add\_obj(p, o)$ **require**

$k.procs.has(p)$

$\forall q \in k.procs, u \in k.handled\_objs(q)\colon u.id \neq o.id$

axioms

$k.add\_obj(p, o).handled\_objs(p).has(o)$

In the opposite direction, the command *remove_obj* removes an object that is handled by a processor in *procs* from the regions.

$remove\_obj$: **REGIONS** $\rightarrow$ **OBJ** $\nrightarrow$ **REGIONS**

$k.remove\_obj(o)$ **require**

$\exists p \in k.procs\colon k.handled\_objs(p).has(o)$

axioms

$\neg \exists p \in k.procs\colon k.remove\_obj(o).handled\_objs(p).has(o)$

In the following, we discuss the commands that deal with the locking aspects of the regions. We declare a command *lock_rqs* to lock the request queues of a set of processors $\overline{q}$ on behalf of a processor p. None of these request queues must be locked beforehand.

$\mathit{lock\_rqs}\colon \textbf{REGIONS} \to \textbf{PROC} \nrightarrow \textbf{SET}[\textbf{PROC}] \nrightarrow \textbf{REGIONS}$

$\quad k.\mathit{lock\_rqs}(p, \bar{l})$ **require**

$\qquad k.\mathit{procs}.\mathit{has}(p)$

$\qquad \forall x \in \bar{l}\colon k.\mathit{procs}.\mathit{has}(x)$

$\qquad \forall x \in \bar{l}\colon k.\mathit{is\_rq\_locked}(x) = \mathit{false}$

$\quad$ **axioms**

$\qquad k.\mathit{lock\_rqs}(p, \bar{l}).\mathit{obtained\_rq\_locks}(p) = k.\mathit{obtained\_rq\_locks}(p).\mathit{push}(\bar{l})$

$\qquad \forall x \in \bar{l}\colon k.\mathit{lock\_rqs}(p, \bar{l}).\mathit{is\_rq\_locked}(x)$

At some point, processor p will not require the obtained request queue locks anymore because p made sure to enqueue all necessary features requests. The command *pop_obtained_rq_locks* is used by processor p to remove his claims on the obtained request queue locks. This requires that processor p is in possession of these locks, i.e. that p did not pass its locks.

$\mathit{pop\_obtained\_rq\_locks}\colon \textbf{REGIONS} \to \textbf{PROC} \nrightarrow \textbf{REGIONS}$

$\quad k.\mathit{pop\_obtained\_rq\_locks}(p)$ **require**

$\qquad k.\mathit{procs}.\mathit{has}(p)$

$\qquad \neg k.\mathit{obtained\_rq\_locks}(p).\mathit{is\_empty}$

$\qquad k.\mathit{are\_locks\_passed}(p) = \mathit{false}$

$\quad$ **axioms**

$\qquad k.\mathit{pop\_obtained\_rq\_locks}(p).\mathit{obtained\_rq\_locks}(p) = k.\mathit{obtained\_rq\_locks}(p).\mathit{pop}$

Removing the locks from p's obtained request queue locks stack does not mean that these request queues are unlocked. It just means that the request queue locks are not claimed by p anymore and therefore p will not enqueue further feature requests on the respective processors. The request queues remain locked until they get unlocked with a call to the command *unlock_rq*. As we discuss later, this happens after the processors whose request queues got locked by p finished all the requested feature applications. The precondition of the command states that a request queue can only be unlocked if it is not claimed by any other processor. This precondition guarantees that the request queue can only be unlocked when it is not used as an obtained or retrieved lock by any other processor anymore. Note that there is no unlock command for call stack locks because the call stack never gets unlocked.

$\mathit{unlock\_rq}\colon \textbf{REGIONS} \to \textbf{PROC} \nrightarrow \textbf{REGIONS}$

$\quad k.\mathit{unlock\_rq}(p)$ **require**

$\qquad k.\mathit{procs}.\mathit{has}(p)$

$\qquad k.\mathit{is\_rq\_locked}(p) = \mathit{true}$

$\qquad \forall q \in k\colon \neg k.\mathit{obtained\_rq\_locks}(q).\mathit{flat}.\mathit{has}(p)$

$\quad$ **axioms**

$\qquad k.\mathit{unlock\_rq}(p).\mathit{is\_rq\_locked}(p) = \mathit{false}$

We just saw that the request queues remain locked until explicitly unlocked with a call to *unlock_rq*. Between the call to *pop_obtained_rq_locks* and the call to *unlock_rq*, the owner of these locks is undefined. In some situations this is not satisfactory. A different solution must be found if another processor wants to claim the locks until they are unlocked. We declare the command *delegate_obtained_rq_locks* to serve for this purpose. The command takes a processor p and a number of processors $\bar{l}$ and makes p the owner of the request queue locks of all processors in $\bar{l}$ by adding these locks to the obtained request queue locks stack of p. This can only work if there is no current owner and the request queues are indeed locked.

$$\mathit{delegate\_obtained\_rq\_locks}\colon \textbf{REGIONS} \to \textbf{PROC} \nrightarrow \textbf{SET}[\textbf{PROC}] \nrightarrow \textbf{REGIONS}$$

$k.\mathit{delegate\_obtained\_rq\_locks}(p, \bar{l})$ **require**

$k.\mathit{procs}.\mathit{has}(p)$

$\forall x \in \bar{l}\colon k.\mathit{procs}.\mathit{has}(x)$

$\forall x \in \bar{l}\colon \neg\exists y \in k.\mathit{procs}\colon k.\mathit{obtained\_rq\_locks}(y).\mathit{flat}.\mathit{has}(x)$

$\forall x \in \bar{l}\colon k.\mathit{is\_rq\_locked}(x) = \mathit{true}$

axioms

$k.\mathit{delegate\_obtained\_rq\_locks}(p, \bar{l}).\mathit{obtained\_rq\_locks}(p) = k.\mathit{obtained\_rq\_locks}(p).\mathit{push}(\bar{l})$

Delegation is different from lock passing: *delegation* is the permanent transfer of ownership and *lock passing* is the temporary transfer of the right to use the locks. In the following, we look at the commands to pass and revoke locks. The command *pass_locks* takes a processor p and a processor q as well as a set of request queue locks $\overline{l_r}$ along with a set of call stack locks $\overline{l_c}$. The result is an updated instance of the **REGIONS** ADT in which $\overline{l_r}$ and $\overline{l_c}$ have been passed from p to q. As a precondition for this task, processor p must be in possession of all these locks. This means that all the locks in $\overline{l_r}$ and $\overline{l_c}$ must be obtained or retrieved locks of p and the locks must not be passed. In addition, we must remember that some or all of p's locks have been passed. However, because the two sets of locks can potentially be empty, we only mark p's locks as passed, if at least one of the two sets of locks is non-empty. Lastly, we must take care of one special case of the lock passing operation. If a processor q different from processor p passed its locks in a previous lock passing operation and now we pass these locks back to q then we have to mark the locks of processor q as not passed. As we discuss later on, this case is important to handle separate callbacks.

$$\mathit{pass\_locks}\colon \textbf{REGIONS} \to \textbf{PROC} \nrightarrow \textbf{PROC} \nrightarrow \textbf{TUPLE}[\textbf{SET}[\textbf{PROC}], \textbf{SET}[\textbf{PROC}]] \nrightarrow \textbf{REGIONS}$$

$k.\mathit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c}))$ **require**

$k.\mathit{procs}.\mathit{has}(p) \wedge k.\mathit{procs}.\mathit{has}(q)$

$\forall x \in \overline{l_r}\colon k.\mathit{procs}.\mathit{has}(x) \wedge \forall x \in \overline{l_c}\colon k.\mathit{procs}.\mathit{has}(x)$

$\forall x \in \overline{l_r}\colon k.\mathit{obtained\_rq\_locks}(p).\mathit{flat}.\mathit{has}(x) \vee k.\mathit{retrieved\_rq\_locks}(p).\mathit{flat}.\mathit{has}(x)$

$\forall x \in \overline{l_c}\colon x = k.\mathit{obtained\_cs\_lock}(p) \vee k.\mathit{retrieved\_cs\_locks}(p).\mathit{flat}.\mathit{has}(x)$

$k.\mathit{are\_locks\_passed}(p) = \mathit{false}$

axioms

$$k.\mathit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c})).\mathit{are\_locks\_passed}(p) = \begin{cases} \mathit{true} & \mathit{if} \neg\overline{l_r}.\mathit{is\_empty} \vee \neg\overline{l_c}.\mathit{is\_empty} \\ \mathit{false} & \mathit{otherwise} \end{cases}$$

$k.\mathit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c})).\mathit{retrieved\_rq\_locks}(q) = k.\mathit{retrieved\_rq\_locks}(q).\mathit{push}(\overline{l_r})$

$k.\mathit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c})).\mathit{retrieved\_cs\_locks}(q) = k.\mathit{retrieved\_cs\_locks}(q).\mathit{push}(\overline{l_c})$

$$\begin{pmatrix} p \neq q \wedge \\ k.\mathit{are\_locks\_passed}(q) = \mathit{true} \wedge \\ k.\mathit{obtained\_rq\_locks}(q).\mathit{flat} \subseteq \overline{l_r} \wedge \\ k.\mathit{retrieved\_rq\_locks}(q).\mathit{flat} \subseteq \overline{l_r} \wedge \\ k.\mathit{obtained\_cs\_lock}(q) \in \overline{l_c} \wedge \\ k.\mathit{retrieved\_cs\_locks}(q).\mathit{flat} \subseteq \overline{l_c} \end{pmatrix} \to k.\mathit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c})).\mathit{are\_locks\_passed}(q) = \mathit{false}$$

The command *revoke_locks* takes a processor p and a processor q. It reverses the effect of a lock passing operation from a processor p to q and returns an updated instance of the k ADT. This is only allowed, if processor p passed locks to q in a preceding lock passing operation. Note that the lock passing operation from p to q potentially marked the locks of q as not passed after they were marked as passed before the lock passing operation. Revoking the locks from q to p requires the reverse action. If p has retrieved locks in common with the locks of q, even after the retrieved locks from p have been removed from q, then q's locks must be marked as passed because they are now in possession of p.

$$
\begin{array}{l}
\textit{revoke\_locks}\colon \mathbf{REGIONS} \to \mathbf{PROC} \nrightarrow \mathbf{PROC} \nrightarrow \mathbf{REGIONS}\\
\quad k.\textit{revoke\_locks}(p,q) \textbf{ require}\\
\qquad k.\textit{procs}.\textit{has}(p) \wedge k.\textit{procs}.\textit{has}(q)\\
\qquad \neg k.\textit{retrieved\_rq\_locks}(q).\textit{is\_empty} \wedge \neg k.\textit{retrieved\_cs\_locks}(q).\textit{is\_empty}\\
\qquad k.\textit{retrieved\_rq\_locks}(q).\textit{top} \subseteq k.\textit{obtained\_rq\_locks}(p).\textit{flat} \cup k.\textit{retrieved\_rq\_locks}(p).\textit{flat}\\
\qquad k.\textit{retrieved\_cs\_locks}(q).\textit{top} \subseteq \{k.\textit{obtained\_cs\_lock}(p)\} \cup k.\textit{retrieved\_cs\_locks}(p).\textit{flat}\\
\qquad k.\textit{retrieved\_rq\_locks}(q).\textit{top} \cup k.\textit{retrieved\_cs\_locks}(q).\textit{top} \neq \{\} \to k.\textit{are\_locks\_passed}(p) = \textit{true}\\
\qquad k.\textit{are\_locks\_passed}(q) = \textit{false}\\
\quad \textbf{axioms}\\
\qquad k.\textit{revoke\_locks}(p,q).\textit{are\_locks\_passed}(p) = \textit{false}\\
\qquad k.\textit{revoke\_locks}(p,q).\textit{retrieved\_rq\_locks}(q) = k.\textit{retrieved\_rq\_locks}(q).\textit{pop}\\
\qquad k.\textit{revoke\_locks}(p,q).\textit{retrieved\_cs\_locks}(q) = k.\textit{retrieved\_cs\_locks}(q).\textit{pop}\\
\qquad \left(\begin{array}{l}
p \neq q \wedge\\
\left(\begin{array}{l}
\exists x \in k.\textit{retrieved\_rq\_locks}(p).\textit{flat}\colon (\\
\quad k.\textit{obtained\_rq\_locks}(q).\textit{flat}.\textit{has}(x) \vee\\
\quad k.\textit{retrieved\_rq\_locks}(q).\textit{pop}.\textit{flat}.\textit{has}(x)\\
) \vee\\
\exists x \in k.\textit{retrieved\_cs\_locks}(p).\textit{flat}\colon (\\
\quad x = k.\textit{obtained\_cs\_lock}(q) \vee\\
\quad k.\textit{retrieved\_cs\_locks}(q).\textit{pop}.\textit{flat}.\textit{has}(x)\\
)
\end{array}\right)
\end{array}\right)\\
\qquad\quad \to k.\textit{revoke\_locks}(p,q).\textit{are\_locks\_passed}(q) = \textit{true}
\end{array}
$$

With these commands, we can wrap up the mapping of processors to objects and the locking aspects of the ADT for regions. We are now at the point to introduce a number of auxiliary queries to simplify access to the presented queries. We saw that the command *add_obj* makes sure that a processor is assigned to each object that gets added. This mapping is available through the query *handled_objs*. Thus it is possible to define an auxiliary query *handler* that is inverse to the query *handled_objs* in a sense that whenever an object o is in the set of handled objects for a processor p then *handler* will return p for o.

We introduced four different categories of locks that each processor can have. For both the request queue locks and the call stack locks, we have queries for obtained and retrieved locks. In some situations it is easier to just work with request queue locks and call stack locks without splitting them into obtained and retrieved locks. The auxiliary queries *rq_locks* and *cs_locks* serve this purpose. The auxiliary query *rq_locks* returns a set that contains all the obtained and the retrieved request queue locks of a processor p. Similarly, the auxiliary query *cs_locks* returns all the call stack locks of a processor p.

$$
\begin{array}{l}
\textit{handler}\colon \mathbf{REGIONS} \to \mathbf{OBJ} \nrightarrow \mathbf{PROC}\\
\quad k.\textit{handler}(o) \textbf{ require}\\
\qquad \exists p \in k.\textit{procs}\colon k.\textit{handled\_objs}(p).\textit{has}(o)\\
\quad \textbf{axioms}\\
\qquad k.\textit{handled\_objs}(k.\textit{handler}(o)).\textit{has}(o)
\end{array}
$$

rq_locks: $\mathbf{REGIONS} \rightarrow \mathbf{PROC} \nrightarrow \mathbf{SET}[\mathbf{PROC}]$

  $k.rq\_locks(p)$ **require**

    $k.procs.has(p)$

  axioms

    $k.rq\_locks(p) = k.obtained\_rq\_locks(p).flat \cup k.retrieved\_rq\_locks(p).flat$

cs_locks: $\mathbf{REGIONS} \rightarrow \mathbf{PROC} \nrightarrow \mathbf{SET}[\mathbf{PROC}]$

  $k.cs\_locks(p)$ **require**

    $k.procs.has(p)$

  axioms

    $k.cs\_locks(p) = \{k.obtained\_cs\_lock(p)\} \cup k.retrieved\_cs\_locks(p).flat$

4.4.3. Creation

We declare a single constructor *make* that creates a new instance of the **REGIONS** ADT. The new instance has no processors.

$make$: $\mathbf{REGIONS}$

  axioms

    $make.procs.is\_empty$

4.5. Store ADT

Each processor in the system has a call stack to execute features. Every time a processor executes a feature, a new call stack frame gets created on top of the call stack. The new call stack frame is used to store the values of formal arguments, local variables, the current object entity, and the result entity for the current feature execution. The call stack is also responsible for the order of feature executions on the same processor. In our formalization, we separate the two concerns of the call stack. As we will see later, we use the action queue of a processor to model the ordering of feature executions. We use the *store* to model the values in each stack frame. A store has a stack of environments for each processor, where each *environment* maps names to values. We first present an ADT for environments and then we present an ADT for the store.

4.5.1. Environments

The **ENV** ADT has a query *names* that stores all the defined names. The query *val* can then be used to get the value for each such name.

$names$: $\mathbf{ENV} \rightarrow \mathbf{SET}[\mathbf{NAME}]$

val: $\mathbf{ENV} \rightarrow \mathbf{NAME} \nrightarrow \mathbf{REF} \cup \mathbf{PROC}$

  $e.val(n)$ **require**

    $e.names.has(n)$

The command *update* takes a name and a value and returns an updated environment. Note that it does not matter whether the name is already defined in the environment or not. In any case, the name will be defined in the updated environment and the name will be mapped to the value. The value can either be a reference or a processor. Environments with processor values are not strictly needed to describe SCOOP, however they will make it possible to have a unified view on attribute values and environment values.

$\mathit{update}\colon \mathbf{ENV} \to \mathbf{NAME} \to \mathbf{REF} \cup \mathbf{PROC} \to \mathbf{ENV}$

axioms

$e.\mathit{update}(n, v).\mathit{names} = e.\mathit{names} \cup \{n\}$

$e.\mathit{update}(n, v).\mathit{val}(n) = v$

The constructor *make* returns an empty environment.

$\mathit{make}\colon \mathbf{ENV}$

axioms

$\mathit{make}.\mathit{names}.\mathit{is\_empty}$

4.5.2. Mapping from processors to environments

The store ADT has a single query *envs* that stores a stack of environments for each processor. Note that we do not make restrictions on the processors because for modularity reasons the store does not know about the regions. The consistency between the store and the regions are expressed in the state ADT.

$\mathit{envs}\colon \mathbf{STORE} \to \mathbf{PROC} \to \mathbf{STACK}[\mathbf{ENV}]$

The command *push_env* pushes a given environment on top a processor's stack of environments. The command *pop_env* pops the top environment from a non-empty stack of environments.

$\mathit{push\_env}\colon \mathbf{STORE} \to \mathbf{PROC} \to \mathbf{ENV} \to \mathbf{STORE}$

axioms

$s.\mathit{push\_env}(p, e).\mathit{envs}(p) = s.\mathit{envs}(p).\mathit{push}(e)$

$\mathit{pop\_env}\colon \mathbf{STORE} \to \mathbf{PROC} \not\to \mathbf{STORE}$

$s.\mathit{pop\_env}(p)$ **require**

$\neg s.\mathit{envs}(p).\mathit{is\_empty}$

axioms

$s.\mathit{pop\_env}(p).\mathit{envs}(p) = s.\mathit{envs}(p).\mathit{pop}$

The constructor *make* creates an empty store.

$\mathit{make}\colon \mathbf{STORE}$

axioms

$\forall p \in \mathbf{PROC}\colon \mathit{make}.\mathit{envs}(p).\mathit{is\_empty}$

4.6. State ADT

The ADT for the *state* has three queries to retrieve the different parts of the ADT.

$\mathit{regions}\colon \mathbf{STATE} \to \mathbf{REGIONS}$

$\mathit{heap}\colon \mathbf{STATE} \to \mathbf{HEAP}$

$\mathit{store}\colon \mathbf{STATE} \to \mathbf{STORE}$

We use one command *set* to set the regions, the heap, and the store at the same time. This has the advantage that we can use the precondition to specify consistency criteria between the parts of the state. For instance, we require that the regions and the heap deal with the same set of references, objects and processors. The first precondition clause states that a processor can handle an object if and only if the object is on the heap. The second precondition clause states that if the heap declares a feature as non-fresh on a processor p then the regions must know about this processor. The third precondition clause requires that all processors stored in attribute values are known by the regions. Note that we already require in the heap ADT that all references stored in attribute values are known. The forth precondition clause states that every non-empty environment in the store must belong to a processor that is known by the regions. The fifth precondition clause states that every value in the store must either be a know reference or a known processor.

$$
\begin{aligned}
&\mathit{set}\colon \mathbf{STATE} \to \mathbf{REGIONS} \nrightarrow \mathbf{HEAP} \nrightarrow \mathbf{STORE} \nrightarrow \mathbf{STATE}\\
&\quad \sigma.\mathit{set}(k, h, s)\ \mathbf{require}\\
&\qquad \exists p \in k.\mathit{procs}, \exists o \in \mathbf{OBJ}\colon k.\mathit{handled\_objs}(p).\mathit{has}(o) \leftrightarrow h.\mathit{objs}.\mathit{has}(o)\\
&\qquad \exists p \in \mathbf{PROC}, f \in \mathbf{FEATURE}\colon \neg h.\mathit{is\_fresh}(p, f) \to k.\mathit{procs}.\mathit{has}(p)\\
&\qquad \forall o \in h.\mathit{objs}, a \in o.\mathit{class\_type}.\mathit{attributes}\colon o.\mathit{att\_val}(a) \in \mathbf{PROC} \to k.\mathit{procs}.\mathit{has}(o.\mathit{att\_val}(a))\\
&\qquad \forall p \in \mathbf{PROC}, e \in s.\mathit{envs}(p)\colon \neg e.\mathit{names}.\mathit{is\_empty} \to k.\mathit{procs}.\mathit{has}(p)\\
&\qquad \forall p \in k.\mathit{procs}, e \in s.\mathit{envs}(p), x \in e.\mathit{names}\colon\\
&\qquad\quad (e.\mathit{val}(x) \in \mathbf{REF} \to e.\mathit{val}(x) = \mathit{void} \vee h.\mathit{refs}.\mathit{has}(e.\mathit{val}(x)))\wedge\\
&\qquad\quad (e.\mathit{val}(x) \in \mathbf{PROC} \to k.\mathit{procs}.\mathit{has}(e.\mathit{val}(x)))\\
&\quad \mathbf{axioms}\\
&\qquad \sigma.\mathit{set}(k, h, s).\mathit{regions} = k\\
&\qquad \sigma.\mathit{set}(k, h, s).\mathit{heap} = h\\
&\qquad \sigma.\mathit{set}(k, h, s).\mathit{store} = s
\end{aligned}
$$

4.6.1. Creation

To create a state, one has to create the three parts of the state. This is done with the constructor *make*.

$$
\begin{aligned}
&\mathit{make}\colon \mathbf{STATE}\\
&\quad \mathbf{axioms}\\
&\qquad \mathit{make}.\mathit{regions} = \mathit{new}\, \mathbf{REGIONS}.\mathit{make}\\
&\qquad \mathit{make}.\mathit{heap} = \mathit{new}\, \mathbf{HEAP}.\mathit{make}\\
&\qquad \mathit{make}.\mathit{store} = \mathit{new}\, \mathbf{STORE}.\mathit{make}
\end{aligned}
$$

4.6.2. Facade

Now we have a complete description of the state. However, it is too cumbersome to work with the **STATE** ADT as it is. For example, the following expression defines a new state σ' in which a new processor has been added to the state σ: $\sigma' \overset{def}{=} \sigma.\mathit{set}(\sigma.\mathit{regions}.\mathit{add\_proc}(\mathit{new}\, \mathbf{PROC}.\mathit{make}), \sigma.\mathit{heap}, \sigma.\mathit{store})$. This expression is too long for this simple task, especially if the expression is used multiple times. It would be easier to have an auxiliary command that does this job for us. To this end, we declare an abstraction with auxiliary features that provide easy access to the state functionality. We call this abstraction the *facade* and we locate the facade in the state ADT. We divide the facade into different aspects and dedicate one section to the description of each aspect. We start with the mapping of processors to objects and the mapping of references to objects. We continue with a section on how to set values, followed by a section on how to get values. We conclude with a section on locking.

4.6.3. Mapping of processors to objects and mapping of references to objects

The regions and the heap manage the references, the objects, the processors, and the mapping between them. The facade unifies all related features in one aspects. First, we define a number of auxiliary queries for the mapping of processors to objects. Next, we define auxiliary queries for the mapping of references to objects. We then define auxiliary commands that work on both aspects.

The two auxiliary queries *procs* and *last_added_proc* give access to all the processors and the last added processor.

$procs\colon \mathbf{STATE} \rightarrow \mathbf{SET}[\mathbf{PROC}]$

axioms

$\sigma.procs = \sigma.regions.procs$

$last\_added\_proc\colon \mathbf{STATE} \nrightarrow \mathbf{PROC}$

$\sigma.last\_added\_proc$ **require**

$\neg\sigma.regions.procs.is\_empty$

axioms

$\sigma.last\_added\_proc = \sigma.regions.last\_added\_proc$

The auxiliary query *handler* gives the handler of an object referenced by r. In contrast to the corresponding auxiliary query in the k ADT, the version here takes a reference instead of an object. The version in the k ADT deals directly with objects rather than references because it does not know about the heap and thus the mapping from references to objects is not available. The facade, however, has access to both the regions and the heap and thus it can use the preferred way of identifying objects: references. The *handler* of the facade takes a reference and uses the heap to get the referenced object. This object is then given to the regions to get the handler.

$handler\colon \mathbf{STATE} \rightarrow \mathbf{REF} \nrightarrow \mathbf{PROC}$

$\sigma.handler(r)$ **require**

$\sigma.heap.refs.has(r)$

axioms

$\sigma.handler(r) = \sigma.regions.handler(\sigma.heap.ref\_obj(r))$

We define the auxiliary query *new_proc* as a shorthand for processor creation.

$new\_proc\colon \mathbf{STATE} \rightarrow \mathbf{PROC}$

axioms

$\sigma.new\_proc = new\,\mathbf{PROC}.make$

The auxiliary query *last_added_obj* returns the object that has been added last to the heap. It uses the corresponding feature of the heap ADT.

$last\_added\_obj\colon \mathbf{STATE} \nrightarrow \mathbf{OBJ}$

$\sigma.last\_added\_obj$ **require**

$\neg\sigma.heap.objs.is\_empty$

axioms

$\sigma.last\_added\_obj = \sigma.heap.last\_added\_obj$

The auxiliary query *ref_obj* returns the object referenced by a reference r.

$\mathit{ref\_obj}\colon \mathbf{STATE} \to \mathbf{REF} \nrightarrow \mathbf{OBJ}$

$\quad \sigma.\mathit{ref\_obj}(r)$ **require**

$\qquad \sigma.\mathit{heap}.\mathit{refs}.\mathit{has}(r)$

$\quad$ **axioms**

$\qquad \sigma.\mathit{ref\_obj}(r) = \sigma.\mathit{heap}.\mathit{ref\_obj}(r)$

The auxiliary query *ref* returns the reference of an object o.

$\mathit{ref}\colon \mathbf{STATE} \to \mathbf{OBJ} \nrightarrow \mathbf{REF}$

$\quad \sigma.\mathit{ref}(o)$ **require**

$\qquad \sigma.\mathit{heap}.\mathit{objs}.\mathit{has}(o)$

$\quad$ **axioms**

$\qquad \sigma.\mathit{ref}(o) = \sigma.\mathit{heap}.\mathit{ref}(o)$

As a shorthand for object creation, we declare an auxiliary query *new_obj* that returns a new object with class type c.

$\mathit{new\_obj}\colon \mathbf{STATE} \to \mathbf{CLASS\_TYPE} \to \mathbf{OBJ}$

$\quad$ **axioms**

$\qquad \sigma.\mathit{new\_obj}(c) = \mathit{new}\, \mathbf{OBJ}.\mathit{make}(c)$

With this, we conclude the auxiliary queries in this section. We now take a look at auxiliary commands that modify the mapping of processors to objects and the mapping of references to objects. Before an object can be added to the set of handled objects of a processor, the processor must exist. If the processor does not exist yet, the command *add_proc* can be used to update a state with a new processor.

$\mathit{add\_proc}\colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{STATE}$

$\quad \sigma.\mathit{add\_proc}(p)$ **require**

$\qquad \neg\sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p)$

$\quad$ **axioms**

$\qquad \sigma.\mathit{add\_proc}(p) = \sigma.\mathit{set}(\sigma.\mathit{regions}.\mathit{add\_proc}(p), \sigma.\mathit{heap}, \sigma.\mathit{store})$

The auxiliary command *add_obj* can then be used to add an object to the processor and the heap. The auxiliary command takes a processor p and an object o and it returns a state in which object o is part of the heap and handled by processor p.

$\mathit{add\_obj}\colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{OBJ} \nrightarrow \mathbf{STATE}$

$\quad \sigma.\mathit{add\_obj}(p, o)$ **require**

$\qquad \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p)$

$\qquad \forall u \in \sigma.\mathit{heap}.\mathit{objs}\colon u.\mathit{id} \neq o.\mathit{id}$

$\qquad \forall a \in o.\mathit{class\_type}.\mathit{attributes}\colon$

$\qquad\quad (o.\mathit{att\_val}(a) \in \mathbf{REF} \to o.\mathit{att\_val}(a) = \mathit{void} \vee \sigma.\mathit{heap}.\mathit{refs}.\mathit{has}(o.\mathit{att\_val}(a)))\wedge$

$\qquad\quad (o.\mathit{att\_val}(a) \in \mathbf{PROC} \to \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(o.\mathit{att\_val}(a)))$

$\quad$ **axioms**

$\qquad \sigma.\mathit{add\_obj}(p, o) = \sigma.\mathit{set}(\sigma.\mathit{regions}.\mathit{add\_obj}(p, o), \sigma.\mathit{heap}.\mathit{add\_obj}(o), \sigma.\mathit{store})$

Next, we define an auxiliary command *update_ref* to update a reference with an updated object. The auxiliary command takes a reference r on the heap and an object o and it returns a state in which o replaced the object u referenced by r on the heap and in the regions. Note that o must indeed be an updated version of the object referenced by r. The auxiliary command first removes u from the set of handled objects and then adds o to the set of handled objects of u's handler. Then it updates the heap with the command *update_ref*, which is declared in the heap ADT.

```
$update\_ref : \mathbf{STATE} \to \mathbf{REF} \nrightarrow \mathbf{OBJ} \nrightarrow \mathbf{STATE}$
  $\sigma.update\_ref(r, o)$ require
    $\sigma.heap.refs.has(r)$
    $o.id = \sigma.heap.ref\_obj(r).id$
    $\forall a \in o.class\_type.attributes:$
      $(o.att\_val(a) \in \mathbf{REF} \to o.att\_val(a) = void \vee \sigma.heap.refs.has(o.att\_val(a))) \wedge$
      $(o.att\_val(a) \in \mathbf{PROC} \to \sigma.regions.procs.has(o.att\_val(a)))$
  axioms
    $\sigma.update\_ref(r, o) = \sigma.set(k, h, s)$
     where
      $u \overset{def}{=} \sigma.heap.ref\_obj(r)$
      $k \overset{def}{=} \sigma.regions.remove\_obj(u).add\_obj(\sigma.regions.handler(u), o)$
      $h \overset{def}{=} \sigma.heap.update\_ref(r, o)$
      $s \overset{def}{=} \sigma.store$
```

4.6.4. Setting values

In this section we take a look at how to set values. To start, we look at a prerequisite for this task: the deep import operation. Setting values includes setting values of formal arguments, values of local variables, the value of the current object entity, the value of the result entity, and attribute values of the current object. All of these values can be written and read without a feature call. We conclude this section with auxiliary commands to set the status of once routines. The SCOOP validity rules exclude other types of value setting operations.

Deep import operation Expanded objects have a copy semantics: If an object o of expanded class type is the source of an attachment then a copy u gets attached to the destination of the attachment. However, a shallow copy is not sufficient if o's handler p is different from u's handler q. If o has an attached non-separate entity then u now has a non-separate entity to which a separate object is attached. We would have introduced a *traitor* - a non-separate entity that points to a separate object. The SCOOP model, as defined by Nienaltowski [22], introduces the *import operation* to solve this issue. Applied to o the import operation creates a copied object structure that mirrors the original object structure in a way that o and all the objects reachable from o through non-separate references are replaced with copied objects that are handled by q. This data structure then gets attached to the destination of the attachment. The import operation computes the non-separate version of an object structure.

Clarification 2 (Deep import operation) The import operation potentially results in a copied object structure that contains both copied and original objects. This can be an issue in case one of the copied objects has an invariant over the identities of objects, as shown in example 4.6.4.

Example 3 (Invariant violation as a result of the import operation) Imagine two objects x and y handled by one processor and another object z handled by another processor. Object x has a separate entity a that points to z and a non-separate entity b that points to y. Object z has a separate entity c that points to y. Object x has an invariant with a query $a.c = b$. An import operation on x executed by a third processor will result in two new objects x' and y' on the third processor. The reference a of object x' will point to the

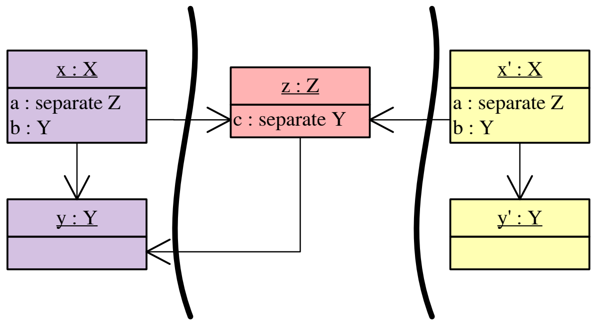

Figure 2. Invariant violation as a result of the import operation

original z. The reference b of object x' will point to the new object y'. This situation is illustrated in figure 2. Now object x' is inconsistent, because $a.c$ and b identify different objects, namely y and y'. □

The *deep import operation* is a variant of the import operation that does not mix the copied and the original objects. □

Instead of copying only the objects that are reachable through non-separate references, the deep import operation makes a full copy of the object structure. The deep importing processor handles all the copies of the objects that are non-separate with respect to the object to be imported. Each other separate object is handled by the processor of the respective original object. The deep import operation does not show the issue with invariants. The drawback of the deep import operation is that more objects must be copied. Nevertheless, we use the deep import operation in our formalization because we cannot tolerate violated invariants. Once routines complicate the deep import operation a bit. We consider a processor p that wants to deep import an object o handled by a different processor q. For every non-separate once function f of every copied object the following must be done: If a non-separate once function f is fresh on p and non-fresh on q then f must be marked as non-fresh on p and the value of f on q must be used as the value of f on p. If a once procedure f is fresh on p and non-fresh on q then f must be marked as non-fresh on p. In all other cases, nothing must be done.

We formalize the deep import operation as an auxiliary command *deep_import*. The command takes an importing processor p and a reference r to be imported. The command returns a state in which the copied object structure exists on the heap and the objects are associated to the respective processors. The copied object structure is accessible through the auxiliary query *last_imported_ref*.

$deep\_import\colon \textbf{STATE} \to \textbf{PROC} \nrightarrow \textbf{REF} \nrightarrow \textbf{STATE}$

$\quad \sigma.deep\_import(p, r)$ **require**

$\quad\quad \sigma.regions.procs.has(p)$

$\quad\quad \sigma.heap.refs.has(r)$

$\quad$ **axioms**

$\quad\quad \sigma.deep\_import(p, r) = \sigma'$

$\quad\quad \sigma.deep\_import(p, r).last\_imported\_ref = r'$

$\quad\quad$ **where**

$\quad\quad\quad w \overset{def}{=} new\, \textbf{MAP}[\textbf{REF}, \textbf{REF}].make$

$\quad\quad\quad (r', w', \sigma') \overset{def}{=} deep\_import\_rec\_with\_map(p, \sigma.handler(r), r, w, \sigma)$

The auxiliary command *deep_import* is based on an auxiliary function *deep_import_rec_with_map*. This function takes a tuple containing an importing processor p, a processor q that handles the root of the object structure to be imported, a reference r to be deep imported, and a state σ to be modified. Note that the object referenced by r is not necessarily handled by q because this object might be on a different processor than the handler of the root of the object structure to be deep imported. The function returns another tuple with a reference r'' to the copied object structure and an updated state σ''. The auxiliary function *deep_import_rec_with_map* works hand in hand with the auxiliary function *deep_import_rec_without_map*. They have the same signature and together they recursively traverse the object structure and make a deep copy of it. The functions must ensure that no object gets copied twice. For this purpose the functions take as an additional argument a map w that maps references to objects in the input data structure to references in the copied data structure. A mapping from one reference x to another reference y means that the object referenced by y is the copy of the object referenced by x. An updated map is returned as part of the result tuple. The auxiliary command *deep_import* starts the recursion with an empty map. The auxiliary function *deep_import_rec_with_map* uses the map to determine whether the object referenced by r has already been copied. In such a case, the result of the function comes from the map. Otherwise the auxiliary function *deep_import_rec_with_map* returns the result of the auxiliary function *deep_import_rec_without_map*. The auxiliary function *deep_import_rec_without_map* creates a copy of the object referenced by r and handles once routines. Finally, it returns a new reference r', an updated map w' in which r is mapped to r', and an updated state σ'.

$$\mathit{deep\_import\_rec\_with\_map}(p,q,r,w,\sigma) = (r'',w'',\sigma'')$$

where

$$(r',w',\sigma') \overset{def}{=} \mathit{deep\_import\_rec\_without\_map}(p,q,r,w,\sigma)$$

$$r'' \overset{def}{=} \begin{cases} w.val(r) & \mathit{if}\, w.keys.has(r) \\ r' & \mathit{if} \neg w.keys.has(r) \end{cases}$$

$$w'' \overset{def}{=} \begin{cases} w & \mathit{if}\, w.keys.has(r) \\ w' & \mathit{if} \neg w.keys.has(r) \end{cases}$$

$$\sigma'' \overset{def}{=} \begin{cases} \sigma & \mathit{if}\, w.keys.has(r) \\ \sigma' & \mathit{if} \neg w.keys.has(r) \end{cases}$$

The auxiliary function *deep_import_rec_without_map* is divided into several steps: a copy step, an attribute values update step, a clients update state, a once status update step, and a result generation step. Each of the steps has several definitions associated to it and each set of definitions depends on the definitions of the previous step. In the following we go through each of these steps in more details.

The copy step includes the definitions of o, o'_0, σ'_0 and w'_0. The definition o is the object referenced by r. This definition is used to define a copy o'_0. In the next step we have to define an updated state σ'_0 that includes the copy o'_0. There are two cases to be differentiated at this point. If o is handled by q then o'_0 must be handled by p. Otherwise o'_0 must be handled by the handler of o. The definition w'_0 is the updated map.

The attribute values update step recursively uses *deep_import_rec_with_map* to import all the non-void reference attribute values of o using the updated map. This leads to an updated object with the deep imported values. This step includes the definition of $\{a_1,\ldots,a_n\}$, as well as the definitions of $\{r'_1,\ldots,r'_n\}$, $\{w'_1,\ldots,w'_n\}$, $\{\sigma'_1,\ldots,\sigma'_n\}$, and $\{o'_1,\ldots,o'_n\}$. The set $\{a_1,\ldots,a_n\}$ contains each attributes of o whose value is a non-void reference. We define (r'_i,w'_i,σ'_i) for $i = 1\ldots n$ as a sequence of tuples. Each of the tuples is responsible for a single recursive deep import operation for one of the attributes in $\{a_1,\ldots,a_n\}$. Each such operation results in an updated map and an updated state that must be used in the next deep import operation. The result of this is an updated map w'_n, and updated state σ'_n, and references $r_1,\ldots,r_n$ to deep imported data structures. Finally, we define a sequence of updated objects $\{o'_1,\ldots,o'_n\}$ that ends with the updated object o'_n. The updated object has the values of the attributes $\{a_1,\ldots,a_n\}$ set to the deep imported data structures referenced by $r_1,\ldots,r_n$.

Up until now we have an updated state σ'_n that contains the initial copy o'_0. In the client update step we update σ'_n such that the reference to o'_0 points to the updated object o'_n. This is done in the clients update step. This step includes the definition σ'_x. Note that σ'_n is derived from the state σ'_0, which includes the object o'_0.

In a next step we take care of the once routines of the imported object. For this, we define a new state σ'_y based on the state σ'_x. We define $\{f_1,\ldots,f_w\}$ as the set of all non-separate once functions of o that are fresh on the processor $\sigma'_x.handler(\sigma'_x.ref(o'_n))$, which handles the copied object, but non-fresh on the processor $\sigma'_x.handler(r)$, which handles the object referenced by r. Note that the two processor can be the same, in which case the set $\{f_1, \ldots, f_w\}$ is empty. Similarly, we define the set $\{f_{w+1}, \ldots, f_m\}$ for once procedures. For each once routine defined in this way, we update the state σ'_x such that the once status is taken over to the handler of the copied object. These definitions deal with the case where a once routine is fresh on the handler of the copied object, but non-fresh on the handler of the object referenced by r. Note that the remaining cases are implicitly taken care of because no change to the state is necessary. The result is the state σ'_y.

The last step defines the result of the function, based on the definitions of the preceding steps. The result generation step defines the resulting reference r' to the imported object o'_n, the resulting map w', and the resulting state σ'.

$$
\begin{array}{l}
\mathit{deep\_import\_rec\_without\_map}(p, q, r, w, \sigma) = (r', w', \sigma') \\
\quad \textbf{where} \\
\quad o \overset{def}{=} \sigma.\mathit{ref\_obj}(r) \\
\quad o'_0 \overset{def}{=} o.\mathit{copy} \\
\quad \sigma'_0 \overset{def}{=} \begin{cases} \sigma.\mathit{add\_obj}(p, o'_0) & \mathit{if}\, \sigma.\mathit{handler}(r) = q \\ \sigma.\mathit{add\_obj}(\sigma.\mathit{handler}(r), o'_0) & \mathit{otherwise} \end{cases} \\
\quad w'_0 \overset{def}{=} w.\mathit{add}(r, \sigma'_0.\mathit{ref}(o'_0)) \\
\quad \{a_1, \ldots, a_n\} \overset{def}{=} \{a \mid o.\mathit{att\_val}(a) \in \mathbf{REF} \wedge o.\mathit{att\_val}(a) \neq \mathit{void}\} \\
\quad \forall i \in \{1, \ldots, n\}\colon (r'_i, w'_i, \sigma'_i) \overset{def}{=} \mathit{deep\_import\_rec\_with\_map}(p, q, o.\mathit{att\_val}(a_i), w'_{i-1}, \sigma'_{i-1}) \\
\quad \forall i \in \{1, \ldots, n\}\colon o'_i \overset{def}{=} o'_{i-1}.\mathit{set\_att\_val}(a_i, r'_i) \\
\quad \sigma'_x \overset{def}{=} \sigma'_n.\mathit{update\_ref}(\sigma'_n.\mathit{ref}(o'_0), o'_n) \\
\quad \sigma'_y \overset{def}{=} \sigma'_x \\
\qquad .\mathit{set\_once\_func\_not\_fresh}(\sigma'_x.\mathit{handler}(\sigma'_x.\mathit{ref}(o'_n)), f_1, \sigma'_x.\mathit{once\_result}(\sigma'_x.\mathit{handler}(r), f_1)) \\
\qquad . \ldots \\
\qquad .\mathit{set\_once\_func\_not\_fresh}(\sigma'_x.\mathit{handler}(\sigma'_x.\mathit{ref}(o'_n)), f_w, \sigma'_x.\mathit{once\_result}(\sigma'_x.\mathit{handler}(r), f_w)) \\
\qquad .\mathit{set\_once\_proc\_not\_fresh}(\sigma'_x.\mathit{handler}(\sigma'_x.\mathit{ref}(o'_n)), f_{w+1}) \\
\qquad . \ldots \\
\qquad .\mathit{set\_once\_proc\_not\_fresh}(\sigma'_x.\mathit{handler}(\sigma'_x.\mathit{ref}(o'_n)), f_m) \\
\quad\quad \textbf{where} \\
\qquad \{f_1, \ldots, f_w\} \overset{def}{=} \begin{array}{l} \{x \in o.\mathit{class\_type}.\mathit{functions} \mid x.\mathit{is\_once} \wedge \exists c, d\colon \Gamma \vdash x : (d, \bullet, c) \wedge \\ \qquad \sigma'_x.\mathit{is\_fresh}(\sigma'_x.\mathit{handler}(\sigma'_x.\mathit{ref}(o'_n)), x) \wedge \\ \qquad \neg \sigma'_x.\mathit{is\_fresh}(\sigma'_x.\mathit{handler}(r), x)\} \end{array} \\
\qquad \{f_{w+1}, \ldots, f_m\} \overset{def}{=} \begin{array}{l} \{x \in o.\mathit{class\_type}.\mathit{procedures} \mid x.\mathit{is\_once} \wedge \\ \qquad \sigma'_x.\mathit{is\_fresh}(\sigma'_x.\mathit{handler}(\sigma'_x.\mathit{ref}(o'_n)), x) \wedge \\ \qquad \neg \sigma'_x.\mathit{is\_fresh}(\sigma'_x.\mathit{handler}(r), x)\} \end{array} \\
\quad r' \overset{def}{=} \sigma'_y.\mathit{ref}(o'_n) \\
\quad w' \overset{def}{=} w'_n \\
\quad \sigma' \overset{def}{=} \sigma'_y
\end{array}
$$

Setting values of formal arguments and the value of the current object entity The deep import operation is used in two ways. It is used when an expanded object handled by one processor gets used as an actual argument for a formal argument on another processor. The deep import operation also gets used when an expanded object handled by one processor gets returned to another processor. In this section, we focus on the argument passing aspect.

To pass actual arguments, we introduce the auxiliary command *push_env_with_feature*. It defines a state in which a processor p receives a new environment. The new environment is initialized for the execution of the feature f with target reference r_0 and actual argument references $(r_1, \ldots, r_n)$. Actual arguments of expanded type must either be copied or they must be deep imported.

$$\mathit{push\_env\_with\_feature}\colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{FEATURE} \to \mathbf{REF} \to \mathbf{TUPLE} \nrightarrow \mathbf{STATE}$$

$$\begin{array}{l}
\quad \sigma.\mathit{push\_env\_with\_feature}(p, f, r_0, (r_1, \ldots, r_n))\ \mathbf{require} \\
\qquad \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p) \\
\qquad f.\mathit{formals}.\mathit{count} = n \\
\qquad \forall i \in \{0, \ldots, n\}\colon r_i \neq \mathit{void} \to \sigma.\mathit{heap}.\mathit{refs}.\mathit{has}(r_i) \\
\quad \mathbf{axioms} \\
\qquad \sigma.\mathit{push\_env\_with\_feature}(p, f, r_0, (r_1, \ldots, r_n)) = \sigma'_n.\mathit{set}(\sigma'_n.\mathit{regions}, \sigma'_n.\mathit{heap}, \sigma'_n.\mathit{store}.\mathit{push\_env}(p, e)) \\
\qquad\ \mathbf{where} \\
\qquad\quad \sigma'_0 \overset{def}{=} \sigma \\
\qquad\quad \forall i \in \{1, \ldots, n\}\colon
\end{array}$$

$$(\sigma'_i, r'_i) \overset{def}{=} \begin{cases}
\mathit{if}\, \exists d, q, c\colon \Gamma \vdash f.\mathit{formals}(i) : (d, q, c) \wedge c.\mathit{is\_exp} \wedge r_i \neq \mathit{void} \wedge \sigma'_{i-1}.\mathit{handler}(r_i) \neq p \\
\quad (\sigma_x, \sigma_x.\mathit{last\_imported\_ref}) \\
\qquad \mathbf{where} \\
\qquad\quad \sigma_x \overset{def}{=} \sigma'_{i-1}.\mathit{deep\_import}(p, r_i) \\
\mathit{if}\, \exists d, q, c\colon \Gamma \vdash f.\mathit{formals}(i) : (d, q, c) \wedge c.\mathit{is\_exp} \wedge r_i \neq \mathit{void} \wedge \sigma'_{i-1}.\mathit{handler}(r_i) = p \\
\quad (\sigma_x, \sigma_x.\mathit{last\_added\_obj}) \\
\qquad \mathbf{where} \\
\qquad\quad \sigma_x \overset{def}{=} \sigma'_{i-1}.\mathit{add\_obj}(p, \sigma'_{i-1}.\mathit{heap}.\mathit{ref\_obj}(r_i).\mathit{copy}) \\
\mathit{otherwise} \\
\quad (\sigma'_{i-1}, r_i)
\end{cases}$$

$$\begin{array}{l}
w \overset{def}{=} \mathit{new}\, \mathbf{ENV}.\mathit{make} \\
\quad .\mathit{update}(f.\mathit{formals}(1).\mathit{name}, r'_1)\ \ldots\ .\mathit{update}(f.\mathit{formals}(n).\mathit{name}, r'_n) \\
\quad .\mathit{update}(f.\mathit{locals}(1).\mathit{name}, \mathit{void})\ \ldots\ .\mathit{update}(f.\mathit{locals}(f.\mathit{locals}.\mathit{count}).\mathit{name}, \mathit{void}) \\
\quad .\mathit{update}(\mathbf{Current}, r_0) \\
e \overset{def}{=} \begin{cases} w & \mathit{if}\, f \in \mathbf{PROCEDURE} \\ w.\mathit{update}(\mathbf{Result}, \mathit{void}) & \mathit{if}\, f \in \mathbf{FUNCTION} \end{cases}
\end{array}$$

In a first step, the auxiliary command defines an updated state, in which p gets a new initialized environment e. The updated state is based on an intermediate state σ'_n, which gets defined in a cascade of state updates with the goal of either copying or deep importing the actual arguments of expanded type. The cascade starts with the definition of a starting state σ'_0. For each formal argument, the cascade defines a tuple (σ'_i, r'_i) with an updated state and a reference. If the corresponding actual argument is of reference class type, nothing needs to be done. If the actual argument is of expanded class type and the referenced object is not handled by p then p must deep import the object structure. This results in an updated state and a new reference to the deep imported object structure. If the actual argument is of expanded class type and the referenced object is handled by p then the expanded object must be copied. This results in an updated state and a new reference to the copy. The resulting state σ'_n contains all the deep imported and copied objects. The resulting references $r'_1, \ldots, r'_n$ will be used for values of the formal argument names.

In a next step, the command defines the environment w as a new environment that gets updated to map formal argument names, local variable names, the current entity name, and the result entity name to the respective values. The names of the formal arguments get mapped to the references $r'_1, \ldots, r'_n$. Names of local variables are mapped to the void reference. The current entity name is mapped to the target reference.

The environment w is used to define the final environment e in which the result name gets mapped to the void reference. This environment and the updated state σ'_n are used to define the result of the command. The auxiliary command *push_env* is used to push e onto p's stack of environments. The auxiliary command *push_env* takes a processor p and an environment e. It returns a state in which e is pushed on top of p's environment stack.

$\mathit{push\_env}\colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{ENV} \to \mathbf{STATE}$

$\quad \sigma.\mathit{push\_env}(p, e)$ **require**

$\quad\quad \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p)$

$\quad$ **axioms**

$\quad\quad \sigma.\mathit{push\_env}(p, e) = \sigma.\mathit{set}(\sigma.\mathit{regions}, \sigma.\mathit{heap}, \sigma.\mathit{store}.\mathit{push\_env}(p, e))$

The effect of a call to *push_env_with_feature* or a call to *push_env* can be undone with a call to the auxiliary command *pop_env*. This auxiliary command takes a processor p and removes the top environment from p's stack of environments.

$\mathit{pop\_env}\colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{STATE}$

$\quad \sigma.\mathit{pop\_env}(p)$ **require**

$\quad\quad \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p)$

$\quad\quad \neg\sigma.\mathit{store}.\mathit{envs}(p).\mathit{is\_empty}$

$\quad$ **axioms**

$\quad\quad \sigma.\mathit{pop\_env}(p) = \sigma.\mathit{set}(\sigma.\mathit{regions}, \sigma.\mathit{heap}, \sigma.\mathit{store}.\mathit{pop\_env}(p))$

Setting values of local variables and the value of the result entity The values of local variables and the value of the result entity are maintained in the store. Note that formal arguments and the current entity are non-writable. The values of these entities can only be set at the beginning during the definition of the environment. We introduce the auxiliary command *set_env_val* to set a value v for the name n in processor p's top environment. For this, we define an updated environment e in which n is set to v. We then define an updated store s by first removing the top environment and then adding the updated environment e. The updated store is then used to define an updated state. The updated state becomes the result of the auxiliary command.

$\mathit{set\_env\_val}\colon \mathbf{STATE} \rightarrow \mathbf{PROC} \nrightarrow \mathbf{NAME} \nrightarrow \mathbf{REF} \cup \mathbf{PROC} \nrightarrow \mathbf{STATE}$

$\sigma.\mathit{set\_env\_val}(p, n, v)$ **require**

$\sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p)$

$\neg\sigma.\mathit{store}.\mathit{envs}(p).\mathit{is\_empty}$

$v \in \mathbf{REF} \wedge v \neq \mathit{void} \rightarrow \sigma.\mathit{heap}.\mathit{refs}.\mathit{has}(v)$

$v \in \mathbf{PROC} \rightarrow \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(v)$

axioms

$\sigma.\mathit{set\_env\_val}(p, n, v) = \sigma.\mathit{set}(k, h, s)$

where

$e \overset{\mathit{def}}{=} \sigma.\mathit{store}.\mathit{envs}(p).\mathit{top}.\mathit{update}(n, v)$

$k \overset{\mathit{def}}{=} \sigma.\mathit{regions}$

$h \overset{\mathit{def}}{=} \sigma.\mathit{heap}$

$s \overset{\mathit{def}}{=} \sigma.\mathit{store}.\mathit{pop\_env}(p).\mathit{push\_env}(p, e)$

Setting attribute values of the current object The auxiliary command *set_att_val* takes an object o, a name n, and a value v. It returns an updated state in which the attribute with name n of object o is set to the value v. In a first step, the auxiliary command defines an updated object with a call to *set_att_val*. This updated object is then used to update the existing reference to o in the state.

$\mathit{set\_att\_val}\colon \mathbf{STATE} \rightarrow \mathbf{OBJ} \nrightarrow \mathbf{NAME} \nrightarrow \mathbf{REF} \cup \mathbf{PROC} \nrightarrow \mathbf{STATE}$

$\sigma.\mathit{set\_att\_val}(o, n, v)$ **require**

$\sigma.\mathit{heap}.\mathit{objs}.\mathit{has}(o)$

$\exists a \in o.\mathit{class\_type}.\mathit{attributes}\colon a.\mathit{name} = n$

$v \in \mathbf{REF} \wedge v \neq \mathit{void} \rightarrow \sigma.\mathit{heap}.\mathit{refs}.\mathit{has}(v)$

$v \in \mathbf{PROC} \rightarrow \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(v)$

axioms

$\sigma.\mathit{set\_att\_val}(o, n, v) = \sigma.\mathit{update\_ref}(\sigma.\mathit{heap}.\mathit{ref}(o), o.\mathit{set\_att\_val}(a, v))$

where

$a \overset{\mathit{def}}{=} o.\mathit{class\_type}.\mathit{feature\_by\_name}(n)$

Setting values of local variables, the value of the result entity, and attribute values of the current object in a unified way The auxiliary command *set_val* is used to attach a value v to an entity with name n. The entity can either be a local variable or the result entity in the top environment of p. It can also be an attribute of the current object on p. In either case, the update affects an entity on p.

The definition of the resulting state is based on the auxiliary definitions o, σ', and v'. The definition o defines the current object, as defined by the top environment of processor p. The precondition makes sure that there is always such an environment on p where the current object is defined. If v is a reference and the referenced object is an object of reference class type then v can be attached directly to the entity with name n. If the object is an expanded object handled by processor p then the referenced object must first be copied. Expanded objects handled by a processor different than p must be deep imported. However, this is done right when the object gets returned from another processor to p. The definitions σ' and v' define a state and a value that are potentially updated according to these rules.

The state σ' must be updated with the value v'. The update can either affect the current object on p or it can affect the top environment of p. Attribute names of the current object, local variable names, and formal argument names are distinct. Therefore it is safe to first check whether the current object o has an attribute with name n, in which case the current object gets updated with a v'. If the current object does not have such an attribute, then it is safe to assume that the top environment contains an entity with name n, in which case the top environment gets updated.

set_val: **STATE** $\to$ **PROC** $\nrightarrow$ **NAME** $\nrightarrow$ **REF** $\cup$ **PROC** $\nrightarrow$ **STATE**

$\sigma.set\_val(p, n, v)$ **require**

$\sigma.regions.procs.has(p)$

$\neg\sigma.store.envs(p).is\_empty \wedge \sigma.store.envs(p).top.names.has(\textbf{Current})$

$v \in \textbf{REF} \wedge v \neq void \to \sigma.heap.refs.has(v)$

$v \in \textbf{PROC} \to \sigma.regions.procs.has(v)$

axioms

$$\sigma.set\_val(p, n, v) = \begin{cases} if\,\exists a \in o.class\_type.attributes\colon a.name = n \\ \quad \sigma'.set\_att\_val(o, n, v') \\ otherwise \\ \quad \sigma'.set\_env\_val(p, n, v') \end{cases}$$

where

$$o \overset{def}{=} \sigma.heap.ref\_obj(\sigma.store.envs(p).top.val(\textbf{Current}))$$

$$(\sigma', v') \overset{def}{=} \begin{cases} if\,v \in \textbf{REF} \wedge v \neq void \wedge \sigma.heap.ref\_obj(v).class\_type.is\_exp \wedge \sigma.handler(v) = p \\ \quad (\sigma_x, \sigma_x.last\_added\_obj) \\ \quad\ \textbf{where} \\ \quad\quad \sigma_x \overset{def}{=} \sigma.add\_obj(p, \sigma.heap.ref\_obj(v).copy) \\ otherwise \\ \quad (\sigma, v) \end{cases}$$

Setting values of once functions Values can also be stored in the status of once functions. A once function can be fresh or non-fresh. If the once function is non-fresh on a processor p then there is a once result for the once function on p. A once function is set as non-fresh during the execution of the once function. In the following, we take a look at how a processor can set the status of once routines in general, i.e. we consider both once functions and once procedures.

The auxiliary command *set_once_func_not_fresh* takes a processor p, a once function f, and a value r. It returns an updated state in which f is set as non-fresh with the once result r. If f is declared as non-separate, then f is set as non-fresh on p with the once result r. If f is declared as separate with or without an explicit processor specification then f is set as non-fresh on all processors.

$set\_once\_func\_not\_fresh$: **STATE** $\to$ **PROC** $\nrightarrow$ **FEATURE** $\nrightarrow$ **REF** $\nrightarrow$ **STATE**

$\sigma.set\_once\_func\_not\_fresh(p, f, r)$ **require**

$\sigma.regions.procs.has(p)$

$f \in \textbf{FUNCTION} \wedge f.is\_once$

$r \neq void \to \sigma.heap.refs.has(r)$

axioms

$$\sigma.set\_once\_func\_not\_fresh(p, f, r) = \sigma.set(\sigma.regions, \sigma.heap.set\_once\_func\_not\_fresh(p, f, r), \sigma.store)$$

The auxiliary command *set_once_proc_not_fresh* does the same for once procedures. It takes a processor p and a once procedure f and it returns a state in which f is set as non-fresh on p.

$set\_once\_proc\_not\_fresh \colon \mathbf{STATE} \to \mathbf{PROC} \to \mathbf{FEATURE} \to \mathbf{STATE}$

$\quad \sigma.set\_once\_proc\_not\_fresh(p, f)$ **require**

$\quad\quad \sigma.regions.procs.has(p)$

$\quad\quad f \in \mathbf{PROCEDURE} \wedge f.is\_once$

$\quad$ **axioms**

$\quad\quad \sigma.set\_once\_proc\_not\_fresh(p, f) = \sigma.set(\sigma.regions, \sigma.heap.set\_once\_proc\_not\_fresh(p, f), \sigma.store)$

4.6.5. Getting values

In this section, we take a look at how a processor can read the a value that got written with one of the mechanisms from Section 4.6.4.

Getting values of local variables, the value of the current object entity, and the value of the result entity The auxiliary query *envs* takes a processor p and returns the stack of environments for p. The auxiliary query *env_val* is more specialized. It takes a processor p and a name n and it returns the value stored under n in the top environment of p.

$envs \colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{STACK}[\mathbf{ENV}]$

$\quad \sigma.envs(p)$ **require**

$\quad\quad \sigma.regions.procs.has(p)$

$\quad$ **axioms**

$\quad\quad \sigma.envs(p) = \sigma.store.envs(p)$

$env\_val \colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{NAME} \nrightarrow \mathbf{REF} \cup \mathbf{PROC}$

$\quad \sigma.env\_val(p, n)$ **require**

$\quad\quad \sigma.regions.procs.has(p)$

$\quad\quad \neg\sigma.store.envs(p).is\_empty \wedge \sigma.store.envs(p).top.names.has(n)$

$\quad$ **axioms**

$\quad\quad \sigma.env\_val(p, n) = \sigma.store.envs(p).top.val(n)$

Getting attribute values of the current object The auxiliary query *att_val* takes an object o and a name n and returns the attribute value for the attribute with name n of object o.

$att\_val \colon \mathbf{STATE} \to \mathbf{OBJ} \nrightarrow \mathbf{NAME} \nrightarrow \mathbf{REF} \cup \mathbf{PROC}$

$\quad \sigma.att\_val(o, n)$ **require**

$\quad\quad \sigma.heap.objs.has(o)$

$\quad\quad \exists a \in o.class\_type.attributes \colon a.name = n$

$\quad$ **axioms**

$\quad\quad \sigma.att\_val(o, n) = o.att\_val(a)$

$\quad\quad\quad$ **where**

$\quad\quad\quad a \overset{def}{=} o.class\_type.feature\_by\_name(n)$

Getting values of local variables, the value of the current object entity, the value of the result entity, and attribute values of the current object in a unified way We use the existing auxiliary queries *env_val* and *att_val* to define a new auxiliary query *val* that deals both with values in the top environment as well as with values stored in attribute values of the current object. The auxiliary query *val* takes a processor p and a name n and it returns the value of n in p's current feature execution context. This context consists of the top environment and its reference to the current object. The auxiliary query requires that the execution context of processor p is setup properly, i.e. there is a top environment with a reference to the current object. The precondition also states that either the top environment has the name n registered or the current object has an attribute with name n. In any valid SCOOP program, any environment variables has a name that is distinct from the attribute names of the current object. This allows us to define the result of the auxiliary query in a simple way. If the name exists in the top environment then the result is the value given by *env_val*. Otherwise the name must be the name of an attribute of the current object, in which case the result is given by *att_val*.

$val\colon \textbf{STATE} \to \textbf{PROC} \nrightarrow \textbf{NAME} \nrightarrow \textbf{REF} \cup \textbf{PROC}$

$\sigma.val(p, n)$ **require**

$\sigma.regions.procs.has(p)$

$\neg\sigma.store.envs(p).is\_empty$

$e.names.has(\textbf{Current})$

$e.names.has(n) \vee \exists a \in o.class\_type.attributes\colon a.name = n$

where

$e \overset{def}{=} \sigma.store.envs(p).top$

$o \overset{def}{=} \sigma.heap.ref\_obj(e.val(\textbf{Current}))$

axioms

$$\sigma.val(p,n) = \begin{cases} if\, e.names.has(n) \\ \quad \sigma.env\_val(p,n) \\ \;\textbf{where} \\ \quad e \overset{def}{=} \sigma.store.envs(p).top \\ if\, \exists a \in o.class\_type.attributes\colon a.name = n \\ \quad \sigma.att\_val(o,n) \\ \;\textbf{where} \\ \quad e \overset{def}{=} \sigma.store.envs(p).top \\ \quad o \overset{def}{=} \sigma.heap.ref\_obj(e.val(\textbf{Current})) \end{cases}$$

Getting values of once functions In the following we take a look at the auxiliary queries to access the status of once routines. We describe once routines in general, i.e. we also describe once procedures.

The auxiliary query *is_fresh* takes a processor p and a once routine f. It returns whether f is fresh on p or not.

$is\_fresh\colon \textbf{STATE} \to \textbf{PROC} \nrightarrow \textbf{FEATURE} \nrightarrow \textbf{BOOLEAN}$

$\sigma.is\_fresh(p, f)$ **require**

$\sigma.regions.procs.has(p)$

$f.is\_once$

axioms

$\sigma.is\_fresh(p, f) = \sigma.heap.is\_fresh(p, f)$

For non-fresh once functions, the auxiliary query *once_result* returns the once result of f on p.

$\mathit{once\_result}\colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{FEATURE} \nrightarrow \mathbf{REF}$

$\quad \sigma.\mathit{once\_result}(p, f)$ **require**

$\qquad \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p)$

$\qquad f \in \mathbf{FUNCTION} \wedge f.\mathit{is\_once}$

$\qquad \neg\sigma.\mathit{heap}.\mathit{is\_fresh}(p, f)$

$\quad$ **axioms**

$\qquad \sigma.\mathit{once\_result}(p, f) = \sigma.\mathit{heap}.\mathit{once\_result}(p, f)$

4.6.6. Locking

In this section we explore the aspect of the facade that deals with locking. The auxiliary query *is_rq_locked* returns whether a processor p's request queue is locked or not. We do not provide auxiliary queries to distinguish between obtained and retrieved locks. Instead, we define the auxiliary queries *rq_locks* and *cs_locks* that return the set of all request queue locks, respectively the set of all call stack locks of a processor p. These locks are only usable if they are not passed. This information can be retrieved with a call to the auxiliary query *are_locks_passed*.

$\mathit{is\_rq\_locked}\colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{BOOLEAN}$

$\quad \sigma.\mathit{is\_rq\_locked}(p)$ **require**

$\qquad \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p)$

$\quad$ **axioms**

$\qquad \sigma.\mathit{is\_rq\_locked}(p) = \sigma.\mathit{regions}.\mathit{is\_rq\_locked}(p)$

$\mathit{rq\_locks}\colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{SET}[\mathbf{PROC}]$

$\quad \sigma.\mathit{rq\_locks}(p)$ **require**

$\qquad \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p)$

$\quad$ **axioms**

$\qquad \sigma.\mathit{rq\_locks}(p) = \sigma.\mathit{regions}.\mathit{rq\_locks}(p)$

$\mathit{cs\_locks}\colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{SET}[\mathbf{PROC}]$

$\quad \sigma.\mathit{cs\_locks}(p)$ **require**

$\qquad \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p)$

$\quad$ **axioms**

$\qquad \sigma.\mathit{cs\_locks}(p) = \sigma.\mathit{regions}.\mathit{cs\_locks}(p)$

$\mathit{are\_locks\_passed}\colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{BOOLEAN}$

$\quad \sigma.\mathit{are\_locks\_passed}(p)$ **require**

$\qquad \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p)$

$\quad$ **axioms**

$\qquad \sigma.\mathit{are\_locks\_passed}(p) = \sigma.\mathit{regions}.\mathit{are\_locks\_passed}(p)$

The facade provides auxiliary commands for locking request queues, removing obtained request queue locks, unlocking request queues, delegating obtained request queue locks, passing locks, and revoking locks.

$\mathit{lock\_rqs}\colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{SET}[\mathbf{PROC}] \nrightarrow \mathbf{STATE}$

$\sigma.\mathit{lock\_rqs}(p, \bar{l})$ **require**

$\sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p)$

$\forall x \in \bar{l}\colon \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(x)$

$\forall x \in \bar{l}\colon \sigma.\mathit{regions}.\mathit{is\_rq\_locked}(x) = \mathit{false}$

axioms

$\sigma.\mathit{lock\_rqs}(p, \bar{l}) = \sigma.\mathit{set}(\sigma.\mathit{regions}.\mathit{lock\_rqs}(p, \bar{l}), \sigma.\mathit{heap}, \sigma.\mathit{store})$

$\mathit{pop\_obtained\_rq\_locks}\colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{STATE}$

$\sigma.\mathit{pop\_obtained\_rq\_locks}(p)$ **require**

$\sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p)$

$\neg\sigma.\mathit{regions}.\mathit{obtained\_rq\_locks}(p).\mathit{is\_empty}$

$\sigma.\mathit{regions}.\mathit{are\_locks\_passed}(p) = \mathit{false}$

axioms

$\sigma.\mathit{pop\_obtained\_rq\_locks}(p) = \sigma.\mathit{set}(\sigma.\mathit{regions}.\mathit{pop\_obtained\_rq\_locks}(p), \sigma.\mathit{heap}, \sigma.\mathit{store})$

$\mathit{unlock\_rq}\colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{STATE}$

$\sigma.\mathit{unlock\_rq}(p)$ **require**

$\sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p)$

$\sigma.\mathit{regions}.\mathit{is\_rq\_locked}(p) = \mathit{true}$

$\forall q \in \sigma.\mathit{regions}.\mathit{procs}\colon \neg\sigma.\mathit{regions}.\mathit{obtained\_rq\_locks}(q).\mathit{flat}.\mathit{has}(p)$

axioms

$\sigma.\mathit{unlock\_rq}(p) = \sigma.\mathit{set}(\sigma.\mathit{regions}.\mathit{unlock\_rq}(p), \sigma.\mathit{heap}, \sigma.\mathit{store})$

$\mathit{delegate\_obtained\_rq\_locks}\colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{SET}[\mathbf{PROC}] \nrightarrow \mathbf{STATE}$

$\sigma.\mathit{delegate\_obtained\_rq\_locks}(p, \bar{l})$ **require**

$\sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p)$

$\forall x \in \bar{l}\colon \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(x)$

$\forall x \in \bar{l}\colon \neg\exists y \in \sigma.\mathit{regions}.\mathit{procs}\colon \sigma.\mathit{regions}.\mathit{obtained\_rq\_locks}(y).\mathit{flat}.\mathit{has}(x)$

$\forall x \in \bar{l}\colon \sigma.\mathit{regions}.\mathit{is\_rq\_locked}(x) = \mathit{true}$

axioms

$\sigma.\mathit{delegate\_obtained\_rq\_locks}(p, \bar{l}) = \sigma.\mathit{set}(\sigma.\mathit{regions}.\mathit{delegate\_obtained\_rq\_locks}(p, \bar{l}), \sigma.\mathit{heap}, \sigma.\mathit{store})$

$\mathit{pass\_locks}\colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{PROC} \nrightarrow \mathbf{TUPLE}[\mathbf{SET}[\mathbf{PROC}], \mathbf{SET}[\mathbf{PROC}]] \nrightarrow \mathbf{STATE}$

$\quad \sigma.\mathit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c}))$ **require**

$\quad\quad \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p) \wedge \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(q)$

$\quad\quad \forall x \in \overline{l_r}\colon \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(x) \wedge \forall x \in \overline{l_c}\colon \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(x)$

$\quad\quad \forall x \in \overline{l_r}\colon \sigma.\mathit{regions}.\mathit{obtained\_rq\_locks}(p).\mathit{flat}.\mathit{has}(x) \vee \sigma.\mathit{regions}.\mathit{retrieved\_rq\_locks}(p).\mathit{flat}.\mathit{has}(x)$

$\quad\quad \forall x \in \overline{l_c}\colon x = \sigma.\mathit{regions}.\mathit{obtained\_cs\_lock}(p) \vee \sigma.\mathit{regions}.\mathit{retrieved\_cs\_locks}(p).\mathit{flat}.\mathit{has}(x)$

$\quad\quad \sigma.\mathit{regions}.\mathit{are\_locks\_passed}(p) = \mathit{false}$

$\quad$ **axioms**

$\quad\quad \sigma.\mathit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c})) = \sigma.\mathit{set}(\sigma.\mathit{regions}.\mathit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c})), \sigma.\mathit{heap}, \sigma.\mathit{store})$

$\mathit{revoke\_locks}\colon \mathbf{STATE} \to \mathbf{PROC} \nrightarrow \mathbf{PROC} \nrightarrow \mathbf{STATE}$

$\quad \sigma.\mathit{revoke\_locks}(p, q)$ **require**

$\quad\quad \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(p) \wedge \sigma.\mathit{regions}.\mathit{procs}.\mathit{has}(q)$

$\quad\quad \neg\sigma.\mathit{regions}.\mathit{retrieved\_rq\_locks}(q).\mathit{is\_empty} \wedge \neg\sigma.\mathit{regions}.\mathit{retrieved\_cs\_locks}(q).\mathit{is\_empty}$

$\quad\quad \sigma.\mathit{regions}.\mathit{retrieved\_rq\_locks}(q).\mathit{top} \subseteq$

$\quad\quad\quad \sigma.\mathit{regions}.\mathit{obtained\_rq\_locks}(p).\mathit{flat} \cup \sigma.\mathit{regions}.\mathit{retrieved\_rq\_locks}(p).\mathit{flat}$

$\quad\quad \sigma.\mathit{regions}.\mathit{retrieved\_cs\_locks}(q).\mathit{top} \subseteq$

$\quad\quad\quad \{\sigma.\mathit{regions}.\mathit{obtained\_cs\_lock}(p)\} \cup \sigma.\mathit{regions}.\mathit{retrieved\_cs\_locks}(p).\mathit{flat}$

$\quad\quad \sigma.\mathit{regions}.\mathit{retrieved\_rq\_locks}(q).\mathit{top} \cup \sigma.\mathit{regions}.\mathit{retrieved\_cs\_locks}(q).\mathit{top} \neq \{\} \to$

$\quad\quad\quad \sigma.\mathit{regions}.\mathit{are\_locks\_passed}(p) = \mathit{true}$

$\quad\quad \sigma.\mathit{regions}.\mathit{are\_locks\_passed}(q) = \mathit{false}$

$\quad$ **axioms**

$\quad\quad \sigma.\mathit{revoke\_locks}(p, q) = \sigma.\mathit{set}(\sigma.\mathit{regions}.\mathit{revoke\_locks}(p, q), \sigma.\mathit{heap}, \sigma.\mathit{store})$

4.7. Simplified state description

In previous sections we formalized the state as an instance of an ADT. Each such instance is uniquely described by the values of all its queries. However, a description produced in this manner is not practical because it is too verbose. In this section, we develop a *simplified state description* that provides an abstract view on the queries of the state. We divide the description into four parts: the locks, the objects, the once status, and the environments. We identify each part with a label.

In the locks part, we mention for each processor the stack of obtained request queue locks, the stack of retrieved request queue locks and the stack of retrieved call stack locks. We do not mention the obtained call stack lock because it is constant. We use two indicators to say that a processor's request queue is locked or unlocked. We also use an indicator to state when a processor passed its locks. The absence of this indicator implies that the locks are not passed.

The objects part shows for each processor the handled objects with their references. We also show the content of the objects. In particular, for objects other than arrays or objects of basic class type we show a list of attribute values. Attributes that are void are omitted. For objects of basic type, we show the basic value. For arrays, we show the cells of the array.

The part on once status shows the status for each non-fresh routine. For once functions we show the value of the once function. For once procedures we simply indicate that they are non-fresh. Once routines can be non-fresh either with respect to a subset of processors or with respect to all processors. In the first case we mention the once routine for each processor in the subset. In the second case we use an indicator to indicate all processors.

The store is represented in the environments part. This part shows the environments for each processor. Each environment maps names to values.

Example 4 (Simplified state description) We assume a system with three processors p_1, p_2, and p_3. The following simplified state description shows a state in which all request queues are locked. Processor p_1 has a stack of obtained request queue locks with two items. On the bottom of the stack there is the set $\{p_2\}$ and on the top there is the set $\{p_3\}$. Processor p_1's stack of retrieved request queue locks and retrieved call stack locks each consists of two empty sets. Processor p_1 passed its locks. From p_3's entry, we can see that these locks have been passed from p_1 to p_3. Processor p_2 does not have any locks, except its own call stack lock.

In the objects part, we can see that processor p_1 handles an object o_1 that is referenced by r_1. Processors p_2 and p_3 each handle multiple objects. Object o_5 is an object with an attribute *id* that references object o_6 through the reference r_6. Objects o_6 and o_4 are objects with the integer values 2 and 1. Object o_2 is a two dimensional array with 2×2 cells. Each of the cells references the object o_3 through the reference r_3.

The once status part shows two items for the once function *id* and the once procedure *initialize* of class *APPLICATION*. The once function *id* is non-fresh with the value r_4 on processor p_2. The once procedure *initialize* is non-fresh on all processors in the system.

In the last part we have the environments for our processors. Processor p_1 has a stack with two environments. The environment on the left is at the bottom of the stack and the environment on the right is at the top of the stack. Processor p_2 has no environments and processor p_3 has one environment. In p_3's environment we have three mappings. The entity *a_root* has the value r_1, the current entity has the value r_5, and the result entity has the void value.

locks:

$p_1 ::$ orq: $(\{p_2\}, \{p_3\})$ rrq: $(\{\}, \{\})$ rcs: $(\{\}, \{\})$ locked passed

$p_2 ::$ orq: $()$ rrq: $()$ rcs: $()$ locked

$p_3 ::$ orq: $()$ rrq: $(\{p_2, p_3\})$ rcs: $(\{p_1\})$ locked

objects:

$p_1 :: r_1 \rightarrow o_1$

$p_2 :: r_2 \rightarrow o_2[[r_3, r_3], [r_3, r_3]], r_3 \rightarrow o_3, r_4 \rightarrow o_4(1)$

$p_3 :: r_5 \rightarrow o_5(id \rightarrow r_6), r_6 \rightarrow o_6(2)$

once status:

$p_2 :: \{APPLICATION\}.id \rightarrow r_4$

all $:: \{APPLICATION\}.initialize$

environments:

$p_1 :: a\_node \rightarrow r_2, \textbf{Current} \rightarrow r_1 \ / \ a\_node \rightarrow r_5, \textbf{Current} \rightarrow r_1$

$p_2 ::$

$p_3 :: a\_root \rightarrow r_1, \textbf{Current} \rightarrow r_5, \textbf{Result} \rightarrow void$

□

5. Formalization of execution

In this section we formalize the execution of a SCOOP program. We start with an introduction to our approach. We then define the starting point and explain the rules. The rules are divided into rules for mechanisms and rules for code elements.

5.1. General approach

We formalize the execution of a SCOOP program with a *structural operational semantics* as described by Plotkin [27]. The idea behind a structural operational semantics is to define the behavior of a program in terms of its parts, i.e. the syntactical elements of the program. Such a semantics is intuitive because it talks directly about elements in the code. It is a very powerful semantics because it allows us to apply structural induction as a proof technique. We reuse parts of the terminology from Ostroff et al. [25]. They provide a limited structural operational semantics for SCOOP.

5.1.1. Computations

A *computation* models the execution of a SCOOP program. We define a computation as a sequence of configurations, where each non-initial configuration is derived from a previous configuration through a transition. Each configuration defines a state and a list of statements for each processor. Each transition is described by an inference rule that maps one configuration to another. The transition from one configuration to the next models an atomic step of one processor. The concurrent execution of a SCOOP program is modeled by the interleaved transitions taken by different processors.

Example 5 (Modeling of parallel execution) Suppose we have two processors p and q. Processor p executes the following sequence of statements: $s_{p,1}; s_{p,2}$. In parallel, processor q executes the following sequence of statements: $s_{q,1}$. We model this execution with any of the following computations: $s_{p,1}; s_{p,2}; s_{q,1}$ or $s_{p,1}; s_{q,1}; s_{p,2}$ or $s_{q,1}; s_{p,1}; s_{p,2}$. $\square$

5.1.2. Configurations

A *configuration* models a snapshot in the execution of a SCOOP program. Each configuration consists of a state and a number of processors, each with a queue of statements. The state is given as an instance of the state ADT. For the processors and their statements, we introduce the notion of a schedule. A *schedule* is a set of processors with associated action queues where each *action queue* is a queue of statements. Each processor must execute the statements in its action queue in a FIFO order. The beginning of the action queue contains the statements for the features that are being executed at the moment. The order of these statements models the way the call stack orders feature executions. The tail of the action queue is the request queue of the processor. A call stack lock is the right to add a feature request to the beginning of the action queue and a request queue lock is the right to add a feature request to the end of the action queue. We write a configuration with processors $p_1, \dots, p_n$, respective action queues $s_1, \dots, s_n$, and state σ as:

$$\langle p_1 :: s_1 \mid \dots \mid p_n :: s_n, \sigma \rangle$$

The configuration is *well-defined* if and only if $\neg\exists i, j \in \{1, \dots, n\} \colon p_i = p_j$. Note that we use $\mid$ to separate processors. This operator is commutative and associative, i.e. $p_1 :: s_1 \mid p_2 :: s_2 = p_2 :: s_2 \mid p_1 :: s_1$ and $p_1 :: s_1 \mid (p_2 :: s_2 \mid p_3 :: s_3) = (p_1 :: s_1 \mid p_2 :: s_2) \mid p_3 :: s_3$. Within an action queue we use ; to separate statements.

5.1.3. Statements

A *statement* is an element of the action queue. A statement is either an instruction or an operation. An *instruction* is user syntax, i.e an action that occurs explicitly in the SCOOP program. The instructions that we take into consideration are shown in Section 3. An *operation* is run-time syntax, i.e an action that does not explicitly occur in a SCOOP program. For example, locking of request queues is not an action that is explicit in a SCOOP program. Instead, locking is based on the formal argument list. It is done implicitly before a feature gets executed. We defined a number of operations later on.

5.1.4. Transitions

A *transition* takes a system in a start configuration and leaves it in a result configuration. The following shows the general form of a transition definition that declares a start configuration $\langle P, \sigma \rangle$ with schedule $P \overset{def}{=} p_1 :: s_1 \mid \dots \mid p_n :: s_n$ and a result configuration $\langle P', \sigma' \rangle$ with schedule $P' \overset{def}{=} p'_1 :: s'_1 \mid \dots \mid p'_m :: s'_m$:

$$\Gamma \vdash \langle P, \sigma \rangle \rightarrow \langle P', \sigma' \rangle$$

The typing environment Γ can be used in the transition definition to access static information about the SCOOP program.

5.1.5. *Inference rules*

An *inference rule* describes the circumstances under which a transition can be used. The inference rule has a premise and a conclusion, divided by a horizontal bar. The *conclusion* is the transition and the *premise* describes the circumstances under which the transition can be used. The premise consists of a number of transitions and a side condition. The premise is satisfied if all transitions in the premise can be taken and if the side condition is true. The following shows a template for inference rules:

General Inference Rule Template

$$\frac{\begin{array}{l}\textit{side condition}\\ \Gamma \vdash \langle P_1, \sigma_{P_1}\rangle \to \langle P_1', \sigma_{P_1}'\rangle\\ \ldots\\ \Gamma \vdash \langle P_n, \sigma_{P_n}\rangle \to \langle P_n', \sigma_{P_n}'\rangle\end{array}}{\Gamma \vdash \langle P_{n+1}, \sigma_{P_{n+1}}\rangle \to \langle P_{n+1}', \sigma_{P_{n+1}}'\rangle}$$

In our formalization, most of the rules have no transition in the premise. Only the structural inference rules make use of this possibility. All other rules only have a side condition in their premise. For these cases we define the following *simplified inference rule template*:

Simplified Inference Rule Template

$$\frac{\begin{array}{l}\textit{condition}\\ \textit{new state}\,\sigma'\ \textit{definition}\\ \textit{fresh channels definitions}\end{array}}{\Gamma \vdash \langle P, \sigma\rangle \to \langle P', \sigma'\rangle}$$

We divide the side condition into three parts. The first part defines a *condition* that is based on the typing environment and the start configuration. The second part is the *new state definition* that defines the state of the result configuration. This new state is based on the state in the start configuration. The last part are the *fresh channels definitions*. Later on we discuss how we define fresh channels and how they are used. Auxiliary definitions can be used in the condition, the new state definition, and the fresh channels definitions. The side condition can mention features of the state ADT. The preconditions of these features serve as additional conditions in the side condition.

5.1.6. *Structural inference rules*

In this section we introduce inference rules that change the structure of configurations. We introduce an inference rule that generalizes a transition by adding processors both to the start configuration and to the result configuration. These additional processors run in parallel but do not take any actions during the generalized transition.

Parallelism

$$\frac{\Gamma \vdash \langle P, \sigma\rangle \to \langle P', \sigma'\rangle}{\Gamma \vdash \langle P \mid Q, \sigma\rangle \to \langle P' \mid Q, \sigma'\rangle}$$

5.1.7. *Scheduler*

Before a processor can execute a feature it must acquire locks and it must wait until the wait condition is satisfied. A locking request encapsulates these two requirements: It consist of the requested locks and the wait condition. At every moment, multiple processors can have conflicting locking requests. The scheduler is the arbiter for these conflicts. The scheduler takes locking requests and stores them in a queue. It then grants requests according to a certain scheduling algorithm.

The model permits a number of possible scheduling algorithms. The algorithms differ in their level of fairness and their performance. In this work, we do not focus on a particular scheduling algorithm. Instead we use the conditions of the inference rules to express locking requests. If more than one processor satisfies the conditions then any of these processors can proceed.

5.2. Initial configuration

The initial configuration is defined by the SCOOP program. Every SCOOP program defines a root class type c and a root procedure f. The root procedure is a creation procedure of the root class type that has no formal arguments and no precondition.

In the beginning, the runtime generates a bootstrap processor p and root processor q with a root object of the root class type. The request queue of the root processor is locked on behalf of the bootstrap processor. This defines our initial state σ:

$$\sigma_x \overset{def}{=} \mathit{new}\,\mathbf{STATE}.\mathit{make}$$

$$\sigma_y \overset{def}{=} \sigma_x.\mathit{add\_proc}(\sigma_x.\mathit{new\_proc})$$

$$p \overset{def}{=} \sigma_y.\mathit{last\_added\_proc}$$

$$\sigma_z \overset{def}{=} \sigma_y.\mathit{add\_proc}(\sigma_y.\mathit{new\_proc})$$

$$q \overset{def}{=} \sigma_z.\mathit{last\_added\_proc}$$

$$\sigma_w \overset{def}{=} \sigma_z.\mathit{add\_obj}(q, \sigma_z.\mathit{new\_obj}(c))$$

$$r \overset{def}{=} \sigma_w.\mathit{ref}(\sigma_w.\mathit{last\_added\_obj})$$

$$\sigma \overset{def}{=} \sigma_w.\mathit{lock\_rqs}(p, \{q\})$$

The bootstrap processor first asks the root processor to execute the root procedure on the root object and then asks the root processor to unlock its request queue as soon as it finished the execution. The bootstrap processor can do this because it has the request queue lock on the root processor. Finally, the bootstrap processor removes the request queue lock from its stack of obtained request queue locks. This is shown in the following initial configuration:

```
⟨
  p :: call(r, f, (), ());
       issue(q, unlock);
       pop_obtained_rq_locks |
   q ::
,
  σ
⟩
```

The statements `call`, `issue`, `unlock`, and `pop_obtained_rq_locks` are operations. In a nutshell, the `call`$(r, f, (), ())$ operation asks the handler of the target r to make a call to the feature f on target r. The `unlock` operation unlocks the request queue of the processor that executes the operation. The `issue`$(q,$ `unlock`$)$ operation adds the `unlock` operation to q's action queue. The `pop_obtained_rq_locks` operation removes the top element from the stack of obtained request queue locks.

Example 6 (Initial configuration) In this example we define the initial configuration of a share market application. The domain of our application consist of a number of markets, a number of investors, and a number of issuers. Each issuer can offer a number of shares on each market. Each investor can have an amount of cash available on each market. With this cash, the investor can buy the shares that are available on the market. Investors can sell a share on the market where they bought the share. Selling shares increases the investor's amount of cash on the market. Each market determines the price for each share. Financial regulations require the investors to keep track of the markets on which they operate. For simplicity, we keep the price constant and we restrict ourselves to one market, two investors, and one issuer with one share.

We introduce a class *MARKET* for the market and a class *INVESTOR* for the investor. The issuers are represented through identifiers of class *INTEGER*. The root class *APPLICATION* contains the root procedure *make*, where the actors get created and where the trade begins.

We start with a bootstrap processor p_0, a root processor p_1 and a root object o_0 of root class type *APPLICATION*. The root object is referenced by r_0. The following initial configuration shows this:

$$\begin{array}{l}
\langle \\
\quad p_0 :: \texttt{call}(r_0, \mathit{make}, (), ()); \\
\qquad\quad \texttt{issue}(p_1, \texttt{unlock}); \\
\qquad\quad \texttt{pop\_obtained\_rq\_locks} \mid \\
\quad p_1 :: \\
, \\
\quad \text{locks:} \\
\qquad p_0 :: \ \text{orq}: (\{p_1\}) \ \ \text{rrq}: () \ \ \text{rcs}: () \ \ \text{unlocked} \\
\qquad p_1 :: \ \text{orq}: () \ \ \text{rrq}: () \ \ \text{rcs}: () \ \ \text{locked} \\
\quad \text{objects:} \\
\qquad p_0 :: \\
\qquad p_1 :: \ r_0 \rightarrow o_0 \\
\quad \text{once status:} \\
\quad \text{environments:} \\
\qquad p_0 :: \\
\qquad p_1 :: \\
\rangle
\end{array}$$

□

5.3. Mechanisms

Mechanisms are the machinery for the execution of code elements. In this section, we study the mechanisms that take place in a SCOOP program execution. Later on, we use these mechanisms to explain the semantics of code elements.

5.3.1. Issuing mechanism

In this section we look at how a processor p can add statements to the action queue of a processor q. We define a number of inference rules for the `issue` operation to get a result configuration in which a processor's action queue is extended with the new statements. We differentiate two main cases: p adds the statements to its own action queue, i.e. $p = q$, or p adds the statements to the action queue of a different processor, i.e. $p \neq q$. The first case is the non-separate case and the second one is the separate case.

For the non-separate case p puts the statements to the beginning of q's action queue, which is the same as putting the statements on top of the call stack. This requires that p is in possession of its own call stack lock.

Issue Operation – Non-Separate

$$\frac{\begin{array}{l} q = p \\ \neg\sigma.\mathit{are\_locks\_passed}(p) \\ \sigma.\mathit{cs\_locks}(p).\mathit{has}(q) \end{array}}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p, \sigma\rangle \rightarrow \langle p :: s_w; s_p, \sigma\rangle}$$

For the separate case we distinguish between a normal and a callback case. In the normal case p adds the statements to the end of q's action queue. This is the same as performing an enqueue operation on q's request queue. This requires that p is in possession of q's request queue lock. We also require that q does not have a lock on p to distinguish the normal case from the callback case.

Issue Operation – Separate

$$\frac{\begin{array}{l} q \neq p \\ \neg\sigma.\mathit{are\_locks\_passed}(p) \\ \sigma.\mathit{rq\_locks}(p).\mathit{has}(q) \\ \neg(\sigma.\mathit{rq\_locks}(q).\mathit{has}(p) \vee \sigma.\mathit{cs\_locks}(q).\mathit{has}(p)) \end{array}}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p \mid q :: s_q, \sigma\rangle \rightarrow \langle p :: s_p \mid q :: s_q; s_w, \sigma\rangle}$$

The callback case is given if q has a lock on p. In this situation we have p that can issue a statement s_w on q and then wait for q to complete the execution of this statement and we have q that could already be waiting for p to complete. If we handle this case the same way as the normal case then we might end up in a deadlock. Processor q would be waiting for p to finish and p would be waiting for q to finish. However, since s_w would be at the end of q's action queue and q would be waiting there cannot be any progress. This type of deadlock can be prevented by adding s_w not to the of q's action queue but to the beginning. This will make sure that q can execute the statement right away and hence p can continue. This in return will enable q to continue. As a prerequisite, p must possess q's call stack lock.

Issue Operation – Separate Callback

$$\frac{\begin{array}{l} q \neq p \\ \neg\sigma.\mathit{are\_locks\_passed}(p) \\ \sigma.\mathit{cs\_locks}(p).\mathit{has}(q) \\ \sigma.\mathit{rq\_locks}(q).\mathit{has}(p) \vee \sigma.\mathit{cs\_locks}(q).\mathit{has}(p) \end{array}}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p \mid q :: s_q, \sigma\rangle \rightarrow \langle p :: s_p \mid q :: s_w; s_q, \sigma\rangle}$$

5.3.2. Delegated execution mechanism

In this section we discuss how a processor q can delegate the execution of statements to a different processor p. We use this mechanism for the asynchronous postcondition evaluation. Processor q must make sure that the statements make sense in the context of processor p: The names that occur in these statements must be defined in the top environment of p and p must have the necessary locks to execute the statements. We restrict the statements that can be delegated to those that satisfy the following conditions:

- All names that occur in the statements are defined in q's top environment.
- Their execution only requires the top set of q's stack of obtained request queue locks.

For example, these conditions exclude statements that involve non-separate calls or separate callbacks because such calls require a call stack lock. If these conditions are met, we can transfer q's top environment and the top of q's obtained request queue locks to p. Given this context we can then execute the delegated statements on p rather than on q.

We introduce the $\texttt{execute\_delegated}(s_w, x, \{q_1, \ldots, q_m\})$ operation to set up a new context on p with an environment x and obtained request queue locks $\{q_1, \ldots, q_m\}$. To set up the new context, the operation uses the commands *push_env* and *delegate_obtained_rq_locks*. The command *delegate_obtained_rq_locks* requires that the request queue locks $\{q_1, \ldots, q_m\}$ are not in possession of an other processor anymore. It also requires that the request queues of $\{q_1, \ldots, q_m\}$ are locked. Once the context is set up, processor p executes the statements s_w and then gets rid of the context. To get rid of the context, we introduce the `leave_delegated` operation.

To delegate the execution of the statements s_w, processor q must make sure that its top environment x is set up correctly and it must make sure that the top set of its obtained request queue locks contains all locks $\{q_1, \ldots, q_m\}$ that are necessary for the execution of s_w. Processor q must then issue a `execute_delegated`$(s_w, x, \{q_1, \ldots, q_m\})$ operation to processor p. Processor q must then remove $\{q_1, \ldots, q_m\}$ from its stack of obtained request queue locks so that the *delegate_obtained_rq_locks* operation can take place.

Execute Delegated Operation

$$\frac{\begin{array}{l}\forall x \in \{q_1, \ldots, q_m\} \colon \neg \exists y \in \sigma.procs \colon \sigma.rq\_locks(y).has(x) \\ \forall x \in \{q_1, \ldots, q_m\} \colon \sigma.is\_rq\_locked(x) \\ \sigma' \overset{def}{=} \sigma.push\_env(p, x).delegate\_obtained\_rq\_locks(p, \{q_1, \ldots, q_m\})\end{array}}{\Gamma \vdash \langle p :: \texttt{execute\_delegated}(s_w, x, \{q_1, \ldots, q_m\}); s_p, \sigma\rangle \rightarrow \langle p :: s_w; \texttt{leave\_delegated}; s_p, \sigma'\rangle}$$

Leave Delegated Execution Operation

$$\frac{\begin{array}{l}\neg\sigma.envs(p).is\_empty \\ \neg\sigma.obtained\_rq\_locks(p).is\_empty \\ \sigma' \overset{def}{=} \sigma.pop\_env(p).pop\_obtained\_rq\_locks(p)\end{array}}{\Gamma \vdash \langle p :: \texttt{leave\_delegated}; s_p, \sigma\rangle \rightarrow \langle p :: s_p, \sigma'\rangle}$$

5.3.3. Notification mechanism

Processors can notify each other. A notification can optionally include a value. We use channels to describe such communication. Channels are described in Milner's π-calculus [20]. In the π-calculus, the expression $c(x).P$ denotes a process that is waiting for a notification sent on a channel c. Once the notification has been received, the value of the notification is bound to the variable x and the process continues with the expression P. The notification comes from a process that executes $\overline{c}y.Q$ that emits the value y on the channel c before executing Q.

The channels from the π-calculus seem to almost fit our requirements. However, we need to introduce the channel idea in two flavors: once as a notification mechanism with a value and once as a notification mechanism without a value. A processor sends a notification with a value r over a channel a as it executes the operation `result`(a, r). Similarly, the process sends a notification without a value over a channel a by executing the operation `notify`(a). For both cases, any processor can wait for a notification by executing the operation `wait`(a). In case a notification on a channel a carries a value, the value can be accessed with $a.data$. This way of accessing the value of a channel is different from the way it is done in the π-calculus. In the π-calculus, each value is bound to a variable. For our channels, we do not define a new variable for the value. Instead we use $a.data$ to identify the value of a channel a. With this approach we save us from introducing too many variables.

A number of inference rules describe the interaction between a processor that sent a notification over a channel and a processor that is waiting for a notification over the same channel. We distinguish two main cases: either a processor sends a notification to itself or it sends a notification to a different processor. The first case is the non-separate case and the latter case is the separate case. For each of these two main cases we distinguish whether the channel carries a notification with or without a value. For each of these sub cases we introduce one inference rule.

In the non-separate case, one processor has a `result`(a, r) operation or a `notify`(a) operation at the beginning of its action queue and a `wait`(a) operation on the same channel later in the action queue. In this case, we can remove the `wait`(a) operation along with the `result`(a, r) operation respectively the `notify`(a) operation. If the channel carries a value then the value must be installed on the processor. We can install the value on the processor, by substituting all occurrences of $a.data$ with the posted value in all the statements s_p after the `wait`(a) operation.

Wait and Result Operation – Non-Separate

$$\frac{}{\Gamma \vdash \langle p :: \texttt{result}(a, r); s_w; \texttt{wait}(a); s_p, \sigma\rangle \to \langle p :: s_w; s_p[r/a.data], \sigma\rangle}$$

Wait and Notify Operation – Non-Separate

$$\frac{}{\Gamma \vdash \langle p :: \texttt{notify}(a); s_w; \texttt{wait}(a); s_p, \sigma\rangle \to \langle p :: s_w; s_p, \sigma\rangle}$$

In the separate case, one processor has a `result`(a, r) or a `notify`(a) operation at the beginning of its action queue and a different processor has a `wait`(a) somewhere in its action queue. In this situation, we can remove the `wait`(a), `result`(a, r), and `notify`(a) from the action queues. In case the notification has a value, we can install the value in the statements s_p proceeding the `wait`(a).

Wait and Result Operation – Separate

$$\frac{}{\Gamma \vdash \langle p :: s_w; \texttt{wait}(a); s_p \mid q :: \texttt{result}(a, r); s_q, \sigma\rangle \to \langle p :: s_w; s_p[r/a.data] \mid q :: s_q, \sigma\rangle}$$

Wait and Notify Operation – Separate

$$\frac{}{\Gamma \vdash \langle p :: s_w; \texttt{wait}(a); s_p \mid q :: \texttt{notify}(a); s_q, \sigma\rangle \to \langle p :: s_w; s_p \mid q :: s_q, \sigma\rangle}$$

The operations presented here must be used so that each `wait` operation can be resolved with exactly one `result` or `notify` operation. To define this more precisely, we say that one statement s_1 weakly precedes a statement s_2 if and only if s_1 occurs earlier than s_2 in the same action queue or s_1 and s_2 occur in different action queues. We say that one statement s_1 strongly precedes a statement s_1 if and only if s_1 occurs earlier than s_2 in the same action queue. With these definitions, we can restate the condition as:

- For each `wait`(a) operation there must be either exactly one `result`(a, r) or exactly one `notify`(a) operation.
- For each `result`(a, r) or `notify`(a) operation there must be exactly one `wait`(a) operation.
- Each `result`(a, r) or `notify`(a) operation weakly precedes the `wait`(a) operation.

5.3.4. Expression evaluation mechanism

An expression can either be a literal, an entity, or a query call. The query call can contain actual arguments that are expressions themselves. In this section we discuss the general mechanism to evaluate expressions. We focus on the general approach and defer the evaluation of particular expressions to later sections.

We introduce an operation `eval`(a, e) that takes a channel a and an expression e. Every `eval`(a, e) operation determines the value r of the expression e and then sends a notification with value r on channel a. This means that every `eval`(a, e) operation creates a `result`(a, r) operation in the action queue. It is therefore important to follow every `eval`(a, e) operation with exactly one `wait`(a) to receive the notification with the value.

5.3.5. Locking and unlocking mechanism

A processor p that wants to execute a feature must first obtain the request queue locks of a number of processors $\{q_1, \ldots, q_n\}$. For this, p adds $\{q_1, \ldots, q_n\}$ on top of its obtained request queue locks stack. Only then can p issue statements to these processors. The `lock`$(\{q_1, \ldots, q_n\})$ operation serves this purpose. The operation requires that none of the request queues is already locked.

Lock Operation

$$\frac{\begin{array}{l}\neg\exists q_i \in \{q_1, \ldots, q_m\} \colon \sigma.\mathit{is\_rq\_locked}(q_i) \\ \sigma' \overset{def}{=} \sigma.\mathit{lock\_rqs}(p, \{q_1, \ldots, q_m\})\end{array}}{\Gamma \vdash \langle p :: \texttt{lock}(\{q_1, \ldots, q_m\}); s_p, \sigma\rangle \to \langle p :: s_p, \sigma'\rangle}$$

Once p is done with the execution of the feature, it asks $\{q_1, \ldots, q_n\}$ to unlock their request queues once they are done with the issued statements. For this, we introduce the `unlock` operation that unlocks the request queue. Processor p issues the `unlock` operation to processors $\{q_1, \ldots, q_n\}$. This operation requires that the request queue is indeed locked and that no processor possesses the request queue lock.

As a side note, Brooke, Paige, and Jacob [5] noticed in Section 7.3 that `unlock` operations are not optimal. In essence, it could be possible to unlock the request queue of a processor q_i directly after p issued all statements. The request queue lock is important to guarantee exclusive access on q_i's request queue. However, as soon as p issued all statements on q_i, this lock is no longer needed. Unlocking the request queue right away could improve the performance in some situations because q_i's request queue could be locked again earlier and hence another processor that is waiting for this lock could proceed earlier.

Unlock Operation

$$\frac{\begin{array}{l}\sigma.\mathit{is\_rq\_locked}(p) \\ \forall q \in \sigma.\mathit{procs} \colon \neg\sigma.\mathit{rq\_locks}(q).\mathit{has}(p) \\ \sigma' \overset{def}{=} \sigma.\mathit{unlock\_rq}(p)\end{array}}{\Gamma \vdash \langle p :: \texttt{unlock}; s_p, \sigma\rangle \to \langle p :: s_p, \sigma'\rangle}$$

After p issued the `unlock` operations, it can remove $\{q_1, \ldots, q_n\}$ from its stack of obtained request queue locks using the `pop_obtained_rq_locks` operation. This ensures that the `unlock` operations can proceed.

Pop Obtained Request Queue Locks

$$\frac{\sigma' \overset{def}{=} \sigma.\mathit{pop\_obtained\_rq\_locks}(p)}{\Gamma \vdash \langle p :: \texttt{pop\_obtained\_rq\_locks}; s_p, \sigma\rangle \to \langle p :: s_p, \sigma'\rangle}$$

5.3.6. Write and read mechanism

A processor p can set a value v of an entity with name x using the `write`(x, v) operation. This operation uses the *set_val* command. Hence, p can both set attribute values of its current object and values of entities in its top environment.

Write Value Operation

$$\frac{\sigma' \overset{def}{=} \sigma.\mathit{set\_val}(p, x, v)}{\Gamma \vdash \langle p :: \texttt{write}(x, v); s_p, \sigma\rangle \to \langle p :: s_p, \sigma'\rangle}$$

Similarly, a processor p can execute the `read`(x, a) operation to read a value of an entity with name x and send the value over channel a. The `read` operation does not present its result in a `result` operation because, unlike an `eval` operation, a `read` operation always produces a result for the surrounding action queue. It is easier to do the substitution of the channel access directly. Later on, we introduce the `eval` operation for entity expressions. This variant of the `eval` operation makes use of the `read` operation and presents the result in a `result` operation.

Read Value Operation

$$\frac{}{\Gamma \vdash \langle p :: \texttt{read}(x, a); s_p, \sigma\rangle \rightarrow \langle p :: s_p[\sigma.val(p, x)/a.data], \sigma\rangle}$$

Finally, we have the `set_not_fresh` operation in a variant for once functions and in a variant for once procedures. This operation sets the once status of a once routine. The variant `set_not_fresh`(f, r) sets the once status of a once function f to non-fresh with value r. If f is of separate type then the once function becomes non-fresh on all processors in the system. If f has a non-separate type then f becomes non-fresh only on processor p. The variant `set_not_fresh`(f) sets the once status of a once procedure f to non-fresh on processor p.

Set Once Routine Not Fresh Operation – Function

$$\frac{\begin{array}{l} f \in \textbf{FUNCTION} \wedge f.is\_once \\ \sigma' \overset{def}{=} \sigma.set\_once\_func\_not\_fresh(p, f, r) \end{array}}{\Gamma \vdash \langle p :: \texttt{set\_not\_fresh}(f, r); s_p, \sigma\rangle \rightarrow \langle p :: s_p, \sigma'\rangle}$$

Set Once Routine Not Fresh Operation – Procedure

$$\frac{\begin{array}{l} f \in \textbf{FUNCTION} \wedge f.is\_once \\ \sigma' \overset{def}{=} \sigma.set\_once\_proc\_not\_fresh(p, f) \end{array}}{\Gamma \vdash \langle p :: \texttt{set\_not\_fresh}(f); s_p, \sigma\rangle \rightarrow \langle p :: s_p, \sigma'\rangle}$$

5.3.7. Flow control mechanism

Next to flow control instructions in the user code we introduce flow control operations. We use these operations to implement flow control in the inference rules. With these operations we can write fewer inference rules because we can handle multiple variants in one inference rule.

The `provided` x `then` s_t `else` s_f `end` operation takes the condition x as an argument. The operation either executes s_t if x indicates that the condition is true or s_f if x indicates that the condition is false. For each possibility there is one inference rule. The condition x can either be an instance of the **BOOLEAN** ADT or it can be a reference that points to an object of class type *BOOLEAN*. To decide which branch to take, we must evaluate x. If x is an instance of the **BOOLEAN** ADT then we can determine which instance x is, i.e. *true* or *false*. If x is a reference then we must get the referenced object and see which boolean value it represents. For this purpose, we evaluate the attribute *item* of the referenced object.

If Operation – True

$$\frac{\begin{array}{l} y \overset{def}{=} \begin{cases} x & if x \in \textbf{BOOLEAN} \\ \sigma.att\_val(\sigma.ref\_obj(x), item) & if x \in \textbf{REF} \wedge \sigma.ref\_obj(x).class\_type = BOOLEAN \\ false & otherwise \end{cases} \\ y = true \end{array}}{\Gamma \vdash \langle p :: \texttt{provided}\ x\ \texttt{then}\ s_t\ \texttt{else}\ s_f\ \texttt{end}; s_p, \sigma\rangle \rightarrow \langle p :: s_t; s_p, \sigma\rangle}$$

If Operation – False

$$\frac{\begin{array}{l} y \overset{def}{=} \begin{cases} x & if\, x \in \mathbf{BOOLEAN} \\ \sigma.att\_val(\sigma.ref\_obj(x), item) & if\, x \in \mathbf{REF} \wedge \sigma.ref\_obj(x).class\_type = BOOLEAN \\ true & otherwise \end{cases} \\ y = false \end{array}}{\Gamma \vdash \langle p :: \ \texttt{provided}\ x\ \texttt{then}\ s_t\ \texttt{else}\ s_f\ \texttt{end}; s_p, \sigma\rangle \rightarrow \langle p :: s_f; s_p, \sigma\rangle}$$

The `provided` x `then` s_t `else` s_f `end` operation has two branches. Sometimes it is necessary to only have one branch. We introduce the `nop` operation that can be executed without an effect. It can be used in the conditional operation to define an empty branch. The `nop` operation can also be used to indicate that an action queue is empty.

No Operation

$$\frac{}{\Gamma \vdash \langle p :: \texttt{nop}; s_p, \sigma\rangle \rightarrow \langle p :: s_p, \sigma\rangle}$$

5.4. Code elements

In this section we explain the semantics of code elements: entity expressions, literal expressions, feature calls, feature applications, creation instructions, flow control instructions, and assignment instructions.

5.4.1. Entity expressions

To evaluate an entity expression we introduce a variant of the `eval`(a, e) operation. In our variant, the expression e is an entity. We use the `read` operation to send a notification with the value of the entity over a new channel a'. We then use the value of this channel to define the result of the `eval` operation.

Entity Expression

$$\frac{\begin{array}{l} e \in \mathbf{ENTITY} \\ a'\ is\ fresh \end{array}}{\Gamma \vdash \langle p :: \texttt{eval}(a, e); s_p, \sigma\rangle \rightarrow \langle p :: \texttt{read}(e.name, a'); \texttt{result}(a, a'.data); s_p, \sigma\rangle}$$

5.4.2. Literal expressions

For the evaluation of a literal expression, we introduce a variant of the `eval`(a, e) operation. In this variant, e is an instance of the **LITERAL** ADT. To evaluate a non-void literal expression, we create a new object of the literal class type so that the new object represents the literal value. For this purpose, we use the query obj of the **LITERAL** ADT. Since the type of every literal is non-separate, we create the new object on the processor that evaluates the literal expression. The reference r to the new object is the result of the evaluation. To evaluate a void literal, we take the void reference.

Literal Expression

$$\frac{\begin{array}{l} e \in \mathbf{LITERAL} \\ \sigma' \overset{def}{=} \begin{cases} \sigma & if\, e = \mathbf{Void} \\ \sigma.add\_obj(p, e.obj) & otherwise \end{cases} \\ r \overset{def}{=} \begin{cases} void & if\, e = \mathbf{Void} \\ \sigma'.ref(\sigma'.last\_added\_obj) & otherwise \end{cases} \end{array}}{\Gamma \vdash \langle p :: \texttt{eval}(a, e); s_p, \sigma\rangle \rightarrow \langle p :: \texttt{result}(a, r); s_p, \sigma'\rangle}$$

5.4.3. Feature calls

A feature call can occur in two ways. First, a feature call can be a call to a command in a command instruction. Second, a feature call can be a call to a query in an expression. In this section we study both variants. A processor p that executes a feature call $e_0.f(e_1, \dots, e_n)$ goes through the following steps:

1. Target evaluation: Evaluate the target expression e_0 and let q denote the handler of the target.
2. Argument passing: Evaluate the actual arguments expressions $(e_1, \dots, e_n)$.
3. Lock passing: Determine which locks to pass to q.
   - Take all request queue locks and call stack locks, if a controlled actual argument gets attached to an attached formal argument of reference type.
   - Take all request queue locks and call stack locks, if the feature call is a separate callback, i.e. q has a lock on p.
   - Otherwise, take no locks.
4. Feature request:
   - Ask q to apply f to the target immediately and wait until the execution terminates, if any of the following conditions holds:
     - The feature call is non-separate, i.e. $p = q$.
     - The feature call is a separate callback, i.e. q has a lock on p.
   - Otherwise, ask q to apply f to the target after the previous feature requests.
5. Wait by necessity: If f is a query then wait for the result.
6. Lock revocation: If lock passing happened then wait for the locks to come back.

A command instruction is a statement in the action queue. A query is an expression on the right hand side of an assignment, a condition in a flow control instruction, or an actual argument in a feature call. Whenever a query occurs in one of these constructs, the inference rule of the construct encloses the query in an `eval` operation. To handle feature calls, we can therefore provide one inference rule for a command instruction and one variant of the `eval` operation for query calls.

In each case, we first have to evaluate the target expression and all actual argument expressions. For each of these expressions e_i we use one $\texttt{eval}(a_{e_i}, e_i)$ operation and a corresponding $\texttt{wait}(a_{e_i})$ operation with a fresh channel a_{e_i}. Each of the channel values gets used in the subsequent `call` operation. With this, we handled the target evaluation and the argument passing step. We defer the attachment of the actual arguments to the formal arguments to the point where the called feature gets applied. The reason for this is simple: At this point the context for the feature application does not exist yet.

The `call` operation takes care of the remaining steps. The operation exists in two variants, one for command instructions and one for queries. The variant for queries takes a channel a' that is used for the result of the query. Since a call to a command does not produce a result, such a channel is not required for command instructions. Both `call` variants take the reference to the target a_{e_0}, the feature f to be called, the references to the actual arguments $(a_{e_1}.data, \dots, a_{e_n}.data)$, and the actual argument expressions $(e_1, \dots, e_n))$. The actual argument expressions are used to check whether there is a controlled actual argument. This information is used to determine whether the locks should be passed.

Command Instruction

$$\frac{\forall i \in \{0, \dots, n\} \colon a_{e_i} \text{ } is\ fresh}{\begin{aligned}&\Gamma \vdash \langle p :: e_0.f(e_1, \dots, e_n); s_p, \sigma\rangle \to \\ &\quad \langle p :: \texttt{eval}(a_{e_0}, e_0); \texttt{eval}(a_{e_1}, e_1); \dots; \texttt{eval}(a_{e_n}, e_n); \\ &\qquad \texttt{wait}(a_{e_0}); \texttt{wait}(a_{e_1}); \dots; \texttt{wait}(a_{e_n}); \\ &\qquad \texttt{call}(a_{e_0}.data, f, (a_{e_1}.data, \dots, a_{e_n}.data), (e_1, \dots, e_n)); \\ &\qquad s_p, \sigma\rangle\end{aligned}}$$

Query Expression

$$\frac{\begin{array}{l}\forall i \in \{0,\dots,n\}\colon a_{e_i}\ \mathit{is\ fresh}\\ a'\ \mathit{is\ fresh}\end{array}}{\begin{array}{l}\Gamma \vdash \langle p :: \mathtt{eval}(a, e_0.f(e_1,\dots,e_n)); s_p, \sigma\rangle \rightarrow\\ \quad\langle p :: \mathtt{eval}(a_{e_0}, e_0); \mathtt{eval}(a_{e_1}, e_1); \dots; \mathtt{eval}(a_{e_n}, e_n);\\ \qquad \mathtt{wait}(a_{e_0}); \mathtt{wait}(a_{e_1}); \dots; \mathtt{wait}(a_{e_n});\\ \qquad \mathtt{call}(a', a_{e_0}.data, f, (a_{e_1}.data, \dots, a_{e_n}.data), (e_1,\dots,e_n));\\ \qquad \mathtt{result}(a, a'.data);\\ \qquad s_p, \sigma\rangle\end{array}}$$

In the following, we take a closer look at the `call` operation. Both variants take the reference to the target r_o, the feature f to be called, the references to the actual arguments $(r_1, \dots, r_n)$, and the actual argument expressions $(e_1, \dots, e_n)$. The variant for queries takes an additional channel a to be used for the result of the query. In a first step, we must evaluate the handler q of the target. The handler is used in an `issue` operation to issue a feature request on the responsible processor. The feature request comes in the form of an `apply` operation. The `apply` operation takes a channel a for the communication between p and q, the target reference r_0, the called feature f, the references to the actual arguments $(r_1, \dots, r_n)$, the caller processor p, and the passed locks $\bar{l}$.

Clarification 3 (Lock passing) Processor p passes all its request queue locks and all its call stack locks, either if there is a controlled actual argument that will get attached to an attached formal argument of reference type or if the feature call is a separate callback. An attached formal argument of reference type means that the request queue lock or the call stack lock on the actual argument's handler is required during the application of f. A controlled actual argument means that p has a request queue lock or a call stack lock on the handler of the actual argument. In short, p has a lock that is required by q and thus we have to pass the locks. A separate callback is given if q has a lock on p. In this situation p can issue a statement to q and then wait for q to complete. However, processor q could already be waiting for p to complete. To handle this case, the `issue` operation in the `call` operation triggers an immediate execution by adding the `apply` to the beginning of q's action queue. The `issue` operation requires that p has the call stack lock of q. If we want q to do an immediate execution, we have to give back q's call stack lock.

In both cases, we have to wait for the locks to come back. Thus it does not hurt to pass all the locks in both cases. In contrast to Nienaltowski's [22] description of SCOOP, p only passes the locks that it really has. In particular, p does not pass its own request queue lock in situations where p does not possess this lock, such as when the processor that called p possesses p's request queue lock. $\square$

In the two cases where we pass the locks, $\bar{l}$ is defined as $(\sigma.\mathit{rq\_locks}(p), \sigma.\mathit{cs\_locks}(p))$. In all other cases there is no lock passing and thus $\bar{l} = (\{\}, \{\})$. At this point we just determine which locks to pass. The actual lock passing action will be executed by q. Similarly, the actual lock revocation action will be executed by q.

For command calls, lock passing is the only reason to wait. In this case, p creates a fresh channel a to wait for a notification from q. The notification arrives when q is ready to return the locks. For query calls, p has to wait for the result. Processor p uses channel a, given in the `call` operation, to wait for the result. This has the advantage that once the result arrives, it will be substituted after the `call` operation, i.e in the `result` operation of the `eval` operation.

Call Operation – Command

$$\dfrac{\begin{array}{l} q \overset{def}{=} \sigma.handler(r_0) \\ \bar{l} \overset{def}{=} \begin{cases} \mathit{if} \\ \quad q \neq p \wedge \exists i \in \{1, \dots, n\} \colon \Gamma \vdash e_i : t \wedge is\_controlled(t) \wedge \Gamma \vdash f.formals(i) : (!, g, c) \wedge c.is\_ref \\ \mathit{then} \\ \qquad (\sigma.rq\_locks(p), \sigma.cs\_locks(p)) \\ \mathit{if} \\ \quad q \neq p \wedge (\sigma.rq\_locks(q).has(p) \vee \sigma.cs\_locks(q).has(p)) \\ \mathit{then} \\ \qquad (\sigma.rq\_locks(p), \sigma.cs\_locks(p)) \\ \mathit{otherwise} \\ \qquad (\{\}, \{\}) \end{cases} \\ a\ is\ fresh \end{array}}{\begin{array}{l} \Gamma \vdash \langle p :: \texttt{call}(r_0, f, (r_1, \dots, r_n), (e_1, \dots, e_n)); s_p, \sigma \rangle \rightarrow \\ \quad \langle p :: \texttt{issue}(q, \texttt{apply}(a, r_0, f, (r_1, \dots, r_n), p, \bar{l})); \\ \qquad \texttt{provided } \bar{l} \neq (\{\}, \{\}) \texttt{ then wait}(a) \texttt{ else nop end}; \\ \qquad s_p, \sigma \rangle \end{array}}$$

Call Operation – Query

$$\dfrac{\begin{array}{l} q \overset{def}{=} \sigma.handler(r_0) \\ \bar{l} \overset{def}{=} \begin{cases} \mathit{if} \\ \quad q \neq p \wedge \exists i \in \{1, \dots, n\} \colon \Gamma \vdash e_i : t \wedge is\_controlled(t) \wedge \Gamma \vdash f.formals(i) : (!, g, c) \wedge c.is\_ref \\ \mathit{then} \\ \qquad (\sigma.rq\_locks(p), \sigma.cs\_locks(p)) \\ \mathit{if} \\ \quad q \neq p \wedge (\sigma.rq\_locks(q).has(p) \vee \sigma.cs\_locks(q).has(p)) \\ \mathit{then} \\ \qquad (\sigma.rq\_locks(p), \sigma.cs\_locks(p)) \\ \mathit{otherwise} \\ \qquad (\{\}, \{\}) \end{cases} \end{array}}{\begin{array}{l} \Gamma \vdash \langle p :: \texttt{call}(a, r_0, f, (r_1, \dots, r_n), (e_1, \dots, e_n)); s_p, \sigma \rangle \rightarrow \\ \quad \langle p :: \texttt{issue}(q, \texttt{apply}(a, r_0, f, (r_1, \dots, r_n), p, \bar{l})); \texttt{wait}(a); s_p, \sigma \rangle \end{array}}$$

Example 7 (Feature call) This example demonstrates the inference rules for feature calls. For this purpose, we come back to the share market example. We start in a situation where the root processor p_1 started with the execution of the procedure *do_transaction* on the root object o_0. This procedure is shown in listing 1.

Listing 1: Application class with implementation

```
class APPLICATION

create
  make

feature -- Initialization
  make
    do
      ...
    end

feature {APPLICATION} -- Implementation
  market: separate MARKET
    -- The market.

  do_transaction (
    a_first_investor: separate INVESTOR;
    a_second_investor: separate INVESTOR;
    a_issuer_id: INTEGER
  )
      -- Make each of the two investors 'a_first_investor' and 'a_second_investor' buy and then sell a
          share of the issuer with identifier 'a_issuer_id'.
    do
      a_first_investor.buy (Current.market, a_issuer_id)
      a_second_investor.buy (Current.market, a_issuer_id)
      a_first_investor.sell (Current.market, a_issuer_id)
      a_second_investor.sell (Current.market, a_issuer_id)
    end
end
```

The following configuration is our starting point. Processor p_1 has one environment for the callee procedure *make* and one for the called procedure *do_transaction*. Processor p_2 handles the market object o_{35}, processor p_3 handles the first investor object o_{44}, and processor p_4 handles the second investor object o_{46}. The market object references three arrays. The cash array o_{25} shows that each investor has the same amount of cash. The available shares array o_{33} shows that there is one issuer with one available share. The owned shares array o_{38} shows that none of the investors owns a share. The action queue of p_1 indicates that p_1 is about to make the feature calls on the two investors. For this purpose, p_1 obtained the request queue locks of their handlers.

$$
\begin{array}{l}
\langle \\
\quad p_1 :: a\_first\_investor.buy(current.market, a\_issuer\_id); \\
\qquad a\_second\_investor.buy(current.market, a\_issuer\_id); \\
\qquad a\_first\_investor.sell(current.market, a\_issuer\_id); \\
\qquad a\_second\_investor.sell(current.market, a\_issuer\_id); \\
\qquad \ldots \mid \\
\quad p_2 :: \mid \\
\quad p_3 :: \mid \\
\quad p_4 :: \\
, \\
\quad \text{locks:} \\
\qquad p_1 :: \text{ orq: } (\{\}, \{p_3, p_4\}) \text{ rrq: } (\{\}, \{\}) \text{ rcs: } (\{\}, \{\}) \text{ locked} \\
\qquad p_2 :: \text{ orq: } () \text{ rrq: } () \text{ rcs: } () \text{ unlocked} \\
\qquad p_3 :: \text{ orq: } () \text{ rrq: } () \text{ rcs: } () \text{ locked} \\
\qquad p_4 :: \text{ orq: } () \text{ rrq: } () \text{ rcs: } () \text{ locked} \\
\quad \text{objects:} \\
\qquad p_1 :: r_0 \to o_0(market \to r_1), r_{39} \to o_{48}(1) \\
\qquad p_2 :: r_1 \to o_{35}(cash \to r_{16}, available\_shares \to r_{23}, owned\_shares \to r_{29}), \\
\qquad\quad r_{16} \to o_{25}[r_{21}, r_{22}], r_{21} \to o_{23}(100), r_{22} \to o_{24}(100), \\
\qquad\quad r_{23} \to o_{33}[r_{28}], r_{28} \to o_{32}(1), \\
\qquad\quad r_{29} \to o_{38}[[r_{34}], [r_{35}]], r_{34} \to o_{41}(0), r_{35} \to o_{42}(0) \\
\qquad p_3 :: r_6 \to o_{44}(id \to r_{36}), r_{36} \to o_{43}(1) \\
\qquad p_4 :: r_8 \to o_{46}(id \to r_{37}), r_{37} \to o_{45}(2) \\
\quad \text{once status:} \\
\quad \text{environments:} \\
\qquad p_1 :: l\_first\_investor \to r_6, l\_second\_investor \to r_8, \textbf{Current} \to r_0 \;/ \\
\qquad\quad a\_first\_investor \to r_6, a\_second\_investor \to r_8, a\_issuer\_id \to r_{39}, \textbf{Current} \to r_0 \\
\qquad p_2 :: \\
\qquad p_3 :: \\
\qquad p_4 :: \\
\rangle
\end{array}
$$

The inference rule for command instructions leads to the following configuration, where a_{59}, a_{60}, and a_{61} are fresh channels:

```
$\langle$
   $p_1$ :: eval($a_{59}, a\_first\_investor$);
          eval($a_{60}, current.market$);
          eval($a_{61}, a\_issuer\_id$);
          wait($a_{59}$);
          wait($a_{60}$);
          wait($a_{61}$);
          call($a_{59}.data, buy, (a_{60}.data, a_{61}.data), (current.market, a\_issuer\_id)$);
          $a\_second\_investor.buy(current.market, a\_issuer\_id)$;
          $a\_first\_investor.sell(current.market, a\_issuer\_id)$;
          $a\_second\_investor.sell(current.market, a\_issuer\_id)$;
          $\ldots$ |
    $p_2$ :: |
    $p_3$ :: |
    $p_4$ ::
,
   $\ldots$
$\rangle$
```

First, we have to evaluate the target expression and the actual argument expressions. After the evaluation, we have the following configuration:

```
$\langle$
   $p_1$ :: call($r_6, buy, (r_1, r_{39}), (current.market, a\_issuer\_id)$);
          $a\_second\_investor.buy(current.market, a\_issuer\_id)$;
          $a\_first\_investor.sell(current.market, a\_issuer\_id)$;
          $a\_second\_investor.sell(current.market, a\_issuer\_id)$;
          $\ldots$ |
    $p_2$ :: |
    $p_3$ :: |
    $p_4$ ::
,
   $\ldots$
$\rangle$
```

During the execution of the `call` operation, p_1 determines that no locks need to be passed. Then p_1 executes an `issue` operation to enqueue an `apply` operation to the action queue of the first investor's handler. □

5.4.4. Feature applications

A feature call by a client processor q results in a feature request for a supplier processor p. A *feature application* is the serving of the feature request. In this section, we discuss how p applies a feature f on a target referenced by r_0. Processor p takes the following steps:

1. Once status update: If f is a once routine then set its status to non-fresh.
2. Lock passing: Pass the locks from q to p.

3. Argument passing: Bind the actual arguments to the formal arguments. Arguments of expanded type that are handled by a different processor than p must be deep imported by p.
4. Synchronization: Involve the scheduler to wait until the following synchronization conditions are satisfied atomically:
   - Processor p owns the request queue lock of each processor q such that:
     - Processor q handles an actual argument of f and the corresponding formal argument has an attached reference type.
     - Processor p and processor q are different.
     - Processor p does not have q's request queue lock.
     - Processor q does not have p's request queue lock.
   - The precondition of f holds.
5. Execution:
   - If f is a non-once routine or a fresh once routine then run its body.
   - If f is a non-fresh procedure then do nothing. If f is a non-fresh function then take its once value as the result.
   - If f is an attribute then evaluate it.
6. Postcondition evaluation: Evaluate the postcondition, if any of the following conditions is given:
   - A feature call in the postcondition requires a lock that was not obtained in the synchronization step.
   - The evaluation of the postcondition involves lock passing.

   Otherwise ask any processor whose request queue lock was obtained in the synchronization step to evaluate the postcondition.
7. Lock releasing: Ask each processor whose request queue has been locked in the synchronization step to unlock its request queue after it is done with the feature requests issued by p.
8. Invariant evaluation: Evaluate the invariant.
9. Result returning: If f is a query then return the result to q. If the result is of expanded type and $p \neq q$ then the result must be deep imported by q.
10. Lock revocation: Return the passed locks from p to q.

Each feature application starts with an operation $\texttt{apply}(a, r_0, f, (r_1, \ldots, r_n), q, \bar{l})$ in the action queue of processor p. The channel a is used to communicate with the client processor q: If the called feature f is a procedure and the caller processor q passed some locks then a is used to signal that the locks returned. If f is query then a is used to return the value. The reference r_0 points to the target of the call. The references $(r_1, \ldots, r_n)$ point to the actual arguments. The tuple $\bar{l}$ contains the locks to be passed from q to p.

If we take a look at the execution step, we can differentiate three cases:

- The feature f is a non-once routine or a fresh once routine.
- The feature f is a non-fresh once routine.
- The feature f is an attribute.

For each of these cases, we introduce one inference rule. Each inference rule covers one variant of the `apply` operation. We continue with the most involved case: The feature f is a non-once routine or a fresh once routine.

Application Operation – Non-Once Routine or Fresh Once Routine

$$\frac{\begin{array}{l} f \in \textbf{ROUTINE} \wedge f.\mathit{is\_once} \rightarrow \sigma.\mathit{is\_fresh}(p, f) \\ \sigma.\mathit{handler}(r_0) = p \\ \neg\sigma.\mathit{are\_locks\_passed}(p) \\ \sigma' \overset{def}{=} \begin{cases} \sigma.\mathit{set\_once\_func\_not\_fresh}(p, f, \mathit{void}) & \mathit{if}\, f \in \textbf{FUNCTION} \wedge f.\mathit{is\_once} \\ \sigma.\mathit{set\_once\_proc\_not\_fresh}(p, f) & \mathit{if}\, f \in \textbf{PROCEDURE} \wedge f.\mathit{is\_once} \\ \sigma & \mathit{otherwise} \end{cases} \\ \sigma'' \overset{def}{=} \sigma'.\mathit{pass\_locks}(q, p, \bar{l}).\mathit{push\_env\_with\_feature}(p, f, r_0, (r_1, \ldots, r_n)) \\ \overline{g}_{\mathit{required\_locks}} \overset{def}{=} \\ \quad \{p\} \cup \{x \in \textbf{PROC} \mid \exists i \in \{1, \ldots, n\}, g, c \colon \Gamma \vdash f.\mathit{formals}(i) : (!, g, c) \wedge c.\mathit{is\_ref} \wedge x = \sigma''.\mathit{handler}(r_i)\} \\ \overline{g}_{\mathit{required\_cs\_locks}} \overset{def}{=} \\ \quad \{x \in \overline{g}_{\mathit{required\_locks}} \mid x = p \vee (x \neq p \wedge (\sigma''.\mathit{rq\_locks}(x).\mathit{has}(p) \vee \sigma''.\mathit{cs\_locks}(x).\mathit{has}(p)))\} \\ \overline{g}_{\mathit{required\_rq\_locks}} \overset{def}{=} \overline{g}_{\mathit{required\_locks}} \setminus \overline{g}_{\mathit{required\_cs\_locks}} \\ \overline{g}_{\mathit{missing\_rq\_locks}} \overset{def}{=} \{x \in \overline{g}_{\mathit{required\_rq\_locks}} \mid \neg\sigma''.\mathit{rq\_locks}(p).\mathit{has}(x)\} \\ \forall x \in \overline{g}_{\mathit{required\_cs\_locks}} \colon \sigma''.\mathit{cs\_locks}(p).\mathit{has}(x) \\ a_{inv}\ \mathit{is\ fresh} \wedge a'\ \mathit{is\ fresh} \end{array}}{\Gamma \vdash \langle p :: \texttt{apply}(a, r_0, f, (r_1, \ldots, r_n), q, \bar{l}); s_p, \sigma\rangle \rightarrow}$$

```
⟨$p$ :: check_pre_and_lock_rqs($\overline{g}_{missing\_rq\_locks}, f$);
    provided $f \in \textbf{FUNCTION} \wedge f.\mathit{is\_once}$ then
        $f.\mathit{body}$
            [$\mathit{result} := y$; read($\textbf{Result}, a_r$); set_not_fresh($f, a_r.\mathit{data}$) where $a_r$ is fresh/$\mathit{result} := y$]
            [create $\mathit{result}.y$; read($\textbf{Result}, a_r$); set_not_fresh($f, a_r.\mathit{data}$) where $a_r$ is fresh/
                create $\mathit{result}.y$]
    else
        $f.\mathit{body}$
    end;
    check_post_and_unlock_rqs($\overline{g}_{missing\_rq\_locks}, f$);
    provided $f.\mathit{class\_type}.\mathit{inv\_exists} \wedge f.\mathit{is\_exported}$ then
        eval($a_{inv}, f.\mathit{class\_type}.\mathit{inv}$); wait($a_{inv}$)
    else
        nop
    end;
    provided $f \in \textbf{FUNCTION}$ then
        read($\textbf{Result}, a'$); return($a, a'.\mathit{data}, q$)
    else
        return($a, q$)
    end;
    $s_p, \sigma''$⟩
```

The condition states that each processor can only apply a feature on one of its own objects. The condition also states the p must not have passed its locks. This part of the condition is always given because p waits whenever it passes its locks. In a first step, we define an updated state σ' to set f's once status to non-fresh, in case f is a once routine. We do this before deep importing the actual arguments to avoid the following contradiction.

Clarification 4 (When to set the status of a fresh once routine to non-fresh) We assume f is either a once procedure or a non-separate once routine. We know that f was fresh at the beginning of the `apply` operation. We assume that we passed an expanded actual argument that is handled by a processor $g \neq p$. Therefore p has to deep import the actual argument. We assume furthermore that the class type of the actual argument has the once routine f and that f is non-fresh on g. If we would deep import before setting f as non-fresh on p, then the deep import operation would take over the once status of f from processor g to processor p. But then the `apply` operation on p would not make much sense anymore because f would now be non-fresh on p. If we set f as non-fresh at the beginning of the `apply` operation, the the deep import operation does not take over the once status from g because f is already non-fresh on p. $\square$

We define an updated state σ'' in which the locks are passed from q to p and in which we created a new environment with the actual arguments $(r_1, \dots, r_n)$. The call to the *push_env_with_feature* feature takes care of copying and deep importing actual arguments of expanded type. The caller processor q can also pass an empty tuple $(\{\}, \{\})$ which simply means that q did not pass any locks.

In the next step we have to synchronize. For each target expressions in the body of f, we can get its controlling entity. Each of these controlling entities is mapped to an object and each of these objects is handled by a processor. For each of these processors we must either have a request queue lock or a call stack lock. We have seen that there are three types of calls: non-separate calls, separate calls, and separate callback. Non-separate calls and separate callbacks require a call stack lock. Separate calls require a request queue lock. This leads to two sets of required locks: one set with required request queue locks and another set with required call stack locks. The set of required call stack locks is composed of p that will lead to a non-separate call and all the processors that will lead to separate callbacks. The set of required request queue locks is composed of the processors that will lead to separate calls. We define two sets for these two categories: $\overline{g}_{required\_cs\_locks}$ and $\overline{g}_{required\_rq\_locks}$.

Each processor initially has its own call stack lock as its obtained call stack lock. This call stack never gets unlocked. This means that other call stack locks cannot be obtained; they must be retrieved through lock passing. The condition of the inference rule expresses this: $\forall x \in \overline{g}_{required\_cs\_locks} \colon \sigma''.cs\_locks(p).has(x)$. We can be assured that p did not pass its own call stack lock because otherwise p would be waiting. The remaining required call stack locks are the ones for the processors that will lead to separate callbacks. Note that the lock passing conditions are not sufficient to guarantee that the call stack locks for separate callbacks are always available.

As for the request queue locks, we calculate the missing request queue locks $\overline{g}_{missing\_rq\_locks}$ as the required request queue locks minus the already owned request queue locks. The already owned request queue locks are the previously obtained request queue locks and the retrieved request queue locks. In the synchronization step, we must obtain the difference on behalf of p. If this is not possible because some of the missing request queue locks are not available then we must wait. This can potentially lead to a deadlock. To get the missing request queue locks, we introduce the `check_pre_and_lock_rqs` operation. This operation takes $\overline{g}_{missing\_rq\_locks}$ and the feature f. If the execution succeeds, p has the request queue locks of $\overline{g}_{missing\_rq\_locks}$ and the precondition of f holds.

We can be assured that each processor g whose obtained request queue lock we got in the synchronization step must be in possession of its call stack lock. If g was not in possession of its call stack lock, it must have passed its locks. This means that g is executing a feature call and still waiting for the locks to return. In order to execute the feature call, there must have been a lock on g's request queue lock so that its action queue can contain the feature call. The request queue must still be locked because g is still executing the feature call. Hence, it would not have been possible to obtain g's request queue lock. The only exception is the bootstrap processor. However this processor only plays a role in the system setup and it never passes its own call stack lock.

Once we got all the required locks, we can execute the body. For once functions we must update the once status whenever we write to the result entity as part of an assignment instruction or as part of a creation instruction. For this purpose we add a `read` operation and a `set_not_fresh` operation after each assignment instruction or creation instruction. For each assignment instruction or creation instruction we have to use a fresh channel.

After the execution of the body, we have to evaluate the postcondition and we have to make sure that the locked request queues get unlocked at the right time. We combine the two steps into one operation `check_post_and_unlock_rqs` that takes the missing request queue locks $\overline{g}_{missing\_rq\_locks}$ and the feature f.

The operation evaluates the postcondition either synchronously or asynchronously. After the evaluation of the postcondition, the operation enqueues an `unlock` operation to each request queue in $\overline{g}_{missing\_rq\_locks}$.

SCOOP relies on the Eiffel invariant mechanism. This mechanism is described in Section 7.5 and Section 8.9.16 of the Eiffel ECMA standard [9]. On one hand, Section 7.5 describes the semantics of invariants: Invariants must be satisfied after the execution of every exported routine and after the execution of every creation procedure. On the other hand, Section 8.9.16 describes the runtime monitoring of invariants: Invariants get evaluated on both start and termination of a qualified call to a routine and after every call to a creation procedure. We had to decide whether to rely on the semantics of invariants or on the runtime monitoring of invariants. We decided to rely on the semantics of invariants for two reasons. First, the runtime invariant monitoring mechanism is only one possible implementation of the invariant semantics. Second, the runtime invariant monitoring mechanism relies on the notion of unqualified calls. However, for simplicity we assume feature calls to be in the canonical qualified form. The `apply` operation reflects our decision: We evaluate the invariant whenever f is exported. Note that the invariant can only contain non-separate target expressions. Hence, each call in the invariant will only require p's call stack lock.

Finally, we have to return the locks and we have to return the result if f is a function. For this purpose we introduce the `return` operation that comes in a variant for queries and in a variant for commands. Both variants take the channel a and the caller processor q in order to communicate with q. The variant for queries additionally takes the value to be returned to q.

Before we go on with the variants of the `apply` operation for non-fresh once routines and attributes, we discuss the operations that we did not discuss in details so far: the `check_pre_and_lock_rqs` operation, the `check_post_and_unlock_rqs` operation, and the `return` operation.

The `check_pre_and_lock_rqs`$(\{q_1, \dots, q_m\}, f)$ operation, executed by processor p, takes a processor set $\{q_1, \dots, q_m\}$ whose request queues must be locked on behalf of p and it takes a feature f whose precondition must be satisfied. The operation treats the precondition as a wait condition. It goes through a number of iterations. Each iteration obtains the request queue locks and then evaluates the precondition. If the precondition is satisfied then the `check_pre_and_lock_rqs` operation finishes. Otherwise it unlocks the request queues and then starts a new iteration. If the `check_pre_and_lock_rqs` operation finishes we can be assured that p obtained all the request queue locks and the precondition holds.

Check Precondition and Lock Request Queues Operation

$$\frac{a \textit{ is fresh}}{\Gamma \vdash \langle p :: \texttt{check\_pre\_and\_lock\_rqs}(\{q_1, \dots, q_m\}, f); s_p, \sigma\rangle \to}$$

```
⟨p :: lock({q_1, ..., q_m});
    provided $f.pre\_exists$ then
        eval($a, f.pre$);
        wait($a$)
    else
        nop
    end;
    provided $\neg f.pre\_exists \vee a.data$ then
        nop
    else
        issue($q_1$, unlock);
        ...
        issue($q_m$, unlock);
        pop_obtained_rq_locks;
        check_pre_and_lock_rqs({$q_1, \dots, q_m$}, $f$)
    end;
    $s_p, \sigma$⟩
```

The `check_post_and_unlock_rqs` operation also takes a processor set $\{q_1, \ldots, q_m\}$ and a feature f. The processor set is the same as the one that was given to the `check_pre_and_lock_rqs` operation, i.e. the set of processors whose request queues got locked in the synchronization step. The operation first determines whether the postcondition should be evaluated synchronously or asynchronously. Then the operation starts the evaluation. Finally, the operation enqueues an `unlock` operation to each request queue in $\{q_1, \ldots, q_m\}$.

Check Postcondition and Unlock Request Queues Operation

$$\overline{q} \overset{def}{=} \{q_1, \ldots, q_m\}$$
$$p \notin \overline{q}$$
$$targets(e) \overset{def}{=} \begin{cases} \{e_0\} \cup \bigcup_{i=0\ldots n} targets(e_i) & if\, e = e_0.w(e_1, \ldots, e_n) \\ \{\} & otherwise \end{cases}$$
$$args(e) \overset{def}{=} \begin{cases} \bigcup_{i=1\ldots n} \{(e_i, w, i)\} \cup args(e_i) & if\, e = e_0.w(e_1, \ldots, e_n) \\ \{\} & otherwise \end{cases}$$
$$g_0 \overset{def}{\in} \begin{cases} if \\ \quad \overline{q} \neq \{\} \wedge \\ \quad \forall x \in targets(f.post) \colon (\Gamma \vdash \sigma.handler(\sigma.val(p, controlling\_entity(x).name)) \in \overline{q}) \wedge \\ \quad \neg\exists(x, y, z) \in args(f.post), t, h, c \colon (\Gamma \vdash x : t \wedge is\_controlled(t) \wedge y.formals(z) : (!, h, c) \wedge c.is\_ref) \\ then \\ \quad \overline{q} \\ otherwise \\ \quad \{p\} \end{cases}$$
$$\{g_1, \ldots, g_j\} \overset{def}{=} \overline{q} \setminus g_0$$
$$a\ is\ fresh$$

```
$\Gamma \vdash \langle p ::$ check_post_and_unlock_rqs$(\{q_1, \ldots, q_m\}, f); s_p, \sigma\rangle \rightarrow$
  $\langle p ::$ provided $f.post\_exists \wedge g_0 \neq p$ then
      issue(
        $g_0$,
        execute_delegated(
            eval$(a, f.post)$; wait$(a)$;
            issue$(g_1,$ unlock$)$; $\ldots$; issue$(g_j,$ unlock$)$
          ,
            $\sigma.envs(p).top, \{q_1, \ldots, q_m\}$
        );
        unlock
      );
      pop_obtained_rq_locks
    else
      provided $f.post\_exists$ then
        eval$(a, f.post)$; wait$(a)$
      else
        nop
      end;
      issue$(q_1,$ unlock$)$; $\ldots$; issue$(q_m,$ unlock$)$;
      pop_obtained_rq_locks
    end;
    $s_p, \sigma\rangle$
```

Clarification 5 (Asynchronous postcondition evaluation) The postcondition can be evaluated asynchronously if every feature call in the postcondition only requires a request queue lock that was obtained in the synchronization step and if the postcondition does not involve lock passing. If the postcondition has a feature call that requires a lock different from the obtained request queue locks then p cannot delegate its obtained request queue lock and then continue because the required lock would be required in another feature execution context as well. Hence the postcondition must be evaluated synchronously in this case. If the postcondition involves lock passing then one of p's lock might be necessary for the evaluation of the postcondition. Hence, p must pass its locks and cannot proceed until the postcondition is evaluated and the passed locks returned. Once again, the postcondition must be evaluated synchronously. In Nienaltowski's description of SCOOP [22] a postcondition can be evaluated asynchronously if the current processor is not involved in the postcondition evaluation. This rule permits configurations in which the evaluating processor does not have the necessary locks for the evaluation. □

If the postcondition can be evaluated asynchronously then we can take one of the processors in $\{q_1, \ldots, q_m\}$. This set does not contain processor p because processor p never obtains its own request queue lock. Each processor in this set is exclusively available in the current execution context and can thus be used to evaluate the postcondition asynchronously. For the `check_post_and_unlock_rqs` operation we define g_0 to be the evaluating processor according to the rule just presented. We also define $\{g_1, \ldots, g_j\}$ to be the set $\{q_1, \ldots, q_m\}$ minus the request queue lock of g_0. If p is the evaluating processor, then this set is the same as $\{q_1, \ldots, q_m\}$. As a result of these definitions, the postcondition can be evaluated asynchronously if $g_0 \neq p$. Otherwise the postcondition must be evaluated synchronously.

In the synchronous case, processor p evaluates the postcondition, enqueues `unlock` operations to each request queue in $\{q_1, \ldots, q_m\}$, and then removes the corresponding locks from its stack of obtained request queue locks. The `unlock` operations won't proceed until the locks have been removed from p's stack of obtained request queue locks. In the asynchronous case, processor p must delegate the postcondition evaluation to processor g_0. For this purpose, p enqueues an `execute_delegated` operation to g_0. The workload involves the postcondition evaluation along with the subsequent issuing of `unlock` operations to all processor in $\{g_1, \ldots, g_j\}$. Processor g_0 unlocks its own request queue after the delegated execution. The evaluation of the postcondition on g_0 requires the environment that defines the values of the entities in the postcondition. Furthermore, the evaluation requires the request queue locks $\{q_1, \ldots, q_m\}$. These locks are sufficient because we only evaluate the postcondition asynchronously if the evaluation only requires these locks. To satisfy these two requirements, p gives its top environment and $\{q_1, \ldots, q_m\}$ to g_0. After g_0 performed the delegated execution, it can unlock its own request queue. In the meantime, processor p removes $\{q_1, \ldots, q_m\}$ from its obtained request queue locks to enable g_0 to proceed with the delegated execution.

The `return` operation comes in two variants: one for queries and one for commands.

Return Operation – Query

$$\frac{(\sigma', r') \overset{def}{=} \begin{cases} \textit{if}\, r \neq \textit{void} \wedge \sigma.\textit{ref\_obj}(r).\textit{class\_type}.\textit{is\_exp} \wedge \sigma.\textit{handler}(r) \neq q \\ \quad (\sigma_x, \sigma_x.\textit{last\_imported\_ref}) \\ \quad\quad \textbf{where} \\ \quad\quad \sigma_x \overset{def}{=} \sigma.\textit{deep\_import}(q, r) \\ \textit{otherwise} \\ \quad (\sigma, r) \end{cases} \\ \sigma'' \overset{def}{=} \sigma'.\textit{pop\_env}(p).\textit{revoke\_locks}(q, p)}{\Gamma \vdash \langle p :: \texttt{return}(a, r, q); s_p, \sigma\rangle \to \langle p :: \texttt{result}(a, r'); s_p, \sigma''\rangle}$$

Return Operation – Command

$$\frac{\sigma' \overset{def}{=} \sigma.\textit{pop\_env}(p).\textit{revoke\_locks}(q, p)}{\Gamma \vdash \langle p :: \texttt{return}(a, q); s_p, \sigma\rangle \to \\ \langle p :: \texttt{provided}\ \sigma.\textit{are\_locks\_passed}(q)\ \texttt{then}\ \texttt{notify}(a)\ \texttt{else}\ \texttt{nop}\ \texttt{end}; s_p, \sigma'\rangle}$$

The variant for queries returns the result and the locks. The variant for commands only returns the locks. Both variants take a channel a and the caller processor q. For queries, the channel is used to return the result. For this purpose, the operation takes a reference r that points to the result. Processor q is waiting for this result on channel a. This can be seen in the `call` operation, which issues an `apply` operation and a subsequent `wait`(a). The `apply` operation calls the `return` operation with the same channel a. To return the result to q, processor p executes a `result` on a. The value to be returned is not always r directly. If r points to an object of expanded class type and $q \neq p$ then q must deep import the object. In all other cases, q can take r as the return value. An explanation why the deep import operation is necessary can be found in Section 4.6.4. For commands, the channel is used to signal to q that the locks have been returned in case q passed its locks. This can be determined by looking at the state: $\sigma.are\_locks\_passed(q)$. In both variants of the `return` operation, p removes the passed locks from the stacks of retrieved locks. In case q did not pass any locks, the removed entries might be the empty set. Processor p also removes its top environment because this environment is no longer needed. In case of an asynchronous postcondition evaluation, this environment temporarily gets delegated to the evaluating processor.

So far we have looked at the `apply` variant for non-once routines and fresh once routines. We left out the discussion of the non-fresh once routines and the attributes. In the following, we make up for this. Non-fresh once functions already have a result. We just need to get this result from the state and return it. For non-fresh once procedures we do not even have to do this. The only obligation we have in both cases is the invariant evaluation. The evaluation of the invariant requires the call stack lock of p. This lock is given if the condition $\neg\sigma.are\_locks\_passed(p)$ holds.

Application Operation – Non-Fresh Once Routine

$$\frac{\begin{array}{l} f \in \textbf{ROUTINE} \wedge f.is\_once \wedge \neg\sigma.is\_fresh(p, f) \\ \sigma.handler(r_0) = p \\ \neg\sigma.are\_locks\_passed(p) \\ \sigma' \overset{def}{=} \sigma.pass\_locks(q, p, \bar{l}).push\_env\_with\_feature(p, f, r_0, (r_1, \ldots, r_n)) \\ a\ is\ fresh \end{array}}{\Gamma \vdash \langle p :: \texttt{apply}(a, r_0, f, (r_1, \ldots, r_n), q, \bar{l}); s_p, \sigma\rangle \to}$$

```
⟨p :: provided $f.class\_type.inv\_exists \wedge f.is\_exported$ then
         eval($a, f.class\_type.inv$); wait($a$)
     else
         nop
     end;
     provided $f \in$ FUNCTION then
         return($a, \sigma'.once\_result(p, f), q$)
     else
         return($a, q$)
     end;
     $s_p, \sigma'$⟩
```

Application Operation – Attribute

$$\frac{\begin{array}{l} f \in \textbf{ATTRIBUTE} \\ \sigma.handler(r_0) = p \\ \neg\sigma.are\_locks\_passed(p) \\ \sigma' \overset{def}{=} \sigma.pass\_locks(q, p, \bar{l}).push\_env\_with\_feature(p, f, r_0, ()) \\ a' \ is \ fresh \end{array}}{\begin{array}{l} \Gamma \vdash \langle p :: \texttt{apply}(a, r_0, f, (), q, \bar{l}); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{eval}(a', f); \\ \quad\quad \texttt{wait}(a'); \\ \quad\quad \texttt{return}(a, a'.data, q); \\ \quad\quad s_p, \sigma'\rangle \end{array}}$$

Example 8 (Feature application) In this example, we look at the application of the feature *buy* on the first investor. Listing 2 shows this feature as part of the class *INVESTOR*.

Listing 2: Investor class

```
class INVESTOR

create
  make

feature -- Initialization
  make (a_id: INTEGER)
      -- Create an investor with identifier 'a_id'.
    do
      id := a_id
    end

feature -- Access
  id: INTEGER
    -- The identifier.

  log: separate UUID
    -- The identifier of the last market.

  buy (a_market: separate MARKET; a_issuer_id: INTEGER)
      -- Buy a share of the issuer with identifier 'a_issuer_id' on the market 'a_market'.
    require
      a_market.can_buy (Current.id, a_issuer_id)
    do
      a_market.buy (Current, a_issuer_id)
      log := a_market.id
    ensure
      a_market.can_sell (Current.id, a_issuer_id)
    end

    ...
end
```

We start with a configuration where processor p_1 finished executing the feature calls in feature *do_transaction*.

These feature calls led to one `apply` operation for the *buy* feature and one for the *sell* feature in the action queue of each investor's handler.

$\langle$

$p_1 :: \ \dots \ |$

$p_2 :: \ |$

$p_3 :: \texttt{apply}(a_{62}, r_6, \mathit{buy}, (r_1, r_{39}), p_1, (\{\}, \{\}));$
$\quad \texttt{apply}(a_{70}, r_6, \mathit{sell}, (r_1, r_{39}), p_1, (\{\}, \{\})) \ |$

$p_4 :: \texttt{apply}(a_{66}, r_8, \mathit{buy}, (r_1, r_{39}), p_1, (\{\}, \{\}));$
$\quad \texttt{apply}(a_{74}, r_8, \mathit{sell}, (r_1, r_{39}), p_1, (\{\}, \{\}))$

,

locks:

$p_1 ::$ orq: $(\{\}, \{p_3, p_4\})$ rrq: $(\{\}, \{\})$ rcs: $(\{\}, \{\})$ locked

$p_2 ::$ orq: $()$ rrq: $()$ rcs: $()$ unlocked

$p_3 ::$ orq: $()$ rrq: $()$ rcs: $()$ locked

$p_4 ::$ orq: $()$ rrq: $()$ rcs: $()$ locked

objects:

$p_1 :: \ r_0 \to o_0(\mathit{market} \to r_1), r_{39} \to o_{48}(1)$

$p_2 :: r_1 \to o_{35}(\mathit{cash} \to r_{16}, \mathit{available\_shares} \to r_{23}, \mathit{owned\_shares} \to r_{29}),$
$\quad r_{16} \to o_{25}[r_{21}, r_{22}], r_{21} \to o_{23}(100), r_{22} \to o_{24}(100),$
$\quad r_{23} \to o_{33}[r_{28}], r_{28} \to o_{32}(1),$
$\quad r_{29} \to o_{38}[[r_{34}], [r_{35}]], r_{34} \to o_{41}(0), r_{35} \to o_{42}(0)$

$p_3 :: \ r_6 \to o_{44}(\mathit{id} \to r_{36}), r_{36} \to o_{43}(1)$

$p_4 :: \ r_8 \to o_{46}(\mathit{id} \to r_{37}), r_{37} \to o_{45}(2)$

once status:

environments:

$p_1 :: \mathit{l\_first\_investor} \to r_6, \mathit{l\_second\_investor} \to r_8, \textbf{Current} \to r_0 \ /$
$\quad \mathit{a\_first\_investor} \to r_6, \mathit{a\_second\_investor} \to r_8, \mathit{a\_issuer\_id} \to r_{39}, \textbf{Current} \to r_0$

$p_2 ::$

$p_3 ::$

$p_4 ::$

$\rangle$

At this point, processor p_3 and p_4 can each take the transition that is described by the inference rule for the `apply` operation for non-once routines. Each processor can then take an additional transition according to the inference rule for the `check_pre_and_lock_rqs` operation. The result configuration is shown below. The channel a_{75} is a fresh channel. Both processors added a new environment that maps the expanded formal argument to a copy of the expanded actual argument. In case of processor p_3, the copied object is referenced by r_{40}. On processor p_4, the copied object is referenced by r_{41}. Since processor p_1 did not pass its locks, both p_3 and p_4 added empty lock sets to their stack of retrieved locks.

```
⟨
  $p_1$ :: … |
  $p_2$ :: |
  $p_3$ :: lock({$p_2$});
       eval($a_{75}$, $a\_market.can\_buy(current.id, a\_issuer\_id)$);
       wait($a_{75}$);
       provided $a_{75}.data$ then
         nop
       else
         issue($p_2$, unlock);
         pop_obtained_rq_locks;
         check_pre_and_lock_rqs({$p_2$}, $buy$)
       end;
       $a\_market.buy(current, a\_issuer\_id)$;
       $log := a\_market.id$;
       check_post_and_unlock_rqs({$p_2$}, $buy$);
       return($a_{62}, p_1$);
       apply($a_{70}, r_6, sell, (r_1, r_{39}), p_1, (\{\}, \{\})$) |
  $p_4$ :: lock({$p_2$});
       …
,
  locks:
    $p_1$ :: orq: $(\{\}, \{p_3, p_4\})$ rrq: $(\{\}, \{\})$ rcs: $(\{\}, \{\})$ locked
    $p_2$ :: orq: $()$ rrq: $()$ rcs: $()$ unlocked
    $p_3$ :: orq: $()$ rrq: $(\{\})$ rcs: $(\{\})$ locked
    $p_4$ :: orq: $()$ rrq: $(\{\})$ rcs: $(\{\})$ locked
  objects:
    $p_1$ :: $r_0 \to o_0(market \to r_1), r_{39} \to o_{48}(1)$
    $p_2$ :: $r_1 \to o_{35}(cash \to r_{16}, available\_shares \to r_{23}, owned\_shares \to r_{29})$,
           $r_{16} \to o_{25}[r_{21}, r_{22}], r_{21} \to o_{23}(100), r_{22} \to o_{24}(100)$,
           $r_{23} \to o_{33}[r_{28}], r_{28} \to o_{32}(1)$,
           $r_{29} \to o_{38}[[r_{34}], [r_{35}]], r_{34} \to o_{41}(0), r_{35} \to o_{42}(0)$
    $p_3$ :: $r_6 \to o_{44}(id \to r_{36}), r_{36} \to o_{43}(1), r_{40} \to o_{49}(1)$
    $p_4$ :: $r_8 \to o_{46}(id \to r_{37}), r_{37} \to o_{45}(2), r_{41} \to o_{50}(1)$
  once status:
  environments:
    $p_1$ :: $l\_first\_investor \to r_6, l\_second\_investor \to r_8$, Current $\to r_0$ /
           $a\_first\_investor \to r_6, a\_second\_investor \to r_8, a\_issuer\_id \to r_{39}$, Current $\to r_0$
    $p_2$ ::
    $p_3$ :: $a\_market \to r_1, a\_issuer\_id \to r_{40}$, Current $\to r_6$
    $p_4$ :: $a\_market \to r_1, a\_issuer\_id \to r_{41}$, Current $\to r_8$
⟩
```

The inference rule for the `lock` operation shows that both p_3 and p_4 require the request queue lock of p_2. We decide to give priority to p_3. This leads to the following configuration, where p_2's request queue is locked on behalf of p_3:

```
⟨
  p_1 :: ... |
  p_2 :: |
  p_3 :: eval(a_75, a_market.can_buy(current.id, a_issuer_id));
       wait(a_75);
       provided a_75.data then
         nop
       else
         issue(p_2, unlock);
         pop_obtained_rq_locks;
         check_pre_and_lock_rqs({p_2}, buy)
       end;
       a_market.buy(current, a_issuer_id);
       log := a_market.id;
       check_post_and_unlock_rqs({p_2}, buy);
       return(a_62, p_1);
       apply(a_70, r_6, sell, (r_1, r_39), p_1, ({}, {})) |
   p_4 :: ...
,
  locks:
    p_1 :: orq: ({}, {p_3, p_4}) rrq: ({}, {}) rcs: ({}, {}) locked
    p_2 :: orq: () rrq: () rcs: () locked
    p_3 :: orq: ({p_2}) rrq: ({}) rcs: ({}) locked
    p_4 :: orq: () rrq: ({}) rcs: ({}) locked
  objects:
    ...
  once status:
  environments:
    ...
⟩
```

The evaluation of the precondition produces a result on p_2 that is awaited by p_3. The result is a deep imported object of class type *BOOLEAN* that is referenced by r_{58}. The boolean value of this object indicates that the precondition is satisfied and hence p_3 can continue with the execution of the body. In the following resulting configuration, the arrays referenced by r_{16}, r_{23}, and r_{29} have been updated; The first investor bought a share of the issuer. Consequently, the first investor has a lower amount of cash and there is one fewer share available. Furthermore, the *log* attribute of the first investor object has been updated with the identifier of the market. This identifier object is referenced by r_{65}. It has been computed by p_2 in a once function of separate type. Hence the object is available as a once result on all processors in the system.

$\langle$

$p_1 :: \ \ldots \ |$

$p_2 :: \ |$

$p_3 :: \texttt{check\_post\_and\_unlock\_rqs}(\{p_2\}, buy);$

$\quad \texttt{return}(a_{62}, p_1);$

$\quad \texttt{apply}(a_{70}, r_6, sell, (r_1, r_{39}), p_1, (\{\}, \{\})) \ |$

$p_4 :: \ \ldots$

,

locks:

$\ldots$

objects:

$p_1 :: \ r_0 \rightarrow o_0(market \rightarrow r_1), r_{39} \rightarrow o_{48}(1)$

$p_2 :: r_1 \rightarrow o_{35}(cash \rightarrow r_{16}, available\_shares \rightarrow r_{23}, owned\_shares \rightarrow r_{29}),$

$\quad r_{16} \rightarrow o_{71}[r_{61}, r_{22}], r_{61} \rightarrow o_{70}(90), r_{22} \rightarrow o_{24}(100),$

$\quad r_{23} \rightarrow o_{73}[r_{62}], r_{62} \rightarrow o_{72}(0),$

$\quad r_{29} \rightarrow o_{75}[[r_{63}], [r_{35}]], r_{63} \rightarrow o_{74}(1), r_{35} \rightarrow o_{42}(0),$

$\quad r_{65} \rightarrow o_{77}$

$p_3 :: \ r_6 \rightarrow o_{78}(id \rightarrow r_{36}, log \rightarrow r_{65}), r_{36} \rightarrow o_{43}(1), r_{40} \rightarrow o_{49}(1), r_{58} \rightarrow o_{67}(true)$

$p_4 :: \ r_8 \rightarrow o_{46}(id \rightarrow r_{37}), r_{37} \rightarrow o_{45}(2), r_{41} \rightarrow o_{50}(1)$

once status:

all :: $\{MARKET\}.id \rightarrow r_{65}$

environments:

$\ldots$

$\rangle$

The postcondition of the feature *buy* contains the expression *current.id*. The controlling entity of the target *current* is the current entity. The handler of the current object is p_3 itself. Processor p_3 did not obtain its own request queue lock. Hence the postcondition must be evaluated synchronously. After the postcondition evaluation, p_3 enqueues the `unlock` operation to p_2 and then removes p_2's request queue lock from its stack of obtained request queue locks. This enables p_2 to unlock its request queue lock. Processor p_3 then executes the `return` operation. This leads to the following configuration, where p_3's top environment and retrieved locks are removed:

$\langle$

$p_1 :: \ \ldots \ |$

$p_2 :: \ |$

$p_3 :: \texttt{apply}(a_{70}, r_6, sell, (r_1, r_{39}), p_1, (\{\}, \{\})) \ |$

$p_4 :: \ \ldots$

,

locks:

$p_1 ::$ orq: $(\{\}, \{p_3, p_4\})$ rrq: $(\{\}, \{\})$ rcs: $(\{\}, \{\})$ locked

$p_2 ::$ orq: $()$ rrq: $()$ rcs: $()$ unlocked

$p_3 ::$ orq: $()$ rrq: $()$ rcs: $()$ locked

$p_4 ::$ orq: $()$ rrq: $(\{\})$ rcs: $(\{\})$ locked

objects:

...

once status:

...

environments:

$p_1 :: l\_first\_investor \rightarrow r_6, l\_second\_investor \rightarrow r_8,$ **Current** $\rightarrow r_0$ /
$a\_first\_investor \rightarrow r_6, a\_second\_investor \rightarrow r_8, a\_issuer\_id \rightarrow r_{39},$ **Current** $\rightarrow r_0$

$p_2 ::$

$p_3 ::$

$p_4 :: a\_market \rightarrow r_1, a\_issuer\_id \rightarrow r_{41},$ **Current** $\rightarrow r_8$

$\rangle$

□

5.4.5. Creation instructions

A creation instruction has the form **create** $b.f(e_1, \ldots, e_n)$ where b is the target entity, f is the creation procedure, and $e_1, \ldots, e_n$ are the actual arguments. We assume that b is of type (d, g, c). A processor p that executes this instruction takes the following steps:

1. Processor q creation:
   - If b is separate, i.e. $g = \top$, then create a new processor.
   - If b has an explicit processor specification, i.e. $g = \alpha$, then
     - take the processor denoted by α, if it already exists.
     - create a new processor, if the processor denoted by α does not exist yet.
   - If b is non-separate, i.e. $g = \bullet$, then take p.
2. Locking: Lock the request queue of q if the following conditions hold:
   - Processor p and processor q are different.
   - Processor p does not have q's request queue lock.
   - Processor q does not have p's request queue lock.
3. Object creation: Ask q to create a new instance with class type c using the creation procedure f. Attach the newly created object to b.
4. Invariant evaluation: If f is not exported then ask q to evaluate the invariant.

5. Lock releasing: If q's request queue has been locked in the locking step, then ask q to unlock its request queue after it is done with the feature request.

   There are four cases in the processor creation step:

- The entity b has a separate type.
- The entity b has an explicit processor specification and the denoted processor already exists.
- The entity b has an explicit processor specification and the denoted processor does not yet exist.
- The entity b has a non-separate type.

For each of these cases we introduce one inference rule. We start with the variant where b has a separate type. In this case we define q as a new processor and o as a new object of class type c. The reference r points to the this object. First we acquire a request queue lock on the new processor q so that we can issue statements on q. Next, we write the value r into the entity b. To make a call to the creation procedure, we execute a command instruction. Once this is done, we must check whether we have to evaluate the invariant. If f is exported then the invariant will be evaluated as part of f's feature application. In this case we must do nothing. However, if f is not exported then we must issue the invariant evaluation to q. After this step, we can issue an `unlock` operation to q and remove the request queue lock from p's obtained request queue locks.

Create Instruction – Top

$$\frac{\begin{array}{l}(d,h,c) \overset{def}{=} type\_of(\Gamma, b)\\ h = \top\\ q \overset{def}{=} \sigma.new\_proc\\ o \overset{def}{=} \sigma.new\_obj(c)\\ \sigma' \overset{def}{=} \sigma.add\_proc(q).add\_obj(q,o)\\ r \overset{def}{=} \sigma'.ref(o)\\ a\ is\ fresh\end{array}}{\Gamma \vdash \langle p :: \mathbf{create}\ b.f(e_1,\ldots,e_n); s_p, \sigma\rangle \rightarrow}$$

```
⟨$p$ :: lock($\{q\}$);
    write($b.name, r$);
    $b.f(e_1,\ldots,e_n)$;
    provided $\neg f.class\_type.inv\_exists \vee f.is\_exported$ then
        nop
    else
        issue($q$, eval($a, f.class\_type.inv$); wait($a$))
    end;
    issue($q$, unlock);
    pop_obtained_rq_locks;
    $s_p \mid q$ :: nop, $\sigma'$⟩
```

In the following, we look at the two variants for the cases where b has an explicit processor specification. There are two forms of explicit processor specifications: unqualified and qualified. An unqualified explicit processor specification is based on a processor attribute x with an attached type. We write the processor specification as $< x >$. The processor denoted by this explicit processor specification is the processor stored in x. A qualified explicit processor specification is based on an entity y. The entity y is a non-writable entity of attached type. We write this processor specification as $< y.handler >$. The processor denoted by this explicit processor specification is the same processor as the one handling the object referenced by y. We notice that a qualified explicit processor specification always denotes an existing processor because this specification is based on an attached entity. This means that there is already an object attached to this entity and thus its handler must exist. This insight helps us to write the conditions for the two inference rule variants.

Create Instruction – Existing Explicit Processor

$$(d,h,c) \overset{def}{=} type\_of(\Gamma,b)$$
$$h =< x > \lor h =< y.handler >$$
$$q \overset{def}{=} \begin{cases} \sigma.val(p,x) & if\, t = (d,< x >,c) \\ \sigma.handler(\sigma.val(p,y)) & if\, t = (d,< y.handler >,c) \end{cases}$$
$$\sigma.procs.has(q)$$
$$\overline{g}_{required\_cs\_locks} \overset{def}{=} \begin{cases} \{q\} & if\, q \neq p \land (\sigma.rq\_locks(q).has(p) \lor \sigma.cs\_locks(q).has(p)) \\ \{\} & otherwise \end{cases}$$
$$\forall x \in \overline{g}_{required\_cs\_locks} : \neg\sigma.are\_locks\_passed(p) \land \sigma.cs\_locks(p).has(x)$$
$$o \overset{def}{=} \sigma.new\_obj(c)$$
$$\sigma' \overset{def}{=} \sigma.add\_obj(q,o)$$
$$r \overset{def}{=} \sigma'.ref(o)$$
$$a\ is\ fresh$$

```
$\Gamma \vdash \langle p ::$ create $b.f(e_1,\ldots,e_n); s_p, \sigma\rangle \rightarrow$
    $\langle p ::$ provided $q \neq p \land \neg\sigma'.rq\_locks(p).has(q) \land \neg\sigma'.rq\_locks(q).has(p)$ then
            lock($\{q\}$)
        else
            nop
        end;
        write($b.name, r$);
        $b.f(e_1,\ldots,e_n)$;
        provided $\neg f.class\_type.inv\_exists \lor f.is\_exported$ then
            nop
        else
            issue($q$, eval($a, f.class\_type.inv$); wait($a$))
        end;
        provided $q \neq p \land \neg\sigma'.rq\_locks(p).has(q) \land \neg\sigma'.rq\_locks(q).has(p)$ then
            issue($q$, unlock);
            pop_obtained_rq_locks
        else
            nop
        end;
        $s_p, \sigma'\rangle$
```

For the variant that handles existing processors, we have to determine the denoted processor for both the qualified and the unqualified possibility. For the qualified option, we can simply lookup the value of the attribute x. For the unqualified option, we must first lookup the value of the entity y and then determine the handler of the referenced object. In either case, the result q is either the denoted processor, if it exists, or the void value. We can then check whether q is in the set of processors of our system. We can only use this inference rule, if q is in this processor set. The overall idea of this inference rule is the same as in the case where b has a separate type. The difference lies in the processor creation, locking, and lock releasing steps. Instead of creating a new processor we can take the existing processor q. If $q = p$ then the call to the creation procedure will be a non-separate call. In this case, p's call stack lock is required. This lock is given because otherwise p would be waiting. If $p \neq q$ and q has a lock on p then the call to the creation procedure will be a separate callback. In this case, p requires q's call stack lock. This is expressed in the condition with the help of the set $\overline{g}_{required\_cs\_locks}$. If $p \neq q$ and q does not have p's request queue lock then the call to the creation procedure will be a separate call. In this case, p must obtain q's request queue lock, provided p does not already have this lock. Only when p obtained q's request queue lock, do we have to issue an `unlock` operation and remove q from p's stack of obtained request queue locks.

For the variant that handles non-existing processors, we have to determine the denoted processor only for unqualified processor specifications. The condition states that the denoted processor must not yet exist. If the condition is satisfied then we create a new processor q with a new object o and reference r. The steps in this variant are similar to those we saw in the variant where b has a separate type. However, we have to set the value of the processor attribute x to the newly created processor. This ensures that the denoted processor will be found to exist in the future.

Create Instruction – Non-Existing Explicit Processor

$$\frac{\begin{array}{l}(d,h,c) \overset{def}{=} type\_of(\Gamma, b)\\ h =< x >\\ \neg\sigma.procs.has(\sigma.val(p,x))\\ q \overset{def}{=} \sigma.new\_proc\\ o \overset{def}{=} \sigma.new\_obj(c)\\ \sigma' \overset{def}{=} \sigma.add\_proc(q).add\_obj(q,o)\\ r \overset{def}{=} \sigma'.ref(o)\\ a\ is\ fresh\end{array}}{\Gamma \vdash \langle p :: \mathbf{create}\ b.f(e_1,\ldots,e_n); s_p, \sigma\rangle \rightarrow}$$

```
⟨p :: write(x.name, q);
     lock({q});
     write(b.name, r);
     b.f(e_1, ..., e_n);
     provided ¬f.class_type.inv_exists ∨ f.is_exported then
        nop
     else
        issue(q, eval(a, f.class_type.inv); wait(a))
     end;
     issue(q, unlock);
     pop_obtained_rq_locks;
     s_p | q :: nop, σ'⟩
```

Lastly, we have the variant for the case where b has a non-separate type. In this case, we create the object on the current processor. Processor creation, locking, and lock releasing is not necessary. The required call stack lock on p is given because otherwise p would be waiting.

Create Instruction – Non-Separate

$$\frac{\begin{array}{l}(d, h, c) \overset{def}{=} \mathit{type\_of}(\Gamma, b) \\ h = \bullet \\ o \overset{def}{=} \sigma.\mathit{new\_obj}(c) \\ \sigma' \overset{def}{=} \sigma.\mathit{add\_obj}(p, o) \\ r \overset{def}{=} \sigma'.\mathit{ref}(o) \\ a\ is\ fresh\end{array}}{\begin{array}{l}\Gamma \vdash \langle p :: \mathbf{create}\ b.f(e_1, \ldots, e_n); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \mathtt{write}(b.\mathit{name}, r); \\ \qquad b.f(e_1, \ldots, e_n); \\ \qquad \mathtt{provided}\ \neg f.\mathit{class\_type}.\mathit{inv\_exists} \vee f.\mathit{is\_exported}\ \mathtt{then} \\ \qquad\quad \mathtt{nop} \\ \qquad \mathtt{else} \\ \qquad\quad \mathtt{eval}(a, f.\mathit{class\_type}.\mathit{inv}); \mathtt{wait}(a) \\ \qquad \mathtt{end}; \\ \qquad s_p, \sigma'\rangle\end{array}}$$

Example 9 (Object creation) To illustrate object creation, we start in a configuration where the root processor p_1 is executing the root procedure *make* on the root object o_0. This procedure is shown in listing 3.

Listing 3: Application class with initialization

```
class APPLICATION

create
  make

feature -- Initialization
  make
      -- Create a market with investors and issuers. Then do some transactions.
    local
      l_first_investor: separate INVESTOR
      l_second_investor: separate INVESTOR
    do
      -- Create the market with 2 investors each of which has 10 units of cash and with 1 issuer that has
          1 share. Create the investors with identifiers 1 and 2.
      create market.make (2, 100, 1, 1)
      create l_first_investor.make (1)
      create l_second_investor.make (2)

      -- Do a transaction.
      Current.do_transaction (l_first_investor, l_second_investor, 1)
    end

feature {APPLICATION} -- Implementation
  market: separate MARKET
    -- The market.

  do_transaction
```

```
    ...
end
```

The following configuration is our starting point:

$\langle$

p_1 :: **create** $market.make(2, 100, 1, 1)$;
        $\dots$

,

locks:

p_1 :: orq: ({}) rrq: ({}) rcs: ({}) locked

objects:

p_1 :: $r_0 \rightarrow o_0(market \rightarrow void)$

once status:

environments:

p_1 :: $l\_first\_investor \rightarrow void, l\_second\_investor \rightarrow void,$ **Current** $\rightarrow r_0$

$\rangle$

Processor p_1 can now take the transition as described by the inference rule for the top variant. The result is a new configuration, in which o_1 is the new market object handled by a new processor p_2:

$\langle$

p_1 :: `lock`$(\{p_2\})$;
        `write`$(market.name, r_1)$;
        $market.make(2, 100, 1, 1)$;
        `issue`$(p_2,$ `unlock`$)$;
        `pop_obtained_rq_locks`;
        $\dots$ | p_2 :: `nop`

,

locks:

p_1 :: orq: ({}) rrq: ({}) rcs: ({}) locked

p_2 :: orq: () rrq: () rcs: () unlocked

objects:

p_1 :: $r_0 \rightarrow o_0(market \rightarrow void)$

p_2 :: $r_1 \rightarrow o_1$

once status:

environments:

p_1 :: $l\_first\_investor \rightarrow void, l\_second\_investor \rightarrow void,$ **Current** $\rightarrow r_0$

p_2 ::

$\rangle$

Now processor p_1 locks the request queue of processor p_2. It then stores the reference r_1 into the entity *market*. With these two steps, processor p_1 set up the context to execute a feature call to the creation procedure *make*. The resulting feature request will be executed by processor p_2. Processor p_1 then asks processor p_2 to unlock its request queue after it is done with the feature request. Then processor p_1 removes the obtained request queue lock from its stack of obtained request queue locks. □

5.4.6. Flow control instructions

The **if** e **then** s_t **else** s_f **end** instruction executes s_t if the expression e evaluates to true. Otherwise the instruction executes s_f. We introduce one inference rule for this instruction. In a first step, we evaluate the expression e using a fresh channel a and then we wait for a notification on a. In a second step, we use the `provided` operator to either execute s_t or s_f, depending on the value of the expression.

If Instruction

$$\frac{a\ is\ fresh}{\Gamma \vdash \langle p :: \textbf{if}\ e\ \textbf{then}\ s_t\ \textbf{else}\ s_f\ \textbf{end}; s_p, \sigma\rangle \rightarrow}$$

```
⟨p :: eval(a, e);
      wait(a);
      provided a.data then
         $s_t$
      else
         $s_f$
      end;
      $s_p, \sigma$⟩
```

The **until** e **loop** s_t **end** operation executes a sequence of s_l instructions until the expression e evaluates to true. If e is true initially then the executed sequence is empty. We introduce one inference rule for this instruction. First, we evaluate e using a fresh channel a. Then we wait for a notification on a. Next, we use the `provided` operation to check whether e evaluates to true or false. If e is true then we are done. Otherwise, we execute s_l followed by another **until** e **loop** s_t **end** operation.

Loop Instruction

$$\frac{a\ is\ fresh}{\Gamma \vdash \langle p :: \textbf{until}\ e\ \textbf{loop}\ s_l\ \textbf{end}; s_p, \sigma\rangle \rightarrow}$$

```
⟨p :: eval(a, e);
      wait(a);
      provided a.data then
         nop
      else
         $s_l$; until $e$ loop $s_l$ end
      end;
      $s_p, \sigma$⟩
```

5.4.7. Assignment instructions

An assignment instruction $b := e$ assigns the value of the expression e to the entity b. The following inference rule implements the assignment instruction. We first evaluate the expression e and then wait for a notification on a fresh channel a. Once we get this notification, we use the `write` operation to set the value to the entity b.

Assignment

$$\frac{a\ is\ fresh}{\Gamma \vdash \langle p :: b := e; s_p, \sigma\rangle \rightarrow \langle p :: \texttt{eval}(a, e); \texttt{wait}(a); \texttt{write}(b.name, a.data); s_p, \sigma\rangle}$$

5.5. Termination

The system terminates when we reach a configuration where all action queues are empty, i.e. when there is no more work to do.

6. Conclusion

In this paper we have presented a formal specification of the SCOOP model, based on operational semantics. We have demonstrated that this level of rigor is necessary if the specification is to be used as a guideline for an implementation. In particular, we were able to clarify a number of omissions and ambiguities in the available informal specification, which had gone undetected in other formalizations:

- Are processor locks fine-grained enough? We require request queue locks and call stack locks.
- Which locks must be passed? Which locks can be passed? We pass all the locks we actually have. We pass these locks both for normal lock passing and for separate callbacks.
- How do we move object structures from one processor to another processor without violating the invariant? The deep import operation must be used.
- When do we set the status of a fresh once routine to non-fresh? The status of the once routine must be set to non-fresh before deep importing.
- When can a postcondition be evaluated asynchronously? The postcondition can be evaluated asynchronously if every feature call in the postcondition only requires a lock that was obtained in the synchronization step and if the postcondition does not involve lock passing.

Because of the complexity of the SCOOP model, our resulting specification is large, the management of which is a challenge for a fully formal development. To address this problem, we used abstract data types and a notation with an object-oriented flavor, which made the specification more readable and more easily extendable, without sacrificing any of the rigor of operational semantics. Furthermore, we introduced a distinction between two kinds of statements, namely instructions (user syntax) and operations (run-time syntax). This made it possible to treat within one inference system both the actual language elements and the implementation details of the runtime system, and to distinguish clearly between them.

The main application of this work is as a formal reference document to guide the implementation of the SCOOP model. This has led to a successful implementation of SCOOP on top of the Eiffel language, which supersedes the previous prototype implementation and is publicly available [29]. The SCOOP model can however be implemented on top of any object-oriented language (support for contracts, as offered by Java or Spec#, is beneficial), and our work also facilitates such future implementation efforts. In the case of Java, first steps towards such an implementation have been taken [30], which could certainly be supported by our work.

A number of other applications of our semantics can be envisioned. First, the semantics can be used to prove correct various properties of the model which have so far only been postulated, such as absence of object-level data races and type safety (absence of traitors). In light of the complexity of the full model, these properties are no longer obvious. For example, as processor locks serve as an abstraction only, it must be shown that locks are not misused in situations such as separate callbacks, which involve call stack locks. Second, our operational semantics can also be used to prove correct an axiomatic semantics for the SCOOP model, which is planned for future work. In the case of sequential Eiffel, a similar development is documented in [24].

Third, we feel our semantics is detailed enough that its rules can directly be implemented as an interpreter for SCOOP programs. Such an interpreter could serve as a true reference implementation, which could in turn be used for conformance checking of real implementations.

Acknowledgments. This work is part of the SCOOP project at ETH Zurich, which has benefited from grants from the Hasler Foundation, the Swiss National Foundation, Microsoft (Multicore award) and ETH (ETHIIRA).

References

[1] E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. A compositional operational semantics for JavaMT. *Lecture Notes in Computer Science*, 2772:290–303, 2003.

[2] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, and G. L. S. Jr. Object-oriented units of measurement. In *Conference on Object Oriented Programming Systems Languages and Applications*, pages 384–403, 2004.

[3] Axum website. http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx/, 2010.

[4] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems*, 26(5):269–804, 2004.

[5] P. J. Brooke, R. F. Paige, and J. L. Jacob. A CSP model of Eiffel's SCOOP. *Formal Aspects of Computing*, 19(4):487–512, 2007.

[6] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. *Lecture Notes in Computer Science*, 1523:157–200, 1999.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Conference on Object Oriented Programming Systems Languages and Applications*, pages 519–538, 2005.

[8] E. G. Coffman, M. J. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

[9] ECMA. ECMA-367 Eiffel: Analysis, design and programming language 2nd edition. Technical report, ECMA International, 2006.

[10] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, 1996.

[11] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[12] M. Joyner, B. L. Chamberlain, and S. J. Deitz. Iterators in chapel. In *International Parallel and Distributed Processing Symposium/International Parallel Processing Symposium*, 2006.

[13] S. Khoshafian and G. P. Copeland. Object identity. In *Conference on Object Oriented Programming Systems Languages and Applications*, pages 406–416, 1986.

[14] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[15] B. Liskov and S. Zilles. Programming with abstract data types. *ACM SIGPLAN Notices*, 9(4):50–59, 1974.

[16] A. Lochbihler. Type safe nondeterminism – A formal semantics of Java threads. In *International Workshop on Foundations of Object-Oriented Languages*, 2008.

[17] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming.* MIT Press, Cambridge (Mass.), USA, 1993.

[18] B. Meyer. A three-level approach to data structure description, and notational framework. In *ACM-NBS Workshop on Data Abstraction, Databases and Conceptual Modelling*, pages 164–166, 1981.

[19] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 2nd edition, 1997.

[20] R. Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, 1999.

[21] B. Morandi, S. S. Bauer, and B. Meyer. SCOOP – A contract-based concurrent object-oriented programming model. *Lecture Notes in Computer Science*, 6029:41–90, 2010.

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

[23] T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle/HOL: A proof assistant for higher-order logic. *Lecture Notes in Computer Science*, 2283, 2002.

[24] M. Nordio, C. Calcagno, P. Müller, and B. Meyer. A sound and complete program logic for Eiffel. In M. Oriol and B. Meyer, editors, *TOOLS-EUROPE*, volume 33 of *Lecture Notes in Business and Information Processing*, 2009.

[25] J. S. Ostroff, F. A. Torshizi, H. F. Huang, and B. Schoeller. Beyond contracts for concurrency. *Formal Aspects of Computing*, 21(4):319–346, 2008.

[26] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[27] G. D. Plotkin. A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

[28] H. W. Schmidt and J. Chen. Reasoning about concurrent objects. *Asia-Pacific Software Engineering Conference*, 0:86, 1995.

[29] SCOOP website. http://scoop.origo.ethz.ch/, 2010.

[30] F. Torshizi, J. S. Ostroff, R. F. Paige, and M. Chechik. The SCOOP concurrency model in Java-like languages. In *Communicating Process Architectures*, pages 155–178. IOS, 2009.