Type Safety for Distributed Concurrent Objects and Runtime Upgrades

Doctoral Dissertation by

Ingrid Chieh Yu

Submitted to the Faculty of Mathematics and Natural
Sciences at the University of Oslo in partial
fulfillment of the requirements for
the degree Philosophiae Doctor in Computer Science

November 2009

Abstract

In today's society, we are becoming increasingly dependent on the reliability and availability of complex software systems distributed across networks such as the Internet. To construct such systems, object-oriented approaches to software development are well suited for encapsulating data and functions into software units that are integrated to form large systems. This support for modular program development combined with the encapsulation property of objects makes object-orientation the leading paradigm for development of distributed systems, where qualities such as limited inter-dependencies between software components, flexibility with regards to changes, and protection of components' integrity are important. Distributed systems are often designed to run continuously, so modifications and extensions of existing software components, as well as additions of new ones, must happen while the overall system is running and available. For critical systems where faults and inaccessibility can have dramatic implications, it is necessary to ensure that system modifications are applied in a safe manner without compromising system availability, and that system crashes do not occur during or after such updates. It is therefore important to develop formal methods that support the development and maintenance of modular non-terminating systems.

This thesis studies type safety of object-oriented distributed systems and dynamic class constructs. Based on the Creol language, the thesis addresses type safety for languages with concurrent objects, multiple inheritance and asynchronous method calls as the communication primitive. Moreover, we investigate an asynchronous upgrade model that captures the distributed nature of systems and study the integration of classical object-oriented features with runtime upgrade mechanisms.

Acknowledgments

First of all, I wish to thank my main supervisor Einar Broch Johnsen. I am grateful for his invaluable guidance, encouragement, stimulating discussions, and for generously sharing his time and knowledge throughout my PhD studies. I want to thank my co-supervisor Olaf Owe for sharing his clear intuition and for advice, especially during the writing and polishing of this thesis.

I would also like to thank everyone involved in the CREOL and CREDO projects at the Department of Informatics, University of Oslo, including PhD students, researchers, and guest scientists. In particular, I thank my former colleague Marcel Kyas for the collaboration on the implementation of the type checker, his hospitality when I visited him in Berlin, and him, Johan Dovland, and Martin Steffen for their varied support over the years and for good conversations, relevant or not to this thesis. I want to thank all my colleagues for their contributions to a friendly working atmosphere and Joakim Bjørk and Hallstein Asheim Hansen for all the highly necessary coffee breaks we shared. My thanks also go to all the researchers who have crossed my path over the years.

Finally, I thank my parents, brother and my extended family and friends for their faith in me, but mostly two persons: Fredrik, who is always there for me, continuously supporting me through all of these years, and Embrik for just being who he is.

Work on this thesis has been done at the Department of Informatics at the University of Oslo and has been supported by a grant from the Research Council of Norway through the CREOL project. Additional funding has been provided by the Department of Informatics at the University of Oslo.

Contents

Part I

Overview

Chapter 1

Introduction and Research Questions

A distributed system can be seen as a program which consists of parts or components that run simultaneously on multiple computers communicating over a network. The considered environments for distributed programs are typically heterogeneous, with network links of varying latencies, and unpredictable failures in the network or the computers. Such systems are often long-lived; hence, the ability to modify the systems by adding, removing or updating the systems' components during their lifetime is important. Examples of such systems are web services, Internet banking systems, mobile phones, air traffic control systems, etc. For distributed systems that are safety-critical, such as air traffic control systems, it is vital that programs behave and run as expected and that changes are applied without compromising system availability, because faults and inaccessibility can create dramatic implications on human lives and can also have great economic consequences. It is therefore important to invest in techniques for preventing errors and making the systems correct during the design process. One of the challenges is to ensure the reliability and correctness of the overall system when components are dynamically (at runtime) modified. Therefore, the development, maintenance, and upgrade of system components should happen in a controlled manner.

One way to promote correct programs is through the features of the programming language. For all software, design is an important part of the development process. A design method guides the software designers in the decomposition of a system into logical components that will eventually be coded in a programming language. The relationship between a software design method and the programming language is therefore important. If the design method and the language paradigm clash, the programming effort increases, making the code hard to understand and error-prone. It is therefore essential to be able to express a program in a way that is natural for the problem, avoiding details and tricks that distract the programmers from the more important activity of problem solving. For distributed programs, it has been claimed that the object-oriented paradigm and distributed systems form a natural match [34]. In object-oriented modeling, components can be represented by objects, and the behavior of the overall system comes from the collaboration of those objects. Also, proper encapsulation of objects supports modular design and allows programs to be changed more easily. To support the design of distributed systems, the language used for writing the programs must be able to reflect the properties of distributed systems such that the design abstractions can be directly mapped into program components. With this in mind, Creol [42,35] emerges as an object-oriented modeling language designed with the explicit goal of supporting the development of distributed systems.

The language targets object-oriented distributed systems by providing a general and intuitive model for concurrency and distribution that is based on *concurrent objects* where each object has its own (virtual) processor, and a notion of *asynchronous method calls*. To support asynchronous method calls in a non-deterministic environment, the language provides constructs for explicit processor release and internal scheduling mechanisms for processes based on guards. Hence, the scheduling of internal activities inside an object is flexible with respect to possible delays and instabilities that occur in distributed computing. Asynchronous method calls are achieved in Creol by decoupling invocations from replies (in this chapter, this is referred to as a call pair), which in the language give return values. Methods can always be asynchronously invoked and because no synchronization nor transfer of execution control is involved (in contrast to synchronous method calls), the execution of the caller can proceed after an invocation, until the return values are needed. To fetch return values, one can use blocking or non-blocking reply constructs. Non-blocking is achieved by using guards, allowing the process to passively wait for the reply of a method invocation by releasing the processor. Such *release points* can then influence the control flow inside objects by allowing processor release when the reply guard for polling the replies, evaluates to false.

Creol supports object-oriented features but emphasizes ease of verification by avoiding reasoning problems related to aliasing, shared variables and thread concurrency. Furthermore, the language has a high level operational semantics in rewriting logic [52] that closely reflects the language syntax. The operational semantics is executable in the Maude framework [18], which also provides an analysis platform for system models. This combination provides a powerful and flexible platform for experimentation with language constructs and suitable concurrent runtime environments, while at the same time, the language is expressive enough to localize and address type safety with respect to the main challenges of such systems, e.g., asynchronicity, concurrency and runtime evolution.

Long-lived distributed applications with high availability requirements need the ability to adapt to new changes that arise over time without compromising application availability. An ideal update system should propagate updates automatically, provide means to control *when* components may be upgraded, and ensure the availability of system services during the upgrade process. This can be achieved by means of runtime class upgrades which allow class definitions to be dynamically reconfigured. In an object-oriented setting such changes imply an evolution of the objects in the running system.

The main questions addressed by this thesis can be summarized by the following questions:

1. How can we statically guarantee type safety for languages with asynchronous method calls, as in the Creol model?

2. How can we statically guarantee type safety for languages with runtime class upgrades?

In order to address these questions, the work in this thesis develops semantics-based analysis methods for systems defined in the Creol language, and for the runtime evolution of the systems by means of a mechanism for *Dynamic classes*, which are classes that are subject to dynamic upgrades. The solutions emphasize implementable analysis techniques and the possibility to be integrated into the Creol framework. This work is presented in four papers, which form the research contribution of this thesis. The aforementioned questions will be further discussed in details below.

One of the most significant approaches to develop correct programs is by employing formal analysis methods such that incorrect programs written in the language can be eliminated at an early stage. The methodology used in this thesis belongs to the family of formal methods. In the framework of programming languages, *type analysis* is a well-established formal method for proving the absence of certain program behaviors by annotating types to programs and examining the flow of values of these types [58, 56]. In this thesis, we use a type and effect system [7, 59, 65] for the verification of programs. Effect systems provide a way of adding context information to the type analysis, and the role of effects is to explicitly approximate the actions of interest that will be performed at runtime. An effect system can denote sets of actions like accessing a value but can also be extended to capture the temporal order and causality of these actions. For example, an effect system can be used to check that a file is not read before the file has been opened. Also, by incorporating type information inside the effects, the effects are made more expressive as the shape of the type information can be influenced by the effects. In this thesis, we are interested in a formal model of distributed systems and the runtime evolution of such systems. In particular, we investigate how the operational semantics can benefit from static information derived by the effect system, such that the soundness of programs and their runtime evolutions can be guaranteed.

To address the research questions stated above, we investigate the following issues:

1. Type safe asynchronous method calls. The type safety property guarantees that a program will not generate type errors when executed. To show that a Creol program is type safe, we consider the different language constructs of Creol and provide techniques that facilitate the verification of programs written using these constructs. In particular, we look into analysis problems associated with method calls:

1. Method binding that fails because the callee does not provide services required by the caller is one kind of runtime error one can experience from systems that are not safe. In Creol, objects are typed by interfaces, so communication with remote objects is defined independent from their implementation. The exact class of an object reference can only be determined at runtime. In this thesis we consider safe bindings of method calls to method bodies in the presence of late binding.

2. With asynchronous method calls, a call is decoupled into several operations: method invocation, polling for a reply, and fetching the reply. Such operations may occur in the context of a given call, which is recognized in the operations by a label name. Hence, label names may be seen as scopes within which these operations refer to a given call. The type analysis must identify the call pairs in a method body and ensure that all possible pairs on each execution path can be correctly bound. This analysis is nontrivial for several reasons. First, the decoupling of calls together with the branching structure and interleaving of release points complicate label scoping. Second, Creol supports method overloading, so the complete actual signature is needed at the call site, but deriving a unique signature depends on the variable types used for the reply operations. There might for example be

several reply operations matching the label of one invocation. Third, the language supports multiple use of reply guards, and replies are consumed by reply operations. Thus, it is desirable to eliminate undesirable combinations of reply operations within the same label scope. Also, for a method invocation, the program may never reach the corresponding reply operation. Therefore, it is semantically desirable to remove superfluous replies once they cannot be accessed by the program.

3. Creol supports *multiple inheritance*, which compared to single inheritance, adds expressiveness to the programming language. With the presence of field and method overloading, ambiguities may occur when fields or methods are accessed. The analysis needs to address ambiguities arising from name conflicts in such programs and handle both late and statically bound method calls.

In Paper #1 and Paper #2 a type analysis method is developed to guarantee the absence of runtime type errors in object-oriented distributed concurrent programs with asynchronous method calls. In order to analyze asynchronous calls, we use an effect system that tracks input and output information for method calls and capture label scoping: the occurrences of calls corresponding to a label must be uniquely identified. Furthermore, when taking a forward analysis approach, deriving a precise signature for a call is done after the analysis of the reply statement and involves a refinement of the previously analyzed invocation. Paper #1 addresses Creol and needs two-passes to add call-site type information to method invocations. In contrast, Paper #2 focuses on a more efficient analysis of asynchronous calls. It introduces a backwards analysis that allows method invocations to be annotated with resulting type information in a single pass. The problem of local memory deallocations related to asynchronous calls and methods for explicit deallocations in the runtime code are also considered in order to avoid memory leakage due to superfluous method replies. For this purpose, a code transformation is incorporated directly into the type and effect system.

2. Type safe dynamic class operations. Long-lived and distributed systems which require continuous system availability still need to be modifiable due to bug fixes, feature extensions, evolving user requirements, etc. For these systems, code change must happen at runtime and changes must be safe. We investigate how runtime class definitions and objects can be modified in a type safe manner. In this thesis, we consider solutions to the following subproblems related to dynamic classes:

1. In an object-oriented model, changes to a class C apply to instances of C, as well as to subclasses of C and theirs instances. We consider operations such as deriving new classes (subclassing) and redefining existing classes. Problems associated with object states, method bindings and upgrades of active objects need to be addressed in the presence of dynamic classes.

2. We are interested in an upgrade model that captures the distributed nature of the system. The ideal upgrade operations should propagate asynchronously through the distributed system and operations should only modify objects that are covered by the upgrade.

3. In an asynchronous system, upgrades may be arbitrarily delayed. They may therefore be injected into the runtime configuration in one order but applied in another. For upgrades that depend on each other, this overtaking may cause unsafe executions. To ensure that upgrades are applied in a correct order, it is necessary to provide a means to control *when* components are updated. In particular, operations for removing fields and methods are nontrivial. For determining when removal is allowed, the upgrade semantics needs to e.g., consider non-terminated processes that may contain statements for method calls and field access to methods and fields which are scheduled for removal, respectively.

The work on type safety continues in the setting of dynamic classes. Program evolution should be user friendly by not imposing particular requirements on the development of the initial programs, and flexible by limiting the constraints on the programming language. In Paper #3 and Paper #4 we propose *dynamic class operations*, a type system for incremental analysis of dynamic upgrades, and a corresponding modular operational semantics. The evolution of a program can be reflected by sequences of typing environments, giving a static view of how programs evolve. However, as mentioned in subproblem 3, the asynchronous nature of distributed systems creates a discrepancy between the runtime view and the static view of the system. To address this discrepancy, we propose a type system where the type derivation assures the type safety of upgrades and at the same time derives the dependencies needed to ensure that upgrades propagate through the system in an appropriate manner. We investigate how the operational semantics of dynamic classes can exploit the statically derived information to impose constraints on the runtime applicability of upgrades, enforcing a delay on certain upgrade operations at runtime such that execution remains type safe.

Type safety for dynamic class upgrades is presented in Paper #3 and Paper #4. In both papers, we address the updates of existing instances. Paper #3 considers adding new classes and features to programs. Adding new interfaces to classes corresponds to an increase in the number of possible behaviors of the objects, whereas redefining the implementations must not violate old interfaces. For system upgrades, we introduce a notion of versions in classes and objects. The work on dynamic class operations continues in Paper #4, where we introduce means to dynamically simplify class definitions, including simplifying the class inheritance hierarchies, and removing fields and methods. The approach for simplifying class definitions is nontrivial and requires more runtime overhead as non-terminated processes that may rely on removed fields and methods must be controlled in order to guarantee soundness.

Tool support Based on the introduced type systems presented in these papers, a prototype type checker for Creol programs including dynamic class operations has been incorporated into the Creol compiler, and, together with the operational semantics, fully integrated in Creol's execution platform. The prototype can be found in [20].

Overview. The introduction part of this thesis is structured as follows: Chapter 2 gives an overview of some general principles of distributed object-oriented systems and introduces the Creol language. Chapter 3 describes how type systems can verify program safety and Chapter 4 motivates the runtime evolution of object-oriented systems. A short summery is given for each paper in Chapter 5, and finally, Chapter 6 concludes and suggests some future directions of research.

Chapter 2

Distributed Object-Oriented Systems

A distributed system can be seen as a program which consists of parts or components that run simultaneously on multiple computers communicating over a network. The considered environments for distributed programs are typically heterogeneous, with network links of varying latencies, and unpredictable failures in the network or the computers that is out of the application control.

It has been claimed that object orientation and distributed systems form a natural match and is recommended by the RM-ODP [34]. Today, object orientation is the leading paradigm for open distributed systems. With object-oriented modeling, the components are represented by objects and the behavior of the overall system results from the collaboration of those objects. The encapsulation property of objects supports several ideals for distributed systems, such as limited inter-dependencies between software components, flexibility with regards to changes, and the protection of the components' integrity. One way to model distributed object-oriented systems is by concurrent objects communicating by method calls. Each object has its own state and when a method is invoked, the activation of the method results in a *process* executing in the called object. Hence, concurrency appears when multiple processes are executed simultaneously. Each process has its own thread of control when initiated and thus multiple threads exist in a distributed object-oriented system. Interference between threads may occur if they operate simultaneously on the same object or on shared variables.

The work in this thesis is based on Creol, a language for distributed concurrent objects. In this chapter, we will briefly introduce some basic concepts of object orientation used in the technical papers in the sequel and present some particular solutions proposed in the Creol language.

2.1 Object Orientation

The concepts underlying object-oriented programming were introduced by the Simula programming language in the 1960's [22]. The idea that a program can be viewed as a collection of cooperating objects, each with its own data structure and methods for processing data became a popular programming paradigm in the 80's. This paradigm has received great attention and popular languages such as Smalltalk [29], C++ [64] and Java [10] emerged, contributing to making object orientation the dominant programming methodology. One characteristic of object-oriented languages above is the use of *classes* as templates for *objects*. Classes have *fields* and *method definitions* and objects are runtime created *instances* of classes. Object creation can for example be done by applying the **new** operator (together with the name of the class). For an object, the values of the fields defined by the object's class represent the internal *state* of the object, and the methods, which comprise program statements, represent operations the object (or other objects) may perform, possibly changing its state and the state of other objects. Objects constitute the dynamic part of a program and may relate to each other through *references* or *pointers*. When communication between objects happens through method *invocations*, an invocation of a method name causes the called object to execute the code that is associated with the method name. In the examples that follow, we use a Java-like syntax and denote by $\overline{\texttt{c}}$ a list of a syntactic construct `c`. The definition of a method with name `m` is represented by $\texttt{T m(}\overline{\texttt{T' x}}\texttt{)\{}\overline{\texttt{s}}\texttt{\}}$, where `T` is the type of the return value, $\overline{\texttt{T' x}}$ is a list of the formal parameters $\overline{\texttt{x}}$ with types $\overline{\texttt{T'}}$, and $\overline{\texttt{s}}$ is the implementation of the method. If `o` is an object reference and `m` is a method of `o`, then we write $\texttt{o.m(}\overline{\texttt{e}}\texttt{)}$ to represent the invocation of method `m` in `o` with actual parameters $\overline{\texttt{e}}$. The following features further introduce some fundamental concepts of object-oriented programming languages [16].

Interfaces. Abstract data types (ADT) can be explained as user-defined types with hidden structure and a set of allowable operations that can be performed on their instances. An ADT is an abstract entity that is specified independent of any particular implementation. For example, ADTs one used in software engineering, when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. In object-oriented programming, ADTs can be defined by *interface specifications* [21]. To illustrate, the following is a definition of a Stack ADT [28], through a `Stack` interface. A stack is a collection of items in which only the most recently added item may be removed. The latest added item is at the top. Examples of other common ADTs are List, Set,Tree, Map and String.

```
interface Stack {
  void push(int i);
  int pop();
}
```

The `Stack` interface has two operations `push` and `pop`. To implement an interface, one may use specific data types or data structures, and provide one method for each ADT operation. In object-oriented programming, interfaces can be implemented using the class construct. Objects of the class that implements the interface type can then be manipulated according to the type's operations, which in this case are the class's methods. There are many ways to implement the same interface, using different concrete data structures. Thus, for example, the abstract stack can be implemented by a linked list or by an array. Below is an implementation of the `Stack` interface using an array.

```
class myStack implements Stack {
  int elements[100];
  int top = 0;
```

```
   void push(int i){
    elements[top] = i;
    top += 1;}

   int pop(){
    top -= 1;
    return elements[top];}
}
```

In this example, the class `myStack` defines a data structure that provides the implementation of the two ADT operations, `push` inserts an element onto the structure and `pop` removes the last inserted element from the structure. By creating an instance of `myStack` and invoking methods `pop` and `push` on this instance, the actual elements are collected and maintained by this structure. If several objects of this class are created, the system will have multiple stacks and the actual content on each stack will differ, depending on the method invocations on the objects at runtime.

One of the important aspects of abstract data types is the encapsulation of internal data structures and operations. Interface specifications describe what operations are provided by objects of classes that implement the interfaces, and the operations' *signatures* specify how the operations can be accessed. An operation's signature typically consists of its name, the type of its the formal parameters, and its return type. The data structure that implements the type is hidden from the user of the type. In our example, users only know which operations to invoke for storing and retrieving elements from a stack given by the interface specification, but it is unknown to the user that the internal structure is an array and that the field `top` gives the next element. This information hiding serves as the basis towards system modularization as we gain a clear distinction between the interface and the implementation. Thus, if objects are typed by interfaces, the use of these operations is independent of how they are implemented, because any correct underlying implementations must satisfy the signatures. This separation of objects' types from implementations also provides more flexibility with respect to code change. For objects that are typed by interfaces, their internal states can no longer be directly accessed, by for example using dot notation to access fields. Instead, accessing the state of an object is done through, e.g., *get* and *set* operations, declared in the interface for retrieving and changing the state. Consequently, provided that changes satisfy old interfaces, modular upgrade mechanisms can be supported as changing the state variables of an object will not break the code of other objects. In our example, a reference `o` with the interface type `Stack` will not need to change its type when the class `myStack` is replaced by a new class `superStack` that gives a different implementation of `Stack`. So, in a context depending on `Stack`, objects of class `myStack` and `superStack` can be used interchangeably. Moreover, different interfaces can be exported by the same class, and different roles of an object can be distinguished through typing. Assume that the class `superStack` is later extended with an interface `Peek` that allows one to inspect and manipulate the contents of a stack. If objects are typed by interfaces, we may export the new interface in a controlled manner by, e.g., creating objects of type `Peek`. Also, we may treat objects of the same class differently depending on their types. For example, a reference with type `Peek` may only occur in some context of the code.

Inheritance. *Class inheritance* defines relationships among classes where new classes can be formed based on already defined classes. Thus, inheritance allows a class to have the same behavior as another class and extend or tailor that behavior to provide special actions for specific needs. Consider the following class:

```
class CountStack inherits myStack{
  int nr_elements()
  {
    return top;
  }
}
```

By letting the class CountStack be a subclass of myStack, the methods push and pop declared in the myStack class are also available to the CountStack class. Furthermore, the subclass provides a new function, nr_elements, that returns the number of elements stored in the stack, by returning the value of top inherited from the myStack class. In this way, inheritance supports the development of classes based on existing ones, without modifying the existing class.

With *Interface inheritance*, the operations and signatures defined in the superinterfaces are inherited in the subinterface. A class that implements the subinterface must therefore provide implementations of methods that are defined both in the subinterface and in all the superinterfaces.

Subtype polymorphism. Subtyping or subtype polymorphism allows a value of type A to masquerade as a value of type B. Assume a function f that is written to take a value of type T as input. By subtyping, if T′ is a subtype of T, we may pass a value of type T′ as input instead and f will still work correctly. One alternative way to define the subtype relation in object-oriented languages is through class inheritance. In Java, where objects can be typed by class names and substitutability is ensured by the language, it is the case that if a class C′ is a subclass of a class C, then C′ is considered to be a specialization of C and we may consider objects of type C′ a subtype of C. This typing relation can be written as $C' \preceq C$. Consider the above example and let s be a reference variable of type myStack. Since the class CountStack only extended myStack with a new method, variables of type myStack may refer to an object of type myStack but also to an object of type CountStack in every context. The assignment s = **new** CountStack() is legal. Any calls to methods specified in myStack will also be understood by the actual CountStack object.

Interfaces also provide the benefits of polymorphism that are available when using derived classes. Assume that both myStack and superStack implement the Stack interface, then we may pass objects of either class to methods where the interface type Stack is expected. In fact, this works for any object that is a subtype of Stack, i.e., an instance of a class that implements an interface that is a subinterface of Stack. Hence, we have subtype polymorphism using *interface inheritance*. Some languages, e.g., Java, support subtype polymorphism for objects through both class and interface inheritance and others, e.g., Creol, defined solely by the interface inheritance. This is because Creol, targeting distributed systems, prioritize strong encapsulation and ease of program verification [24].

A strongly typed language ensures type safety, thus, strongly typed languages limit the polymorphism of variables. A variable of type `T′` is allowed to refer to objects of type `T′` or any subtype of `T′` but not supertypes of `T′`. If the assignment is allowed such that a variable of type `T′` is assigned to an object of type `T` and `T′` $\preceq$ `T`, a runtime error will occur when trying to invoke a method defined in `T′` but not in `T`. Again, we consider the above class hierarchy and let `cs` and `s` be variables of type `CountStack` and `myStack`, respectively. The assignment `cs = s` may potentially lead to a runtime error when the statement `cs.nr_elements()` is evaluated. If the actual runtime object is of type `myStack` then the method name `nr_elements` will not be understood by the object and method binding fails. Consequently, assignments of this kind will not be allowed for strongly typed languages. One way to evade the type system is, for example done in Java, through type casts. In Java, a down-cast allows one to move the reference down the inheritance hierarchy and allows one to assign types to terms that the type checker cannot derive statically. For example if we know that `s` is a `CountStack` object, then we can by down-cast assign `s` to `cs` by `cs = (CountStack) s`. However, unlike up-cast which is always safe, down-cast may fail because the exact type of the object can not be checked until runtime, in which case an exception would be raised if the cast does not succeed (for example by raising a ClassCastException in Java). In order to cover this, Java also supports runtime inspections of types before a type cast, for example by using `instanceof` in Java for type comparison.

Late binding of method calls to method definitions. Inheritance provides not only a means for subclasses to add new functionality but also to redefine or override methods provided in the superclass. The new methods will have the same signatures as the ones provided by the superclass, but with different implementations. In this setting, when a method is called, choices need to be made with respect to the proper binding of the call. Statically, the declaration of the polymorphic variable gives the type of the class, thus, the call can be directed to the statically given class. Dynamically, the binding can be determined at runtime based on the type of the actual object referred to by the variable. The latter is known as *late binding* and gives great flexibility as it allows programmers to invoke methods on an object of unknown class as long as the object has a type that guarantees that the methods exist with the appropriate signatures. This way, objects from different classes may be used interchangeably as long as they have the same super type. Consider the above `myStack` class and assume the following `CountStack` class:

```
class CountStack inherits myStack{
  public int nr_elements()
  {
    return top;
  }

  public int push()
  {
    // new code for push
  }
}
```

In this example, the assignment `s = cs` determines the implementation to execute for the statement `s.push()`. If the object is an instance of `myStack`, the definition in class `myStack` will be executed otherwise the new code in class `CountStack` is chosen. In general, the binding starts from the class of the actual object. If the method is defined in the class, the binding will succeed and the method body is bound to the call. Otherwise, method binding continues to the immediate superclass. Static type checking can be employed to guarantee successful method binding in the presence of polymorphism and late binding.

2.2 Object Communication

The three basic interaction models for concurrent processes are *shared variables*, *remote method invocation* and *message passing* [8]. In the first model, concurrent processes communicate by altering the contents of shared variables. A method call is a request to an object to perform some task. The object that makes the call is referred to as the *caller* whereas the object that receives the request and processes the call is referred to as the *callee*. With a synchronous (remote) method invocation the callee executes the specified process using the supplied actual parameters. When the process terminates, the result is returned to the caller and the caller may continue executing its process. Observe that with this communication model, an object is activated by a method call. The thread of control is transferred with the call so there is a master-slave relationship between the caller and the callee. Caller activity is blocked until the return values from the method call have been received. This synchronized interaction is well suited for tightly coupled systems.

In contrast to remote method calls, message passing is a communication form without any transfer of control between concurrent objects. Message passing may be synchronous, as in Ada's Rendezvous mechanism, in which case both the sender and receiver process must be ready before communication can occur. Hence, the objects synchronize on message transmission. A method call can there be modeled by an invocation and a reply message. Remote method invocations may be captured in this model if the calling object *blocks* between the two synchronized messages representing the call [8]. If the calling object is allowed to proceed with its execution for a while before resynchronizing on the reply message we obtain a different model of method calls which from the caller perspective resembles *future variables* [23,31,67,17,55]. A future variable represents the result of an asynchronous computation where the value is given by an operation that is executed in a separate thread. The value is accessed when needed and the result may be unknown if the computation of its value has not completed. Accessing the future before the future is resolved may lead the current thread or process to block, or lead to an exception. Communication by asynchronous message passing is well-known from, e.g., the Actor model [2,3]. In the asynchronous setting message emission is always possible, regardless of when the receiver accepts a message. Languages with notions of future variables are usually based on asynchronous message passing. In this case, the caller's activity is synchronized with the arrival of the reply message rather than with the emission of the invocation, and the activities of the caller and the callee need not directly synchronize.

In standard object-oriented models, the specific challenges of distributed computation are not addressed. Over time, the environment of a distributed system may be subject to change and is not necessarily within the control of the program developer. Therefore, the environment may not respond as expected from a particular subsystem. The shared variable communication model violates the encapsulation of objects and thus does not capture object interaction in a distributed setting well. Object interaction by means of remote method invocations is usually synchronous. In a distributed setting, synchronous communication gives rise to undesired and uncontrolled waiting, and possibly deadlock. In addition, separating execution threads from distributed objects breaks the modularity and encapsulation of object orientation, and often leads to a low-level style of programming based on, e.g., the explicit manipulation of locks.

The asynchronous message passing approach reflects the fact that communication in a network is not instantaneous and seems well suited to model communication in distributed environments. Without synchronization and transfer of the thread of control, this form of communication avoids unnecessary waiting in the distributed setting and takes into account the inherent latency and unreliability that exist in a network. Thereby this communication model provides better control and efficiency. However, method calls imply an ordering on communication not easily captured in these models. Actors do not distinguish replies from invocations, so capturing method calls with Actors quickly becomes unwieldy [2]. The integration of the message concept in an object-oriented setting is unsettled, especially with respect to inheritance and redefinition (the actor model has no direct notion of inheritance or hierarchy).

An asynchronous communication model that combines the advantages of asynchronous message passing with the structuring mechanism provided by the method concept is studied in this thesis as the basis for modeling object interactions.

2.3 The Creol Language

Creol is a high-level object-oriented language addressing distributed systems and provides a formal framework for modeling and reasoning about such systems [41,42,43,44]. The language is developed by the Precise Modeling and Analysis group at the University of Oslo [19].

Some characteristics of Creol are:

- Concurrent objects.
- Asynchronous method calls.
- Process release points.
- Behavioral interfaces.
- Multiple inheritance.
- Dynamic class upgrades. This feature will be discussed in Chapter 4.

Figure 2.1 gives the syntax of Creol. The main features of the language are discussed in this section, we don't consider standard constructs such as variable declarations, but focus on the syntax related to the call mechanism and process release.

$$
\begin{array}{l}
\texttt{g} \text{ in } \texttt{\textbf{Guard}}\\
\texttt{v} \text{ in } \texttt{\textbf{Var}}\\
\texttt{l} \text{ in } \texttt{\textbf{Label}}\\
\texttt{s} \text{ in } \texttt{\textbf{Stm}}\\
\texttt{m} \text{ in } \texttt{\textbf{Mtd}}\\
\texttt{r} \text{ in } \texttt{\textbf{MtdCall}}\\
\texttt{e} \text{ in } \texttt{\textbf{Expr}}\\
\texttt{b} \text{ in } \texttt{\textbf{BoolExpr}}\\
\texttt{x} \text{ in } \texttt{\textbf{ObjExpr}}
\end{array}
\qquad
\begin{array}{rcl}
\textit{CL} & ::= & [\texttt{\textbf{class}}\ \texttt{C}\ [(\textit{Vdecl})]^{?}[\texttt{\textbf{contracts}}\ [\texttt{I}]^{+}_{;}]^{?}\\
 & & \quad[\texttt{\textbf{implements}}\ [\texttt{I}]^{+}_{;}]^{?}\ [\texttt{\textbf{inherits}}\ [\texttt{C}[(\overline{\texttt{e}})]^{?}]^{+}_{;}]^{?}\\
 & & \quad\texttt{\textbf{begin}}\ [\texttt{\textbf{var}}\ \textit{Vdecl}]^{?}\ [[\texttt{\textbf{with}}\ \texttt{I}]^{?}\ \textit{Mtds}]^{*}\ \texttt{\textbf{end}}]^{*}\\
\textit{Mtds} & ::= & [\texttt{\textbf{op}}\ \texttt{m}\ ([\texttt{\textbf{in}}\ \textit{Vdecl}]^{?}\ [\texttt{\textbf{out}}\ \textit{Vdecl}]^{?}) == [\texttt{\textbf{var}}\ \textit{Vdecl};]^{?}\ \texttt{s}]^{+}\\
\textit{Vdecl} & ::= & [\texttt{v} : \texttt{T}]^{+}_{,}\\
\texttt{g} & ::= & \texttt{b} \mid \texttt{l?} \mid \texttt{g} \wedge \texttt{g} \mid \texttt{g} \vee \texttt{g}\\
\texttt{r} & ::= & \texttt{x.m} \mid \texttt{m} \mid \texttt{m@C}\\
\texttt{s} & ::= & \texttt{s;s} \mid \texttt{l!r}(\overline{\texttt{e}}) \mid \texttt{!r}(\overline{\texttt{e}}) \mid \texttt{l?(v)} \mid \texttt{\textbf{await}}\ \texttt{g} \mid \texttt{v} := \texttt{e} \mid \texttt{v} := \texttt{\textbf{new}}\ \texttt{C}(\overline{\texttt{e}})\\
 & & \mid \texttt{s} \Box \texttt{s} \mid \texttt{s} \parallel\!\mid \texttt{s} \mid \texttt{\textbf{if}}\ \texttt{b}\ \texttt{\textbf{then}}\ \texttt{s}\ \texttt{\textbf{else}}\ \texttt{s}\ \texttt{\textbf{fi}}\\
 & & \mid \texttt{\textbf{while}}\ \texttt{b}\ \texttt{\textbf{do}}\ \texttt{s}\ \texttt{\textbf{od}}
\end{array}
$$

Figure 2.1: An outline of the syntax of Creol. `C` is a class name, `T` a type and `I` an interface name. Statements for loop, conditional, assignment and object creation are standard.

Concurrent objects. A concurrent object in Creol is an encapsulated unit of computation: Its internal state can only be accessed by the object itself, and the object also conceptually encapsulates its own thread of control. Interference between processes inside an object is by shared attributes. However, each process has exclusive access to the object's internal representation as only one process is active at any time. Objects are concurrent in the sense that each object has a processor executing its processes and different objects execute in parallel. Concurrency is implicit rather than explicit in the language syntax. Creol objects are high-level in the sense that local data structures in objects are defined by abstract data types. We assume a functional sublanguage for defining the data structures and functions performing local computations on terms of such data types. Expressions in **`Expr`** and **`BoolExpr`** belongs to the functional sublanguage which we assume given and without side-effects.

Asynchronous method calls and process release points. As discussed in Chapter 2.2, synchronous communication can give rise to undesired and uncontrolled waiting, and if the thread of control is never returned to the caller, the process deadlocks. Asynchronous method calls provide more flexibility and allow concurrent execution of processes between different caller and callee objects. However, if a reply is not available when needed, the calling process may block the processor while waiting for the reply unless **`await`** reply. Such blocking will exclude other processes that also compete for the processor, leading to reduced performance in the calling object. Mechanisms for processor release allow processes to better adapt to the external environment such as delays, and introduce a more flexible use of the processor. In the Creol language, this is supported by so-called *process release points* declared in the program code. Release points, expressed using Boolean guards, influence the implicit control flow inside objects by allowing processor release when the guard evaluates to false. This makes it straightforward to combine active (e.g., non-terminating) and reactive processes in an object. Thus, an object may behave both as client and server for other objects without the need for an active loop to control the interleaving of these different roles. By using processor release points with guards for polling the reply between a method invocation and access to the return values, a process allows the processor to be released while waiting for the reply. Asynchronous call mechanisms add flexibility to method calls in distributed settings because a waiting process may yield processor control to suspended and enabled processes instead of actively blocking the processor. In Creol, the statement l!o.m($\overline{\texttt{e}}$) represents an invocation of method m on object o with actual parameters $\overline{\texttt{e}}$ and label l, which identifies the call. Since the call is asynchronous, the execution of the caller continues after the method is invoked. When the result is needed, the reply statement l?(v) fetches the return value identified by l and assigns the value to variable v. If the reply is not available, the process is blocked. However, by using a guard **await** l? for polling the reply before fetching the reply, the process can instead be released. Note that the label l ties the invocation to its corresponding guard and reply. If a return value is not needed, the call !o.m($\overline{\texttt{e}}$) suffices (ignoring the label associated with the call). Synchronous calls can be encoded by having reply statements immediately following the invocations. For label-free variations of Creol, a synchronous call is represented by o.m($\overline{\texttt{e}}$; v) and similarly, **await** o.m($\overline{\texttt{e}}$; v) represents a label-free asynchronous call, defined by l!o.m($\overline{\texttt{e}}$);l?(v) and l!o.m($\overline{\texttt{e}}$); **await** l?; l?(v), respectively.

In order to allow different tasks within the same process to be selected depending on the order in which communication with other objects actually occurs, the language provides a more fine-grained control of processes, increasing component adaptability. Creol has in addition to standard sequential composition operators between program statements, the *non-deterministic choice* and the *non-deterministic merge* operators. These operators introduce high-level branching structures that allow the process in a concurrent object to take advantage of non-deterministic delays in the environment in a flexible way. The process may adapt itself to the distributed environment without yielding control to a competing process. The non-deterministic choice operator can be used to encode interrupts for process suspension such as timeouts, or model race conditions between competing asynchronous calls [42]. For example, one may write l!o.m(e); l′!o′.m′(e′); (l?(v) $\Box$ l′?(v)) which allows a non-deterministic assignment of one of the reply values to the variable v. With the non-deterministic merge operator, we may express that both program statements $\overline{\texttt{s}}$ and $\overline{\texttt{s}'}$ in (**await** l?; $\overline{\texttt{s}}$) ||| (**await** l′?; $\overline{\texttt{s}'}$) will be executed, but the order is determined by the arrival of the method replies. If the reply associated with l (or l′) arrives first, $\overline{\texttt{s}}$ (or $\overline{\texttt{s}'}$) will be chosen to execute first. If neither reply has arrived, the processor is released.

We illustrate the use of asynchronous method calls and control flow operations by means of an example from [46] (inspired by [54]), considering services that provide news at request. Let News be an interface with method news, which for a given date returns the news of that day in an XML format. Let CNN and BBC be two news sites. By calling CNN.news(d) and BBC.news(d), news from CNN and BBC for the specified date d will be downloaded, respectively. Email is a service for sending emails to clients at request. By calling method send(m, **caller**), where m is a message and **caller** the identity of the calling object, m will be sent to client **caller**. Following is a class with method newsRelay that requests news from both CNN and BBC, but only the earliest arriving response will be relayed to the client. The latest arriving response will be ignored.

```
class RelayingNewsService(CNN:News, BBC:News, e:Email)
```

```
  implements NewsService
begin
  with Client
    op newsRelay(in d:Date) ==
     var v:XML, l:Label, l':Label; l!CNN.news(d); l'!BBC.news(d);
     ((await l?; l?(v)) □ (await l'?; l'?(v))); !e.send(v, caller)
end
```

In method `newsRelay` the news on both sites are invoked asynchronously and the labels is used to couple invocations with replies. The process release points expressed by **`await`** `l?` and **`await`** `l'?` allow the executing process to be suspended while waiting for `CNN` and `BBC` to respond. Thus, the processor is not blocked if neither service responds. The non-deterministic choice operator between release points is used to capture the fact that news relay depends on the environment and the remote service. The `e.send` service will be invoked with the result received from either `CNN` or `BBC`, depending on the news received first.

Behavioral interfaces. The interface specification of a communicating component provides information about what services can be expected from the component. It describes the legal interactions between components, and object communication is therefore limited to methods that are specified in the objects' interfaces. An *interface* consists of a set of method names with signatures, and semantical constraints on the use of these methods. Interface inheritance is restricted to a form of *subtyping*. An interface may inherit from several interfaces, in which case the interface is extended with the syntactic and semantical requirements of its super-interfaces. In many object-oriented languages, the type of objects generated by a subclass is a subtype of objects generated by a superclass, but this identification of types and code has been subject to criticism [62, 6]. If one separates the concerns between external service specifications, given as interfaces, and implementation code, organized in classes, objects should be typed by interfaces and not by classes. In this setting, interface inheritance should respect behavioral subtyping restrictions [50]. Consequently, a class may inherit from another class, but the type of the object generated by the subclass may not be a subtype of the type of the objects generated by the superclass. Compatibility between objects is determined by their observable behavior. One object may therefore have various types, defined by the object's respective interfaces. In an open distributed setting, this allows an object to play different roles depending on different contexts. Moreover, in a context depending on interface `I`, an object that supports interface `I` may be replaced by another object supporting `I` or a subinterface of `I`, although the latter object may be of unrelated class with a different implementation of the methods. This supports modularity as the actual class of the object implementing `I` is not visible to users. The call `o.m()` where `o` is of type `I` may rely on the specification of `m` as given by the interface `I`. As object variables (references) are typed by interfaces, late binding applies to all external method calls, because the runtime class of the called object is not statically known.

Creol proposes means to restrict access to methods by restricting calling objects to those supporting the *cointerface* [38, 39]. The cointerface, specified by the **with** construct in Figure 2.1, is considered as a part of the method's signature during method binding. In the above example, `Client` is the cointerface of method `newsRelay`. So, in order for `newsRelay`

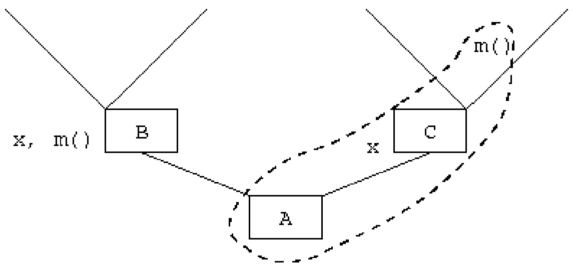

Figure 2.2: A multiple inheritance class hierarchy. Class A inherits the classes B and C. Class B contains a field x and a definition of method m. Class C contains a field x and inherits a definition of m through one of its superclasses.

to be called on objects of type NewsService, the calling object, defined by the keyword **caller**, must implement the interface Client. In addition, it states that the type of the **caller** is a subtype of Client. The cointerface allows type correct call-backs, for example send(v, **caller**) and **caller**.m() where m is a method in the interface Client.

Multiple inheritance. *Multiple inheritance*, using the keyword **inherits**, allows a class to inherit directly from several ancestor classes. It is a powerful and expressive object-oriented mechanism and is included in Creol to study the full power of object orientation as well as to formalize and reason about integration of classical object-oriented features with upgrade mechanisms and labels. With multiple inheritance, ambiguities may occur when attributes or methods are accessed [60,63,66,48]. Name conflicts are vertical when a name occurs in a class and in one of its ancestors, and horizontal when the name occurs in distinct branches of the inheritance tree [48]. In Creol, vertical name conflicts for method names are generally resolved by choosing the first matching definition while ascending a branch of the inheritance tree and horizontal by means of a *pruned binding strategy* [40], which dynamically depends on the class of the object and the context of the call. Figure 2.2 shows a horizontal conflict where a method m is defined twice, once in class B and once in a superclass of C. Creol orders superclasses from left to right as given textually by the inheritance declaration in each class. When binding a method call to m on an instance of class A, the binding strategy will traverse the inheritance tree and search for the definition of m in a left-first, depth-first manner. If this call to m occurs in a method defined in class C, it is undesirable to incorrectly bind it to the method definition in class B since m may behave differently. An incorrect binding is avoided by statically identifying the call with a declaration of m in some superclass of C and let the call be constrained by this superclass [45]. The dashed region in Figure 2.2 illustrates method binding restricted by the superclass of C where m is defined. Note that if A redefines m, a late bound call will bind to the definition of m in A.

Creol uses qualified names to internally refer to an attribute or method in a class in a unique way. The **qua** construct from Simula [22] (originally not affecting virtual (late bound) methods) is adapted to multiple inheritance, where static calls can be directed to any superclass. This is useful in the presence of horizontal name conflicts, when there are different paths through an inheritance hierarchy and the programmer wants to override the default binding for late bound calls. For example, by using m@B the call is directed statically to m as defined in B or inherited from the superclasses of B. Similarly, attribute names in different classes can be reused by expanding them to qualified names. As shown in Figure 2.2 where x is defined in class B and in class C, and both x'es can be referred to by instances of class A. If x occurs in method m defined in class B, the compiler may expand it to x@B, specifying that in the body of m, x defined in B shall be dereferenced. In this way, when methods of superclasses assign to overloaded attributes, the intended variable will be selected.

With objects typed by interfaces, a class may inherit from a superclass without supporting the same interfaces. Consequently, a remote self call in the context of the superclass may statically satisfy the cointerface requirement associated with the method, but the actual calling object may be of a subclass and typed by an interface that does not support the cointerface. To address this, Creol introduces the key word **contracts**, representing interfaces implemented by a class and all subclasses, whereas the key word **implements** represents interfaces implemented by the class, but not imposed on subclasses.

2.3.1 Example: A peer-to-peer network

In this example, we illustrate the usefulness of asynchronous method calls and processor release points in Creol. We consider a peer-to-peer file sharing system consisting of nodes distributed across a potentially unstable network. In the example, a node plays both the role of a server and of a client. A node may request a file from another node in the network and download it as a series of packets until the file download is complete. As nodes may appear and disappear in the unstable network, the connection to a node may be blocked, in which case the download should automatically resume if the connection is reestablished. A client may run several downloads concurrently, at different speeds. We assume that every node in the network has an associated database with shared files. Downloaded files are stored in the database, which implements an interface DB and is not modeled here:

```
interface Client
begin with Any
   op availFiles (in sList: List[Server]
             out files: List[[Server, List[String]]])
   op reqFiles(in sId: Server, fId: String)
end

interface Server
begin with Server
   op enquire(out files: List[String])
   op getLength(in fId: String out lth: Int)
   op getPack(in fId: String, pNbr: Int out pack: Data)
end

interface Peer inherits Client, Server
begin end

class Node(db: DB) contracts Peer
begin with Server
   op enquire(out files: List[String]) ==
      await db.listFiles(; files)
    op getLength(in fId: String out lth: Int) ==
      await db.getLength(fId; lth)
    op getPack(in fId: String, pNbr: Int out pack: Data) ==
      var f: List[List[Data]];
      await db.getFile(fId; f); pack := nth(f, pNbr)

  with Any
    op availFiles (in sList: List[Server]
           out files: List[[Server , List[String]]]) ==
      var l1: Label, l2: Label, fList: List[String];
      if (sList = nil) then files := nil
      else
        l1!head(sList).enquire();
        l2!this.availFiles(tail(sList));
        await l1? ∧ l2?;
        l1?(fList); l2?(files);
        files := files |- (head(sList), fList)
      fi

    op reqFile(in sId: Server, fId: String) ==
      var file: List[Data], pack: Data, lth: Int;
      await sId.getLength(fId; lth);
      while (lth > 0) do
        await sId.getPack(fId, lth; pack);
        file := pack -| file; lth := lth - 1
      od;
      !db.storeFile(fId, file)
end
```

Nodes in the peer-to-peer network are modeled by the class `Node`, which implements the `Peer` interface. In this example, the `Peer` interface inherits two interfaces, `Server` and `Client`, allowing `Peer` nodes to act according to both the client role and the server role. The cointerface requirement of each superinterface restricts the use of the methods inherited from that superinterface. The `Server` interface provides methods `enquire`, `getLength` and `getPacket` that are only available to other servers in the network. Hence the **with**-clause **`with`** `Server`. In contrast, the `Client` interface has no communication constraints, the cointerface of the provided methods is `Any`, which is the superinterface of all interfaces.

In the `Node` class, the methods `reqFile`, `enquire`, and `getPack` requests the file associated with a file name from a node, enquires which files are available from a node, and fetches a particular packet in a file transmission, respectively. For the functional sublanguage of Creol, list appending and prepending are represented by `|-` and `-|`, respectively. Let `x:T` and `s:List[T]`, the functions `head` and `tail` are given by `head(x -| s) = x`, `tail(x -| s) = s`, and `nth(s, i)` gives the `i`'th element of `s`.

To motivate the use of asynchronous method calls and release points, we consider the method `availFiles` in detail. The method takes as formal parameter a list `sList` of nodes and, for each node, finds the files that may be downloaded from that node. The method returns a list of pairs, where each pair contains a node name and a list of (available) files. All method calls are *asynchronous*. Therefore, after the first call, identified by label `l1`, the rest of the node list may be explored recursively, without waiting for the reply to the first call. *Process release points* ensure that if a node temporarily becomes unavailable, the process is suspended and may resume at any time after the node becomes available again. Thus, when the statement **`await`** `l1?`$\wedge$ `l2?` evaluates to false, the processor will not be blocked, and other processes may execute. A node may have several interleaved activities: several downloads may be processed simultaneously with uploads to other nodes. When **`await`** `l1?`$\wedge$ `l2?` is enabled, the return value is assigned to the out parameter of the caller (which is retrieved by `l2?(files)`), which completes the process.

The whole program uses asynchronous calls, in the form of **`await`** `o.m(`$\overline{e}$`)`, `!o.m(`$\overline{e}$`)`, and `l!o.m(`$\overline{e}$`)` with the corresponding guard **`await`** `l?` before fetching the reply. Consequently, the same node may serve many peers at the same time and handles any overtaking of invocations and replies. If the connection to some nodes disappears or if there are unpredictable delays in the network, other peers using the node will not be affected by it. It is therefore trivial to see that the system will not deadlock.

Chapter 3

Type Analysis

Formal methods for predicting safe approximations to the set of values a program computes, or more general how the program might behave at runtime, are essential in order to ensure that systems behave properly. In programming languages, a formalized *type analysis* is a well-established formal method for proving the absence of certain program behaviors by annotating programs with types and examining the flow of the values that occurs in the programs [56, 58, 59]. A type analysis typically tries to guarantee that operations expecting values of a certain type are not used with values that the operations do not understand. Otherwise, an incorrect usage of an operation may appear during the execution and cause *runtime type errors*. These errors can be related to situations where the runtime system does not define what to do because the program has reached some undefined state. In concrete implementations, these states corresponds to runtime type errors or possibly if not detected they may lead to other faults such as overflow, segmentation faults, execution of illegal instructions, etc. In object-oriented programs, *method not understood errors*, which occur when method binding fails, is a typical example of a runtime type error that one tries to avoid by applying type analysis.

Recently, type systems and the corresponding soundness results have been shown for formalizations of object-oriented systems, which challenge the traditional understanding of class hierarchies. FJIG [49] studies flexible notions of subclassing, Fickle [26] addresses dynamic object re-classifications, and UpgradeJ [14] allow programs to have multiple versions of a class. Our work fits in this line of research and similar to Fickle, we use type and effect systems in this thesis.

3.1 Type Systems

A *type system* is a formalization of a type analysis in terms of a set of rules used for a language to structure and organize its collection of types and to limit the set of legal programs that can be written in the language. A program is well-typed if it is accepted by the type system. If the typing rules are syntax directed then one can read the rules top-down and yield a bottom-up type checking algorithm in the sense that the inner most expressions are analyzed first. Typing judgments for terms have the form $\Gamma \vdash t : T$, where Γ is the *typing environment*, t the term, and T the type of t. For example, the typing rules for variables and method invocations in Featherweight Java [33] are as follows:

$$\text{(T-VAR)} \quad \Gamma \vdash x : \Gamma(x) \qquad\qquad \text{(T-INVK)} \quad \frac{\begin{array}{c}\Gamma \vdash t_0 : C_0 \\ mtype(m, C_0) = \overline{T} \rightarrow U \\ \Gamma \vdash \bar{t} : \overline{T'} \quad \overline{T'} \preceq \overline{T}\end{array}}{\Gamma \vdash t_0.m(\bar{t}) : U}$$

Here, the typing environment Γ is a finite mapping from variables to types. Rule (T-VAR) checks if a variable x is declared. The variable type is stored in the typing environment and can be inspected by $\Gamma(x)$. The typing statements on sequences are abbreviated, the shorthand $\Gamma \vdash \overline{t : T'}$ stands for $\Gamma \vdash t_1 : T'_1, \cdots, \Gamma \vdash t_n : T'_n$ and $\overline{T'} \preceq \overline{T}$ for $T'_1 \preceq T_1, \cdots, T'_n \preceq T_n$ (where $T' \preceq T$ expresses that T' is a subtype of T, as discussed in Chapter 2.1). For a well typed external method call, Rule (T-INVK) states that if t_0 evaluates to an expression of type C_0 and the auxiliary function *mtype*, given the method name m and the class name C_0, returns the declared signature of m above C_0, then each argument has a type T'_i that is a subtype of the type T_i declared for the corresponding formal parameter, i.e., $T'_i \preceq T_i$. Assuming no method overloading, the method name is sufficient for finding the method signature in the class hierarchy for a given class, and the return type of the call is U.

A type system is *strong* if it guarantees *type safety* or *soundness*, which means that programs type-checked by the type system are guaranteed to be semantically free of type violation, thereby the slogan *"Well-typed programs cannot go wrong"* [53]. A language with a strong type system is is said to be a *strongly typed* language. This implies that functions and method calls that disregard typing requirements are rejected. Soundness of the type system consists of proving the *subject reduction theorem* [16] showing that if a term is well-typed, then it will never in the future reach an untrapped error state (e.g, not handled by some exception handlers in the runtime system). In order to prove soundness of a type system, it is necessary to have a formalization of the runtime system, i.e., an *operational semantics* for the language. Let e and e' be terms and let $e \rightarrow e'$ denote a runtime evaluation of e to e' according to the rules of the operational semantics. The subject reduction theorem for systems with subtyping can be formulated as follows: If $\Gamma \vdash e : T$ and $e \rightarrow e'$, then $\Gamma \vdash e' : T'$ where $T' \preceq T$. The subject reduction theorem guarantees that the type system is consistent with the computation rules and gives semantic correctness of programs. Consider Rule (T-INVK) and let r be the body of m and $x_1 : T_1, \ldots, x_n : T_n$ the formal parameters of m. The subject reduction requires that the beta reduction $r[x_1/t_1 ... x_n/t_n]$, where x_i/t_i denotes an substitution of t_i for every free occurrence of x_i, is also well-typed. By establishing the subject reduction property for a program, all instantiations of the method body with the actual parameters provided by the calls in the program, as well as any further executions of these, are well-typed.

3.2 Static v.s. Runtime Analysis

There are two categories of type analysis, *static* and *dynamic*, which means that type checking may occur either at compile-time (a static check) or at runtime (a dynamic check). Static type checking allows early error detection and ensures that type errors will not occur in any possible execution of a program. In static type checking, *types are associated with variables*. As dynamic type information is not available at compile-time, static type checkers are conservative. They can prove the absence of some bad behavior but cannot in general prove their presence and hence may reject programs because they cannot be statically determined to be well-typed even if they are actually well-behaved at runtime. In turn, static typing can find type errors reliably at compile time and hence increases the reliability of the delivered program. Moreover, static typing usually results in compiled code that executes more quickly. When the compiler knows the exact data types that are in use, it can produce optimized machine code and dynamic type checking can be avoided or simplified.

In dynamic typing, *types are associated with values*, not variables. Dynamically typed languages, compared to statically typed ones, make fewer compile-time checks on the source code. Runtime checks can potentially be more sophisticated, since they can use dynamic information as well as any information that was present during compilation. Dynamic checking can allow programs to generate types based on runtime data and can therefore accept programs that have acceptable runtime behavior but would otherwise be rejected by static type checkers. Dynamic checking requires the program to be executed on sample input data so program development in dynamically typed languages is often combined with programming practices such as unit testing [12]. This allows potential errors to be detected at runtime but one must first provide input that reveals the error. Runtime checks only assert that conditions hold in a particular execution of the program, and since errors may only exist in a portion of the program, these checks need to be repeated for every execution of the program. In contrast, as mentioned, static type checkers are able to verify that checked conditions hold for all possible executions of the program, but as a consequence, they may conservatively reject programs that actually have acceptable runtime behavior. (Languages with membership tests enables further flexibility.) With dynamic checking, the cost of the flexibility means fewer a priori guarantees and may result in errors with variables having unexpected types, e.g., string instead of integer. The error could appear a long time after creating the assignment, making the bug potentially difficult to locate.

Static type checkers are typically built into compilers, which means that type checking can be done automatically. The compilers can do code transformations, insert runtime checks, and apply other static analysis techniques on programs, e.g., code optimizations or, as in this thesis, incorporate garbage collection in program code for explicitly de-allocating memory.

3.3 Static Type Analysis for Asynchronous Method Calls

This section discusses some of the challenges associated with the type analysis of asynchronous systems, motivated through the Creol language.

For language safety, asynchronous method calls complicate the otherwise standard static analysis of method calls as the invocation and the corresponding replies are decoupled as seen from the caller. Especially, the scoping of labels combined with branching of program statements challenges the decision procedure of method bindings. The type system must track the different invocation and reply pairs that may occur, in all execution paths, and give a deterministic signature for each call. Furthermore, the type system seeks a linear usage of reply statements while processor release points must follow a method invocation. Breaking these requirements would result in permanently blocking the processes or objects, which is clearly undesirable.

Typing with labeled invocation and reply statements, where the binding depends on out-parameters, is illustrated by the following example. Consider the example from Chapter 2.3 and assume that class `CNNsite` implements interface `News`, which provides the following two methods:

```
class CNNsite implements News
begin
  op news(in d:Date out m:XML) == ...
  op news(in d:Date out m:StreamDescription) == ...
end
```

Based on the typing information of the reply statement `l?(v)` in method `newsRelay`, the call `l!CNN.news(d)` will have the output type `XML`. This typing information can be passed on to the runtime system to ensure proper binding of the call, and the call will be bound to the first declaration of `news` in `CNNsite`. In this example, the reply on `l` only occurs in one execution path. Determining the scope of `l` is straight-forward. However, consider the statement list bellow and assume that `l` does not occur in $\overline{\mathtt{s}}$ and $\overline{\mathtt{s'}}$:

$$\mathtt{(l!o.m(}\overline{\mathtt{e}}\mathtt{)}\ \Box\ \mathtt{l!o'.m'(}\overline{\mathtt{e}}'\mathtt{));}\ \overline{\mathtt{s}}\mathtt{;}\ \mathtt{(l?(v)}\Box\ \mathtt{l?(v')}\ \Box\ \overline{\mathtt{s'}}\mathtt{)}$$

At an execution point, several potential invocations and replies are associated with the same label `l`. It is non-deterministic which invocation and reply pair will be evaluated at runtime, although replies to either invocation may never be reached if $\overline{\mathtt{s'}}$ is evaluated. The static type system needs to derive a signature for each call so that runtime evaluations are well-typed independent of the selected execution branch.

An *effect system* [7, 56, 59] is a formal system that approximates some computational effects of computer programs and adds context information to the analysis, such as side effects. Effect systems often come as an extension to the type system. A effect system can denote sets of actions like accessing a value but can also be extended to capture the temporal order and causality of these actions. In this thesis, the analysis of asynchronous object communication is based on extending the type system with an effect system. This technique of incorporating effect systems into type systems is also used in the analysis of program evolution and for detecting memory leakage. The memory leakage problem is related to return values from method calls that will never be used. In this case, Baker and Hewitt [11] observed that the asynchronous method call gives rise to memory leakage through passive storage of the future containing the method reply. Therefore the method replies can be considered as semantically garbage in specific execution paths. Whereas a traditional tracing garbage collector cannot free memory space that is reachable from the state of an object [47], we exploit type and effect analysis to identify method replies that are semantically garbage in specific execution paths and insert operations to explicitly free memory in those paths (even if the replies could still be referred to in the process memory).

Chapter 4

Runtime System Evolution

Long-lived distributed applications with high availability requirements need the ability to adapt to new changes that arise over time without compromising application availability. These changes include bug fixes but also new or improved features. Examples of such applications are found in e.g. financial transaction processes, aeronautics, space missions, and Internet applications. In some of these areas, it is a great advantage that upgrades are applied at runtime, while for others, an absolute requirement. Many distributed applications that require high availability are safety critical systems. For these systems, it is vital that updates are applied in a safe manner and that system crashes do not occur during or after such updates.

An ideal update system should propagate updates automatically, provide means to control *when* components may be upgraded, and ensure the availability of system services during the upgrade process [4, 61]. With a modular design choice, developers can create a new module that provides new services and insert the module into the environment. When the system is ready for the update, the new services can be accessed. Also, existing modules can be updated and components using these modules should not be affected by the changes. For example, version control systems target modular upgrade support. Some approaches allow multiple module versions to coexist after an upgrade [13, 9, 27, 30, 32], while others keep only the last version [57,51,4,15]. Part two of this thesis considers runtime evolution in the setting of Creol and how to develop static techniques based on type checking to ensure that evolution is type safe in the asynchronous setting with Creol's concurrent objects and its mechanism for dynamic class upgrades. In the object-oriented setting of Creol, we propose to support runtime evolution by updating class definitions at runtime. This chapter describes some of the different design approaches to program evolution for object-oriented systems.

4.1 Updating Instances

There are several choices for handling existing objects when a class is updated: none, some, or all instances can change to match the new class definition. Figure 4.1 shows three possible models for instance updates: version barrier, passive partitioning, and global update [32]. Objects and their lifespan are represented with tailed bullets. In each model, a white bullet represents an object's old state, while a black bullet represents objects with updated state.

With *barrier solution* (A), an update of a class cannot occur until all existing objects of the

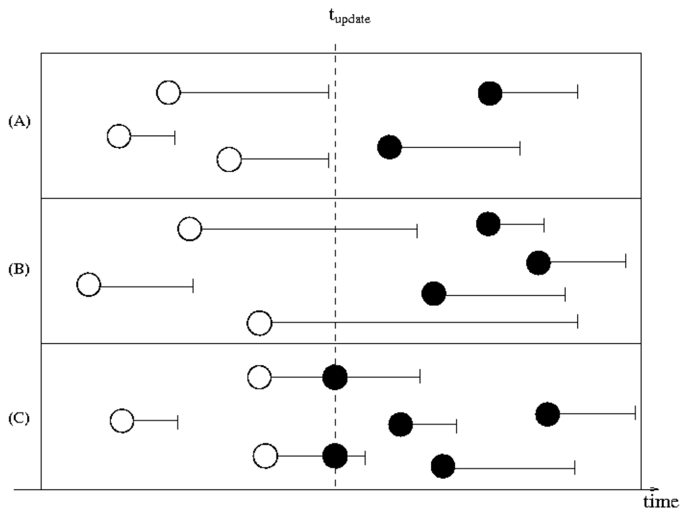

Figure 4.1: Models for instance updates

old version of the class have expired, which for example in Java means that they can no longer be accessed by any pointers. This approach is conceptually equivalent to halting, modifying and restarting the system. *Passive partitioning* (B) supports the coexistence of multiple versions of a class by not taking any action on existing objects. All new objects are created with the new version and existing objects are uninterrupted, thus only newly created objects may use the functionality introduced by an upgrade. In a program with multiple class versions, multiple implementations of each method can coexist. When a method is invoked on a particular object, the call must be redirected, e.g., through mapping to the correct implementation. Also, object creation is non-trivial with inheritance as a class may inherit from a particular version of some class. Then, when an object of a subclass is created, to establish the object's state, the correct versions of its superclasses must be identified. Moreover, system modularity can quickly be broken as client code may be dependent on particular versions. The approach of Hjálmtýsson and Gray [32] for C++ is based on passive partitioning, and uses proxy classes that link to the actual classes (reference indirection). To avoid program modules depending on particular versions, public operations must remain constant across all versions. In UpgradeJ [14] the programmer must explicitly refer to the different class versions in the code as versions are made explicit, for example using `instanceof` to dynamically identify the class version of an object. With this approach, the dynamic class upgrade mechanism can be made more lightweight without heap update, and objects of different class versions can be arbitrarily created, but this can result in bristle code and encapsulation becomes more difficult to achieve. In distributed systems, objects are often long-lived, so a well-suited update model for such systems needs to provide means for updating the objects' states.

Finally, in the model of *global update* (C), all instances of a class and its subclasses are updated [51,57]. This approach is often realized by recreating all existing objects using the new version, which means copying object states and thus requires an understanding of the objects' semantics. A characteristic of this model is to atomically migrate all instances from the old to the new version of the class, and no instances of the old classes should be executing after the update is completed. In [51], the Java virtual machine is modified to support dynamic classes. Updating an object consists of locking the object, allocating a new object where the old object

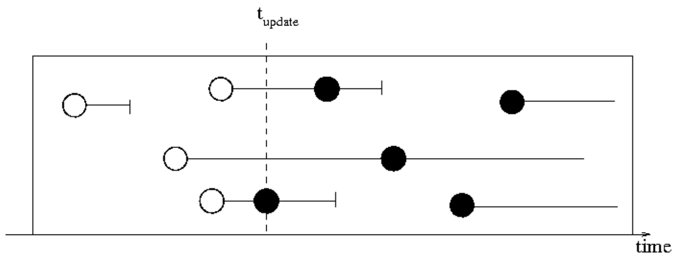

Figure 4.2: An asynchronous upgrade model (used in the Creol model)

state is copied onto the new one, redirect all references, and garbage collect the old object. To avoid runtime errors during class redefinition, all threads must be blocked until the redefinition is complete. Upgrades by lazy global updates have been proposed for distributed objects [5] and persistent object stores [15], where instances of upgraded classes are updated, but inheritance and (non-terminating) active code are not addressed, limiting the effect and modularity of class upgrades.

Global update places a challenge on the update of active methods while running: Given that methods m_{old} and m_{new} have no particular relationship and assume that m_{old} is currently running. It is in general impossible to determine where and how to continue execution in m_{new}. For example, if m_{old} and m_{new} solve the same problem using different algorithms, there may not be any program point in m_{new} corresponding to the current location in m_{old}. By continuing the execution in m_{new} based on the program counter of m_{old}, one could easily gets unexpected results. Also m_{new} may even use variables not present in m_{old}. In [57], substituting a class requires that no methods of the class are executing during the update.

In large distributed systems, it seems reasonable to assume that there is always some active code being executed somewhere in the system. Moreover, halting such systems is usually undesirable, and for critical systems, disastrous. Therefore, runtime updates need to be applied in an asynchronous and modular way, and propagate gradually throughout the distributed system. The ability to address active methods and object states is important as objects are often active and long-lived. Global update proposes object update but lacks the possibility to update active objects. Moreover, the property that, at any time, all objects of a particular class are of the same version is too restrictive and unnatural with respect to the characteristics of such systems. Figure 4.2 depicts the upgrade model introduced in this thesis. If an update is applied on a class, only the latest version of the class is kept. Consequently, objects created after the class upgrade will automatically have the new state and functionality. The upgrade mechanism is asynchronous: When a class is upgraded, its objects (including subclass objects) will gradually be affected by this change. Some objects may be updated prior to others in a non-deterministic order. The updated objects may execute a new implementation of some methods while other objects are still running the old method implementations. With this situation in mind, upgrades of existing instances must be closely controlled as errors may occur if new or redefined methods, relying on fields that are not yet available in the object, were executed. Similarly, errors occur if there are objects running old implementations containing references to fields or calls to methods which are removed by some updates.

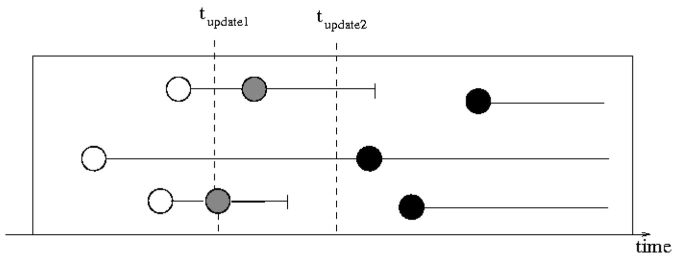

Figure 4.3: Race conditions in the asynchronous upgrade model

4.2 Timing

Besides different models of updating instances, there are also different approaches to the treatment of *when* updates are applied. Some are synchronized on annotated program points, others have exceptions handling invalid upgrades. In [14], upgrade statements are included explicitly in the original program. When such statements are executed, the program will wait to receive an upgrade. Such upgrade mechanisms allow programmers to control the timing of upgrades of the application. However, it requires that the original design is well thought through with respect to future changes. In [15], upgrades are serialized to avoid race conditions, ensuring that upgrades are applied in the order they are inserted, and in [51] invalid upgrades raise exceptions. The aforementioned methods are not ideal in our setting. Also, with recursive or non-terminating methods objects cannot generally be expected to reach a state without pending processes, even if the activation of new method calls were postponed as suggested in [5]. Consequently, it is too restrictive to wait for the completion of all processes before applying an upgrade.

With the global update approach, if a class C is of some version n, then all objects of C will comply with this latest version. Consequently, to update C from version n to version $n+1$, it may suffice to only consider the state change from the last version to the new one. However, with distributed systems, nodes may be temporarily unavailable and communication may fail. It may be too restrictive to require that all objects of a class are up to date with the current class version, before the next update of the class can be applied. In the asynchronous setting, upgrade mechanisms may face the following problems: objects may not be synchronized with their current class versions when facing the next update. For these objects, some variables may not be available and there may be some pending processes with old code (code not stemming from the current class version) and this code may contain method calls to methods removed later. Also, for languages with message passing as the communication primitive, there may be some pending messages in the configuration. The treatment of these processes and messages with respect to an upgrade must be made clear. Another problem associated with an asynchronous upgrade mechanism is that several updates may be competing to be applied. Figure 4.3 shows the possibility of race conditions where an object has missed `update1` and is updated to `update2`. Also, `update1` may have been injected into the runtime environment before `update2`, but the actual update is overtaken by `update2`. This is particularly problematic in the case where `update2` crucially depends on changes to the class which were introduced in `update1`.

4.3 Type Soundness of Asynchronous Upgrades

In the setting of Creol, we consider the asynchronous upgrade model depicted in Figure 4.2 as it captures the nature of distributed systems well. Program evolution needs to be done in a type safe manner. Consequently, the type system of Creol is extended to cover the dynamicity of the class and interface hierarchy of a program. The typing environment, which keeps track of the typing information for various program constructs, must reflect the evolution of the program. Conceptually, this is straight-forward and in this thesis it is done as a sequence of typing environments, but the evolution of the type system becomes a challenge when taking the asynchronous nature of Creol programs into account. The sequence of typing environments $\ldots\ \Gamma^i, \Gamma^{i+1} \ldots$ reflects the evolution of a program. However, the update from Γ^i to Γ^{i+1} can not be expected to instantaneously be applied globally in the runtime system. The *discrepancy* between the static environment Γ^i and the current runtime environment must be considered, and this entails the following problems: any changes to the implementation must be type-checked before the change becomes effective in any object, but with the discrepancy between the runtime and the static environments, it is unclear how to guarantee the soundness of well-typed upgrades. Changes may depend on each other and the dependencies must be identified: a method m may be re-implemented to make use of a new method. Clearly, the new method definition must be in place before the update of m is effective at any object offering this method. Also, with processor release, it is necessary to ensure that removed variables or method definitions are not accessed by any pending processes.

In this thesis, we investigate how a type and effect system can be used to capture dependencies between class versions, and how these dependencies can be exploited at runtime to control the evolution of classes and objects in distributed systems. All problems presented in Chapter 4.2 will be addressed in Paper #3 and Paper #4.

Chapter 5

Overview of the Papers

This chapter gives a short summary of each research paper in Part II of this thesis. The contents of the papers appear as in their original publication, but have been reformatted to fit the layout of this thesis. The language syntax and notations used in the papers may vary, some papers are Creol specific, while others are based on a more general object-oriented language. In Paper #1 and Paper #2, the syntax is Creol specific, including labels for asynchronous calls, process release points, and Creol's high-level branching structures. Paper #3 and Paper #4 focus on the upgrade mechanism and therefore use a higher abstraction for the call mechanism than in Paper #1 and Paper #2.

5.1 Paper #1: Creol: A Type-Safe Object-Oriented Model for Distributed Concurrent Systems

Authors: Einar Broch Johnsen, Olaf Owe and Ingrid Chieh Yu
Publication: Theoretical Computer Science 2006 [45].

Summary: This paper presents a statically and strongly typed language for distributed systems and shows type soundness with proofs for subject reduction. The paper covers in detail the syntax and semantics of Creol, and gives a type system for the language with particular attention to the following issues: the subtyping relation, the bindings of asynchronous method calls, the typing of control structures like while-loops, conditionals and the Creol specific high-level branching structures (i.e., non-deterministic choice and non-deterministic merge), multiple inheritance for both interfaces and classes, and soundness of the type system. The subtyping relation is defined for both basic and interface types and is extended to function spaces. In particular, object variables are typed by behavioral interfaces. For asynchronous communication, we introduce method call identifier types, whose values provide unique references to the calls.

With method overloading, type checking of a method call requires not only the types of the callee and caller objects, but also the type of the input parameter and the return type. In this setting, the type analysis of asynchronous method calls requires a more sophisticated analysis strategy than standard synchronous method calls, as the correspondence between input and output parameters is controlled by label names. In order to derive signatures for asynchronous calls, an *effect system* is employed to capture combinations of call pairs that are yet to be fully analyzed. The effect system comprises mappings from labels to tuples containing the information needed to later refine the type analysis of the calls, and their complete analysis is postponed until their corresponding replies are reached. The type system also considers calls without reply statements by providing a preliminary signature, provided it is sufficiently precise in the given context. With high-level branching such as explicit use of non-deterministic choice, the scoping of labels further complicates the analysis of asynchronous calls. A reply statement may potentially correspond to several invocations, or an invocation can reach several matching replies. Our effect system distinguishes between the different invocations and handles the possible matches by indexing the label mapping. Also, a mapping from labels to unions of indices is used to bookkeep and identify calls that are to be refined when type checking a reply. For guarded statements involving labels, this mapping is used as the basis for the verification. Finally, we consider the constraints on types that are needed to ensure deterministic signatures for the invocations without restricting the language too much. This involves operations on the typing environments from the different execution branches.

With multiple inheritance, the main typing concerns are static and virtual bindings of calls. Also, formal parameters of a class may be instantiated with values from its subclasses. Our approach ensures type-correct instantiation of superclasses. Problems concerning cointerface restrictions and remote calls to self are addressed by allowing remote calls to self when the class itself and all subclasses, have an interface allowed as cointerface for the method.

5.2 Paper #2: Backwards Type Analysis of Asynchronous Method Calls

Authors: Einar Broch Johnsen and Ingrid Chieh Yu
Publication: Journal of Logic and Algebraic Programming 2008 [46]. A shorter version of this paper appears in Proceedings of Nordic Workshop on Programming Theory 2006 .

Summary: This paper considers a different type checking approach than the one in Paper #1. The former approach requires two passes of analysis in order to obtain runtime code extended with static type information for method binding, whereas the type analysis presented in this paper can be done in a single pass and the analysis is simplified. In this paper, type analysis is backwards, allowing each asynchronous invocation to be type-checked directly as the corresponding reply has already been analyzed. The type system directly translates source code to runtime code and infers type information in method invocations, which is not done in Paper #1, ensuring a type-correct runtime method lookup for asynchronous method calls.

In addition, the memory leakage problem caused by redundant method replies is addressed. The type and effect analysis identifies method replies that are semantically garbage in specific execution paths and automatically inserts deallocation operations for local garbage collection explicitly freeing memory in those paths. This is a fine-grained deallocation strategy. The deallocation operations can be inserted even if the reply is reachable from the object state. Moreover, deallocation is eager in the sense that deallocation instructions are inserted as early as possible in each branch after method invocations, but to avoid deadlock, after any operations for polling replies.

The correctness of the deallocation operations is asserted from the type system and the subject reduction property is established for the language and for the deallocation mechanism.

5.3 Paper #3: Type-Safe Runtime Class Upgrades in Creol

Authors: Ingrid Chieh Yu, Einar Broch Johnsen and Olaf Owe
Publication: Proceedings of the 8th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006) [69].

Summary: This paper presents *dynamic classes* and is an extension of [44], which considers the propagation of upgrade messages. The dynamic class upgrade mechanism allows class hierarchies to be updated in such a way that existing objects of the upgraded class and its subclasses gradually evolve at runtime. New external services may be introduced and old services may be reprogrammed. In this paper we develop semantics-based analysis where type soundness of dynamic classes is considered, and introduce an operational semantics that benefit from informations derived by the type and effect system. The paper covers problems related to an asynchronous upgrade mechanism, such as race conditions and upgrade dependencies. The formalization is shown in a label-free version of the Creol language, and allows the following *dynamic class operations*: adding new class definitions, interfaces, and updating existing classes by implementing new interfaces, adding new fields, superclasses, and method definitions as well as redefining existing methods.

For old processes to remain well-typed, the proposed static type system disallows method signatures to be arbitrarily redefined. The redefined method signatures can change covariant on the return types and contravariant on the input parameters ensuring type safety. Also, types of fields can not be altered. To guarantee that the update mechanisms maintain type safety in asynchronous distributed settings, we introduce a type and effect system that infers and collects dependencies on previous upgrades. These dependencies are accumulated throughout the analysis of the program statements and are exploited at runtime to impose constraints on the runtime applicability of a particular upgrade. In this paper, we show that the applicability of an upgrade depends on the *class constraints* generated by the type analysis, the individual objects are not inspected.

5.4 Paper #4: Dynamic Classes: Modular Asynchronous Evolution of Distributed Concurrent Objects

Authors: Einar Broch Johnsen, Marcel Kyas and Ingrid Chieh Yu
Publication: Research Report, Department of Informatics, University of Oslo [37]. A shorter version of this report is published in the Proceedings of the 16th International Symposium on Formal Methods (FM 2009) [36].

Summary: This paper extends the previous approach to type safe asynchronous program evolution with operations to simplify class definitions such as removing superclasses, fields and method definitions. With asynchronous method calls and processor release, dynamic class operations for removal are nontrivial and necessitate additional runtime overhead: the applicability of an upgrade depends not only on the classes but also propagates to the runtime objects. We show that *class constraints* and *object constraints* suffice to guarantee soundness of upgrades.

The language studied in this paper includes first-class futures and is based on [23]. Futures are similar to labels in Paper #1 and Paper #2, but restricted to avoid type inference for the asynchronous calls.

Chapter 6

Discussion

In this chapter, we recapture the main research questions formulated in Chapter 1 and in the light of these questions, we discuss the contributions of this thesis.

6.1 Contributions

1. Type safe asynchronous method calls. Based on the Creol language, a type system for asynchronous method calls is introduced in Paper #1 and Paper #2. These papers treat the subproblems formulated in Chapter 1:

1. The static type analysis of class definitions is modular. Based on the provided interface definitions, each class can be analyzed purely in the context of its superclasses, ignoring the implementations of other (referenced) objects. A class must be well-typed with respect to the interfaces it implements. Hence, a class will provide at least one implementation for every method defined in its interfaces. Statically, a method call only relies on the type of the called object, the parameter types, and that the object's interface provides a matching method. The connection between the class and the object's type is established during the analysis of object creation. A well-typed **new** statement guarantees that an object, which is an instance of a given class, has the type that is implemented by the class. Consequently, during runtime, it is ensured that any call to the object, understood by the interface, will also be understood by the object's class. By the established subject reduction property of the type system and that the well-typed program has a well-typed initial state, late binding is guaranteed to succeed.

2. With asynchronous method calls, a call is decoupled into several operations: method invocation, polling for a reply, and fetching the reply. The typing information needed for binding asynchronous method calls can be captured by the effects of the different Creol statements. The typing information associated with a label can be further refined during the analysis. However, in languages with branching structures, labels may occur in the context of different execution branches with dynamic scoping. To resolve the scoping problems, we introduce operations on effects and derive a result type when effects of two or more branches overlap on a common label, which requires operations on types. To avoid deadlock, the effect system also captures the temporal order and causality of

actions associated with labels, such as linear read operations on labels and that replies are not polled after a reply operation on a given label. We observed that similar to live variable analysis, the analysis of asynchronous method calls and garbage collection operations can benefit from a backwards analysis approach. The effects are propagated in a reverse manner, transforming the source code to runtime code where statements for garbage collection and method signatures are inserted.

3. The type and effect system is extended to support *multiple inheritance* illustrating that the analysis also handles horizontal and vertical name conflicts. Soundness of method bindings and object creations are, among others issues, being treated.

The formalization is carried out for the Creol language, and depends on Creol's notion of asynchronous method calls. In particular, the use of label names and label values to associate reply statements with invocation statements is essential. Thus, the type checking of the label-free version of Creol (where more high-level constructs are provided for synchronous and asynchronous calls, but without the fine-grained control of calls as in the full language) is significantly simpler. The results are relevant for other languages with a notion of asynchronous method calls with a decoupling of invocation and reply statements as in Creol, e.g., ABCL [68] in combination with futures and method overloading.

2. Type safe dynamic class operations. A formal model of runtime evolution in distributed systems is considered. Type safe dynamic class operations and their semantics are covered in Paper #3 and Paper #4. For the subproblems presented in Chapter 1, we observed the following:

1. *Dynamic class operations* and the type system are introduced as a part of this thesis. For introducing new classes or redefining existing ones by redefining existing or adding new methods, the analysis is incremental in the sense that the updates are analyzed based on already established typing environments. Existing superclasses, subclasses and methods not redefined by the given upgrade, do not need to be re-analyzed. For removing method definitions, the classes to be re-analyzed are limited to the subclasses of the updated class. The dynamic class operations are formalized without the notion of multiple class versions, therefore exempting programmers from coding with explicit class versions, which are not part of the Creol syntax. Objects created after an upgrade will automatically have the new state. In Creol, non-terminating activity is defined by recursion or loops. This facilitates updates of active objects, where the objects' states can be updated when the processor is released at each call cycle. Updates propagate to subclasses through version comparisons. For this purpose, the version change of a class will cause a version change of its direct subclasses, and in turn affect classes further down in the inheritance hierarchy. For objects that are affected by the upgrade, a new state will be established. In our approach, instance updates are not eager, but happen before new extended states are accessed. New interfaces allow new services to be exported. After a class has been upgraded to implement a new interface, calls to the methods of this interface can be made on instances with the new type. If new subclasses are introduced which redefine some methods, calls on these methods will be bound to the new definitions. The type system ensures that method

redefinitions in classes are safe with respect to the old signatures. Removing method definitions is limited to externally invisible methods. The type system verifies that calls made by subclasses can still be bound somewhere in the inheritance hierarchy, when methods are removed from a given class.

2. We were interested in a formal upgrade model that captures the distributed nature of systems and the suggested asynchronous and distributed upgrade mechanism seems to fit well in distributed configurations. With updates represented as messages, they will propagate through the distributed system, similar to the asynchronous call mechanism. The upgrade is modular as once the class is updated, only instances affected by the upgrade will be asynchronously notified. This applies to all instances of the upgraded class and its subclasses.

3. Effects are attached to each program statement and an effect system is used to capture dependencies from an upgrade to different classes in the system. This statically derived information is used at runtime to resolve the discrepancy problem that exists between the runtime system and the static view of the classes, and between runtime classes and their instances, as discussed in Chapter 4. Thus statically derived information controls when the upgrade can be safely applied. Especially, we illustrate how statically derived information can be used to solve nontrivial problems related to operations for removing superclasses, fields and methods, where the runtime applicability of an upgrade depends on the processes and messages in the configuration. The statically derived information forces dependent upgrades to be applied in a correct order while other upgrades can be applied simultaneously or in any order. In order to prove soundness, the typing rules for runtime configurations encode delayed upgrades to simulate future configurations.

In Paper #3 and Paper #4, we show how the Creol upgrade mechanism can be applied to a general object-oriented language. In order to apply the upgrade mechanism to languages with a more liberal use of labels (or future variables) as studied in Paper #1 and Paper #2, the type system for labeled calls and the type system for dynamic class operations may be combined to derive a type system for the complete language. This can be achieved by including effects from both type systems, in the typing environment of the target system.

6.2 Future Work

A natural extension of our work in the direction of formal analysis methods is to look at the *reasoning* aspect of dynamic class operations that is concerned with the behavioral correctness of code. *Lazy behavioral subtyping* [25] supports incremental reasoning where new subclasses can be introduced without re-verifying already established properties of the superclasses. A new subclass is analyzed in the context of its superclasses and the established properties of the superclasses are maintained by verification conditions on the subclass. Such a modular reasoning approach seems well-suited for verifying programs that are incrementally developed and is directly applicable to dynamic class operations where class hierarchies are being extended by new subclasses. The lazy behavioral subtyping approach to program verification relaxes the conditions for behavior preservation to context-dependent subtyping. Therefore, a method in a subclass that redefines a method in its superclass does not need to preserve the entire specification of the superclass' method. It suffices to adhere to the parts of the specification that are actually used at the call-sites. This relaxation gives a greater flexibility to the possible behavioral changes provided by dynamic class operations. However, dynamic class upgrades adds a level of complexity to proof systems for object-oriented programs, as subclasses are not only added incrementally, but classes in the middle of the hierarchy can change after the sub- and superclasses have been verified. It is interesting to investigate how a framework like lazy behavioral subtyping can be adapted to dynamic class operations and what kinds of restrictions must be imposed on operations. For example, removing fields may have consequences for already established proof outlines, and the degree of verification and re-verification that is necessary when a method is introduced, redefined or removed, remains to be investigated.

The operational semantics for dynamic class operations considers singleton class representations. In distributed computing, class definitions may not be centralized, but rather duplicated and spread out to different locations. Therefore, one possible extension to the current approach is to consider how multiple copies of classes can be updated independently. One possible approach is updating the master copy of the class and let all classes synchronize with the master copy.

Java is a conventional general-purpose object-oriented language and one interesting direction of future work is to investigate how our proposed dynamic class operations can be adopted in the Java setting. In Creol, when a method is called, the process will have a copy of the method body. To increase computation effectiveness, we may consider Creol with shared code and derive a model that is similar to Java, where the code is kept in a runtime class table. Conceptually, one possible solution in the code sharing setting could be to keep the old class definitions when classes are redefined. At runtime, new method calls are directed to the latest version of the class. When (eventually) all method calls are bound to the latest versions, old class definitions become superfluous and may be removed, avoiding having different co-existing class versions except for old method activations. For objects' states, the operational semantics of Creol for dynamic class operations ensures that the objects' states are updated before new methods which may depend on the new state, are executed. This is done by equational reductions in our semantics, which have priority over regular reductions [52]. In the Java setting, object updates may be more involved and may be reflected by some local locking discipline corresponding to the priority of equational reductions in our semantics.

Bibliography

[1] *Proceedings of the General Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*. USENIX, 2003.

[2] G. A. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In E. Najm and J.-B. Stefani, editors, *Proc. 1st IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'96)*, pages 135–153, Paris, 1996. Chapman & Hall.

[3] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, Jan. 1997.

[4] S. Ajmani, B. Liskov, and L. Shrira. Scheduling and simulation: How to upgrade distributed systems. In *9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, pages 43–48, Lihue, Hawaii, May 2003.

[5] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In D. Thomas, editor, *Proc. 20th European Conference on Object-Oriented Programming (ECOOP'06)*, volume 4067 of *Lecture Notes in Computer Science*, pages 452–476. Springer-Verlag, 2006.

[6] P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In N. Meyrowitz, editor, *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 25(10), pages 161–168, New York, NY, Oct. 1990. ACM Press.

[7] T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.

[8] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[9] J. L. Armstrong and S. R. Virding. Erlang - an experimental telephony programming language. In *XIII International Switching Symposium*, June 1990.

[10] K. Arnold, J. Gosling, and D. Holmes. *Java(TM) Programming Language, The (4th Edition)*. Addison-Wesley Professional, 4 edition, 2005.

[11] H. G. Baker and C. E. Hewitt. The incremental garbage collection of processes. *ACM SIGPLAN Notices*, 12(8):55–59, 1977.

[12] Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[13] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating. In *Proc. 2nd Intl. Workshop on Unanticipated Software Evolution (USE)*, April 2003.

[14] G. Bierman, M. Parkinson, and J. Noble. UpgradeJ: Incremental typechecking for class upgrades. In *Proc. 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *Lecture Notes in Computer Science*, pages 235–259. Springer-Verlag, 2008.

[15] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In R. Crocker and G. L. S. Jr., editors, *Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, pages 403–417. ACM Press, 2003.

[16] K. B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. The MIT Press, Cambridge, Mass., 2002.

[17] D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer-Verlag, 2005.

[18] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. *Maude 2.4 Manual*. Computer Science Laboratory, SRI International, 2008.

[19] Creol homepage. `http://www.ifi.uio.no/~creol`.

[20] Creol Tools. `http://heim.ifi.uio.no/~ingridcy/creoltools`.

[21] O.-J. Dahl. *Verifiable Programming*. International Series in Computer Science. Prentice Hall, New York, N.Y., 1992.

[22] O.-J. Dahl, B. Myrhaug, and K. Nygaard. (Simula 67) Common Base Language. Technical Report S-2, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, May 1968.

[23] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, Mar. 2007.

[24] J. Dovland. *Incremental Reasoning about Distributed Object-Oriented Systems*. PhD thesis, University of Oslo, 2009.

[25] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. In J. Cuellar and T. Maibaum, editors, *Proc. 15th International Symposium on Formal Methods (FM'08)*, volume 5014 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, May 2008.

[26] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: Fickle$_{\text{II}}$. *ACM Transactions on Programming Languages and Systems*, 24(2):153–191, 2002.

[27] D. Duggan. Type-Based hot swapping of running modules. In C. Norris and J. J. B. Fenwick, editors, *Proc. 6th Intl. Conf. on Functional Programming (ICFP'01)*, volume 36, 10 of *ACM SIGPLAN notices*, pages 62–73, New York, Sept. 3–5 2001. ACM Press.

[28] C. Ghezzi and M. Jazayeri. *Programming Language Concepts*. John Wiley & Sons, 3rd edition, 1998.

[29] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[30] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, 1996.

[31] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.

[32] G. Hjálmtýsson and R. S. Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proc. USENIX Tech. Conf. (USENIX '98)*, May 1998.

[33] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[34] International Telecommunication Union. Open Distributed Processing - Reference Model parts 1–4. Technical report, ISO/IEC, Geneva, July 1995.

[35] E. B. Johnsen, J. C. Blanchette, M. Kyas, and O. Owe. Intra-object versus inter-object: Concurrency and reasoning in Creol. In *Proc. 2nd Intl. Workshop on Harnessing Theories for Tool Support in Software (TTSS'08)*, volume 243 of *Electronic Notes in Theoretical Computer Science*, pages 89–103. Elsevier, July 2009.

[36] E. B. Johnsen, M. Kyas, and I. C. Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In A. Cavalcanti and D. Dams, editors, *Proc. 16th International Symposium on Formal Methods (FM'09)*, volume 5850 of *Lecture Notes in Computer Science*, pages 596–611. Springer-Verlag, Nov. 2009.

[37] E. B. Johnsen, M. Kyas, and I. C. Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. Research Report 383, Department of Informatics, University of Oslo, 2009.

[38] E. B. Johnsen and O. Owe. A compositional formalism for object viewpoints. In B. Jacobs and A. Rensink, editors, *Proc. 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)*, pages 45–60. Klüwer Academic Publishers, Mar. 2002.

[39] E. B. Johnsen and O. Owe. Object-oriented specification and open distributed systems. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science*, pages 137–164. Springer-Verlag, 2004.

[40] E. B. Johnsen and O. Owe. A dynamic binding strategy for multiple inheritance and asynchronously communicating objects. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. 3rd International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *Lecture Notes in Computer Science*, pages 274–295. Springer-Verlag, 2005.

[41] E. B. Johnsen and O. Owe. Inheritance in the presence of asynchronous method calls. In *Proc. 38th Hawaii International Conference on System Sciences (HICSS'05)*. IEEE Computer Society Press, Jan. 2005.

[42] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[43] E. B. Johnsen, O. Owe, and E. W. Axelsen. A run-time environment for concurrent objects with asynchronous method calls. In N. Martí-Oliet, editor, *Proc. 5th International Workshop on Rewriting Logic and its Applications (WRLA'04), Mar. 2004*, volume 117 of *Electr. Notes Theor. Comput. Sci.*, pages 375–392. Elsevier Science Publishers, Jan. 2005.

[44] E. B. Johnsen, O. Owe, and I. Simplot-Ryl. A dynamic class construct for asynchronous concurrent objects. In M. Steffen and G. Zavattaro, editors, *Proc. 7th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, volume 3535 of *Lecture Notes in Computer Science*, pages 15–30. Springer-Verlag, June 2005.

[45] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.

[46] E. B. Johnsen and I. C. Yu. Backwards type analysis of asynchronous method calls. *Journal of Logic and Algebraic Programming*, 77(1–2):40 – 59, 2008.

[47] R. E. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley, 1996.

[48] J. L. Knudsen. Name collision in multiple classification hierarchies. In *Proc. European Conference on Object-Oriented Programming (ECOOP'88)*, pages 93–109. Springer-Verlag, 1988.

[49] G. Lagorio, M. Servetto, and E. Zucca. Featherweight jigsaw: A minimal core calculus for modular composition of classes. In *Proc. 23rd European Conference on Object-Oriented Programming (ECOOP'09)*, pages 244–268, Berlin, Heidelberg, 2009. Springer-Verlag.

[50] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.

[51] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In E. Bertino, editor, *Proc. 14th European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, pages 337–361. Springer-Verlag, June 2000.

[52] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[53] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[54] J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Software and Systems Modeling*, 6(1):83–110, Mar. 2007.

[55] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364:338–356, 2006.

[56] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, Germany, 2 edition, 2005.

[57] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. In *Proc. International Conference on Software Maintenance (ICSM'02)*, pages 649–658. IEEE Computer Society Press, Oct. 2002.

[58] B. C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Mass., 2002.

[59] B. C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[60] A. Snyder. Inheritance and the development of encapsulated software systems. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 165–188. The MIT Press, 1987.

[61] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. D. Silva, G. R. Ganger, O. Krieger, M. Stumm, M. A. Auslander, M. Ostrowski, B. S. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *USENIX Annual Technical Conference, General Track* [1], pages 141–154.

[62] N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Proc. 5th International Conference on Software Reuse (ICSR5)*, pages 206–215. IEEE Computer Society Press, 1998.

[63] B. Stroustrup. Multiple inheritance for C++. *Computing Systems*, 2(4):367–395, Dec. 1989.

[64] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2000.

[65] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[66] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety proof for multiple inheritance in c++. In *Proc. Object oriented programming, systems, languages, and applications (OOPSLA'06)*. ACM Press, 2006.

[67] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. Object oriented programming, systems, languages, and applications (OOPSLA'05)*, pages 439–453, New York, NY, USA, 2005. ACM Press.

[68] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System*. Series in Computer Systems. The MIT Press, 1990.

[69] I. C. Yu, E. B. Johnsen, and O. Owe. Type-safe runtime class upgrades in Creol. In R. Gorrieri and H. Wehrheim, editors, *Proc. 8th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 of *Lecture Notes in Computer Science*, pages 202–217. Springer-Verlag, June 2006.

Part II

Research Papers

Chapter 7

Paper 1: Creol: A Type-Safe Object-Oriented Model for Distributed Concurrent Systems

Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu

Publication.
Published in Theoretical Computer Science, 2006.

Abstract.
Object-oriented distributed computing is becoming increasingly important for critical infrastructure in society. In standard object-oriented models, objects synchronize on method calls. These models may be criticized in the distributed setting for their tight coupling of communication and synchronization; network delays and instabilities may locally result in much waiting and even deadlock. The Creol model targets distributed objects by a looser coupling of method calls and synchronization. Asynchronous method calls and high-level local control structures allow local computation to adapt to network instability. Object variables are typed by interfaces, so communication with remote objects is independent from their implementation. The inheritance and subtyping relations are distinct in Creol. Interfaces form a subtype hierarchy, whereas multiple inheritance is used for code reuse at the class level. This paper presents the Creol syntax, operational semantics, and type system. It is shown that runtime type errors do not occur for well-typed programs.

7.1 Introduction

The importance of distributed computing is increasing in society with the emergence of applications for electronic banking, electronic police, medical journaling systems, electronic government, etc. All these applications are critical in the sense that system breakdown may have disastrous consequences. Furthermore, as these distributed applications are nonterminating, they are therefore best understood in terms of non-functional or structural properties. In order to reason about the non-functional properties of distributed applications, high-level formal models are needed.

It is often claimed that object orientation and distributed systems form a natural match. Object orientation is the leading paradigm for open distributed systems, recommended by the RM-ODP [43]. However, standard object-oriented models do not address the specific challenges of distributed computation. In particular, object interaction by means of (remote) method calls is usually synchronous. In a distributed setting, synchronous communication gives rise to undesired and uncontrolled waiting, and possibly deadlock. In addition, separating execution threads from distributed objects breaks the modularity and encapsulation of object orientation, and leads to a very low-level style of programming. Consequently, we believe that distribution should not be transparent to the programmer as in the RPC model, rather communication in the distributed setting should be explicitly asynchronous. Asynchronous message passing gives better control and efficiency, but does not provide the structure and discipline inherent in method declarations and calls. In particular, it is unclear how to combine message passing with standard notions of inheritance. It is unsettled how asynchronous communication and object orientation should be combined. Intuitive high-level programming constructs are needed to unite object orientation and distribution in a natural way. We propose a model of distributed systems based on concurrent objects communicating by asynchronous method calls.

This paper presents the high-level object-oriented modeling language Creol [49,47,50,48], which addresses distributed systems. The language is based on concurrent objects typed by behavioral interfaces, communication by asynchronous method calls, and so-called processor release points. Processor release points support a notion of non-blocking method calls, and allow objects to dynamically change between active and reactive behavior. The model integrates asynchronous communication and multiple inheritance, including method overloading and redefinition. In order to allow flexible reuse of behavior as well as of code, behavior is declared in interfaces while code is declared in classes. Both interfaces and classes are structured by multiple inheritance, but inheritance of code is separated from inheritance of behavior. Consequently, the implementation code of a class may be reused without inheriting the external behavior of the class. Creol has an operational semantics defined in rewriting logic [58], which is executable with Maude [21] and provides an interpreter and analysis platform for system models.

This paper extends previous work on Creol by introducing a nominal type system for Creol programs. It is shown that the execution in objects typed by behavioral interfaces and communicating by means of asynchronous method calls, is type-safe. In particular, method binding always succeeds for well-typed programs. Behavioral interfaces make all external method calls virtually bound. The typing of mutually dependent interfaces is controlled by a notion of *contract*. Furthermore, it is shown that the language extended with high-level constructs for local control, allowing objects to better adapt to the external nondeterminism of the distributed environment at runtime, remains type-safe. Finally, the full Creol language is considered, with multiple inheritance at the class level and a *pruned binding strategy* for late bound internal method calls. It is shown that executing programs in the full language is type-safe.

Paper overview. The rest of this paper is structured as follows. Section 7.2 presents behavioral interfaces used to type object variables. Section 7.3 presents an executable language with asynchronous method calls, its type system, and its operational semantics. The language, type system, and operational semantics are extended in Section 7.4 with local control structures, and in Section 7.5 with multiple inheritance and the pruned binding strategy. Section 7.6 discusses related work and Section 7.7 concludes the paper.

7.2 Behavioral Object Interfaces

In object-oriented viewpoint modeling [77, 80, 38], an object may assume different roles or views, depending on the context of interaction. These roles may be captured by specifications of certain parts of the externally observable behavior of objects. The specification of a role will naturally include both syntactic and semantic information about objects. A *behavioral interface* consists of a set of method names with signatures, and semantic constraints on the use of these methods. Interface inheritance is restricted to a form of behavioral subtyping. An interface may inherit several interfaces, in which case it is extended with their syntactic and semantic requirements.

Object variables (references) are typed by behavioral interfaces. Object variables typed by different interfaces may refer to the same object identifier, corresponding to the different roles the object may assume in different contexts. An object *supports* an interface *I* if it complies with the role specified in *I*, in which case it may be referred to by an object variable typed by *I*. A class *implements* an interface if its object instances support the behavior described by the interface. A class may implement several interfaces. Objects of different classes may support the same interface, corresponding to different implementations of the same behavior. Reasoning control is ensured by substitutability at the level of interfaces: *an object supporting an interface I may be replaced by another object supporting I or a subinterface of I* in a context depending on *I*, although the latter object may be of another class. This substitutability is reflected in the executable language by the fact that virtual (or late) binding applies to all external method calls, as the runtime class of the called object is not statically known.

For active objects we may want to restrict access to the methods provided in an interface, to calling objects of a particular interface. This way, the active object may invoke methods of the caller and not only complete invocations of its own methods. Thus callback is supported in the run of a protocol between distributed objects. For this purpose, an interface has a semantic constraint in the form of a so-called *cointerface* [45, 46]. The communication environment of an object, as considered through the interface, is restricted to external objects supporting the given cointerface. For some objects no such knowledge is required. In this case the cointerface is *Any*, the superinterface of all interfaces. *Mutual dependency* is specified if two interfaces have each other as cointerface.

$$\begin{array}{lll} IL & ::= & [\textbf{interface}\ I\ [\textbf{inherits}\ [I]^+_{,}]^?\ \textbf{begin}\ [\textbf{with}\ I\ Msig^*]^?\ \textbf{end}]^* \\ Msig & ::= & \textbf{op}\ m\ ([\textbf{in}\ Param]^?\ [\textbf{out}\ Param]^?) \\ Param & ::= & [v : T]^+_{;} \end{array}$$

Figure 7.1: A syntax for the abstract representation of interface specifications. Square brackets are used as meta parenthesis, with superscript $^?$ for optional parts, superscript * for repetition zero or more times, whereas $[\ldots]^+_d$ denotes repetition one or more times with d as delimiter. E denotes a list of expressions.

7.2.1 Syntax

A syntax for behavioral interfaces is now introduced. Let Mtd denote the set of method names, v a program variable, and T a type. The type T may be either an interface or a data type.

Definition 1 A *method* is represented by a term

$$method(Name, Co, Inpar, Outpar, Body),$$

where $Name \in \mathsf{Mtd}$ is a method name, Co is an interface, $Inpar$ and $Outpar$ are lists of parameter declarations of the form $v : T$, and $Body$ is a pair $\langle Var, Code\rangle$ consisting of a list $Vdecl$ of variable declarations (with initial expressions) and a list $Code$ of program statements. If Mtd is a set of methods, denote the subset of Mtd with methods of a given name by

$$Mtd(Name) = \{method(Name, Co, Inpar, Outpar, Body) \in Mtd\}.$$

Let $\mathcal{M}$ denote the set of method terms, and $\tau_{\mathcal{M}}$ the set of method names with typical element m. For convenience, the elements of a method tuple may be accessed by dot notation. The symbol ε denotes the empty sequence (or list). To conveniently organize object viewpoints, interfaces are structured in an inheritance hierarchy.

Definition 2 An *interface* is represented by a term

$$interface(Inh, Mtd)$$

of type I, where Inh is a list of interfaces, defining inheritance, and Mtd is a set of methods such that $m.Body = \langle\varepsilon, \varepsilon\rangle$ for all $m \in Mtd$.

Let τ_I denote the set of interface names, with typical elements I and J. Names are bound to interface terms in the typing environment. If I inherits J, the methods declared in both I and J must be available in any class that implements I. Dot notation may be used to refer to the different elements of an interface; e.g., $interface(Is, M).Mtd = M$. The name $Any \in \tau_I$ is reserved for $interface(\varepsilon, \emptyset)$, and the name $\varsigma \in \tau_I$ is reserved for type checking purposes (ς is used as the cointerface for purely internal calls, see page 57). An abstract representation of an interface may be given following the syntax of Figure 7.1. In the abstract representation all methods of an interface have the same cointerface, declared in a **with**-clause, encouraging an aspect-oriented specification style [51].

Even if two interfaces have the same set of methods, it may be undesirable to (accidentally) identify them. Consequently, we use a nominal subtype relation [68]. An interface is a subtype of its inherited interfaces. The subtype relation may also be explicitly extended by subtype declarations. The extension of this notion of interface with semantic constraints on the observable communication history and the refinement of such interfaces is studied in [45, 46].

7.2.2 Example

In order to illustrate the interface notion and pave the way for future examples, we consider the interfaces of a node in a peer-to-peer file sharing network. A *Client* interface captures the client end of the node, available to any user of the system. It offers methods to list all files available in the network and to request the download of a given file from a given server. A *Server* interface offers a method for obtaining a list of files available from the node and a method for downloading packets, i.e., parts of a target file. The *Server* interface is only available to other servers in the network. Due to the cointerface, type soundness [68] will guarantee that any caller of a server request understands the *enquire* and *getPacket* methods. The two interfaces may be inherited by a third interface *Peer* which describes nodes that are able to act according to both the client role and the server role. In the *Peer* interface, the cointerface requirement of each superinterface restricts the use of the methods inherited from that superinterface. For simplicity method signatures are omitted here, these are discussed in Section 7.3.3.

```
interface Client     interface Server     interface Peer
begin                begin                 inherits Client, Server
 with Any             with Server         begin
  op availFiles         op enquire        end
  op reqFile            op getLength
end                     op getPacket
                     end
```

7.3 Object Interaction by Asynchronous Method Calls

Inter-process communication is becoming increasingly important with the development of distributed computing, both over the Internet and over local networks. While object orientation is the leading framework for distributed and concurrent systems, standard models of object interaction seem less appropriate for distributed concurrent objects. To motivate Creol's asynchronous method calls, we give a brief review of the basic interaction models for concurrent processes with respect to distributed interaction.

The three basic interaction models for concurrent processes are shared variables, remote method calls, and message passing [6]. Shared memory models do not generalize well to distributed environments, so shared variables are discarded as inappropriate to capture object interaction in the distributed setting. With the *remote method invocation* (RMI) model, an object is activated by a method call. The thread of control is transferred with the call so there is a master-slave relationship between the caller and the callee. Caller activity is blocked until the return values from the method call have been received. A similar approach is taken with the execution threads of; e.g., Hybrid [65] and Java [37], where concurrency is achieved through multithreading. The interference problem related to shared variables reemerges when threads operate concurrently in the same object, which happens with non-synchronized methods in Java. Reasoning about programs in this setting is a highly complex matter [19, 2]: Safety is by convention rather than by language design [12]. Verification considerations therefore suggest that all methods should be synchronized, which is the approach taken in, e.g., Hybrid. However, when the language is restricted to synchronized methods, an object making a remote method call must *wait* for the return of the call before it can proceed with its activity. Consequently, any other activity in the object is prohibited while waiting. In a distributed setting this limitation is severe; delays and instabilities may cause much unnecessary waiting. A nonterminating method will even block the evaluation of other method activations, which makes it difficult to combine active and passive behavior in the same object.

In contrast to remote method calls, message passing is a communication form without any transfer of control between concurrent objects. A method call can here be modeled by an invocation and a reply message. Message passing may be synchronous, as in Ada's Rendezvous mechanism, in which case both the sender and receiver process must be ready before communication can occur. Hence, objects synchronize on message transmission. Remote method invocations may be captured in this model if the calling object blocks between the two synchronized messages representing the call [6]. If the calling object is allowed to proceed for a while before resynchronizing on the reply message we obtain a different model of method calls which from the caller perspective resembles *future variables* [82] (or eager invocation [29]). For distributed systems, even such synchronization must necessarily result in much waiting.

Message passing may also be asynchronous. In the asynchronous setting message emission is always possible, regardless of when the receiver accepts a message. Communication by asynchronous message passing is well-known from, e.g., the Actor model [3, 4]. Languages with notions of future variables are usually based on asynchronous message passing. In this case, the caller's activity is synchronized with the arrival of the reply message rather than with its emission, and the activities of the caller and the callee need not directly synchronize [82, 81, 17, 24, 9, 44]. This approach seems well-suited to model communication in distributed environments, reflecting the fact that communication in a network is not instantaneous. Asynchronous message passing, without synchronization and transfer of the thread of control, avoids unnecessary waiting in the distributed setting by providing better control and efficiency. Generative communication in, e.g., Linda [18] and Klaim [10] is an approach between shared variables and asynchronous message passing, where messages without an explicit destination address are shared on a possibly distributed blackboard. However, method calls imply an ordering on communication not easily captured in these models. Actors do not distinguish replies from invocations, so capturing method calls with Actors quickly becomes unwieldy [3]. Asynchronous message passing does not provide the structure and discipline inherent in method calls. The integration of the message concept in the object-oriented setting is unsettled, especially with respect to inheritance and redefinition. A satisfactory notion of method call for the distributed setting should be asynchronous, combining the advantages of asynchronous message passing with the structuring mechanism provided by the method concept. Such a notion is proposed in Creol's communication model.

7.3.1 Syntax

A simple language for concurrent objects is now presented, which combines so-called *processor release points* and *asynchronous method calls*. Processor release points influence the internal control flow in objects. This reduces time spent waiting for replies to method calls in the distributed setting and allows objects to dynamically change between active and reactive behavior.

At the imperative level, attributes and method declarations are organized in classes. Classes may have parameters [26] which can be data values or objects. Class parameters are similar to constructor parameters in Java [13], except that they form part of the state, making the assignment of parameter values to object attributes redundant. Objects are dynamically created instances of classes, their persistent state consists of declared class parameters and attributes. The state of an object is encapsulated and can only be accessed via the object's methods. Among the declared methods, we distinguish the method *run*, which is given a special treatment operationally. After initialization the *run* method, if provided, is started. Apart from *run*, declared methods may be invoked internally and by other objects supporting the appropriate interfaces. When called from other objects, these methods reflect reactive (or passive) behavior in the object, whereas *run* initiates active behavior. Methods need not terminate and all method activations may be temporarily *suspended*. The activation of a method results in a *process* executed in a Creol object. In fact, execution in a Creol object is organized around an unordered queue of processes competing for the object's processor.

In order to focus on the communication aspects of concurrent objects, we assume given a functional language for defining local data structures by means of data types and functions performing local computations on terms of such data types. Data types are built from basic data types by type constructors.

Definition 3 Let τ_B be a set of basic data types and τ_I a set of interface names, such that $\tau_B \cap \tau_I = \emptyset$. Let τ denote the set of all types including the basic and interface types; i.e., $\tau_B \subseteq \tau$ and $\tau_I \subseteq \tau$.

Let T_B and T be typical elements of τ_B and τ. The nominal subtype relation $\preceq$ is a reflexive partial ordering on types, including interfaces. We let $\perp$ represent an undefined (and illegal) type; thus for any type T we have $\neg(\perp \preceq T)$ and $\neg(T \preceq \perp)$. We denote by Data the supertype of both data and interface types. Apart from Data, a data type may only be a subtype of a data type and an interface only of an interface. Every interface is a subtype of *Any*, except ς which is only related to itself (i.e., $\varsigma \preceq \varsigma$). Nominal constraints restrict a structural subtype relation which ensures substitutability: if $T \preceq T'$ then any value of T may masquerade as a value of T' [55,13]. For product types R and R', $R \preceq R'$ is the point-wise extension of the subtype relation; i.e., R and R' have the same length l and $T_i \preceq T_i'$ for every i $(0 \leq i \leq l)$ and types T_i and T_i' in position i in R and R', respectively. To explain the typing and binding of methods, $\preceq$ is extended to function spaces $A \rightarrow B$, where A and B are (possibly empty) product types:

$$A \rightarrow B \preceq A' \rightarrow B' \Leftrightarrow A' \preceq A \wedge B \preceq B'$$

For types U and V, the intersection $T = U \cap V$ is such that $T \preceq U$, $T \preceq V$, and $T' \preceq T$ for all T' such that $T' \preceq U$ and $T' \preceq V$. (If no such T exists, $U \cap V = \perp$.) For every type T, we let

d_T denote the *default value* of T (e.g., d_I is called *null* for $I \in \tau_I$, d_{Nat} may be *zero*, etc).Type schemes such as parameterized data types may be applied to types in τ to form new types in τ. It is assumed in the examples of the sequel that τ_B includes standard types such as the Booleans Bool, the natural numbers Nat, and the strings Str, and that the type schemes include $\mathsf{Set}[T]$ and $\mathsf{List}[T]$. Expressions without side effects are given by a functional language $\mathcal{F}$ defined as follows:

Definition 4 Let $\mathcal{F}$ be a type sound functional language which consists of expressions $e \in \mathsf{Expr}$ constructed from

- constants of the types in τ_B,
- variables of the types in τ, and
- functions defined over terms of the types of τ.

In particular, ObjExpr and BoolExpr are subsets of Expr typed by interfaces and Booleans, respectively. There are no constructors or field access functions for terms of interface types, but object references may be compared by equality.

Assume given a typing environment $\Gamma_{\mathcal{F}}$ which provides the type information for the constants and functions in $\mathcal{F}$, and let Γ extend $\Gamma_{\mathcal{F}}$ with type information for variables. If d is a variable, constant, or function in $\mathcal{F}$, then $\Gamma(d)$ denotes the type of d in Γ. In particular, $\Gamma(d_T) = T$. For $e \in \mathsf{Expr}$, $\Gamma \vdash_{\mathrm{F}} e : T$ denotes that e is type-correct and has type T in Γ (i.e., $T \neq \bot$). If e is type-correct in Γ and v is a variable occurring in e, then $\Gamma(v) \neq \bot$. Let Var be the type of variable names. Variable names are bound to actual values in a state.

Definition 5 Let $\sigma : \mathsf{Var} \mapsto \mathsf{Data}$ be a *state* with domain $\{v_1, \ldots, v_n\}$. If $\Gamma(\sigma(v_i)) \preceq \Gamma(v_i)$ for every i $(1 \leq i \leq n)$, then the state σ is *well-typed*.

The *evaluation* of an expression $e \in \mathsf{Expr}$, relative to a state σ, is denoted $eval(e, \sigma)$. It is assumed in Definition 4 that $\mathcal{F}$ is *type sound* [68]: well-typed expressions remain well-typed during evaluation. Technically, the type soundness of $\mathcal{F}$ is given as follows: Let $\vec{v}$ be the variables in an expression $e \in \mathsf{Expr}$ and assume that σ is a well-typed state defined for all $v \in \vec{v}$. If $\Gamma \vdash_{\mathrm{F}} e : T$ then $\Gamma \vdash_{\mathrm{F}} eval(e, \sigma) : T'$ such that $T' \preceq T$.

The assumption of type soundness for $\mathcal{F}$ leads to a restricted use of partial functions. For instance, one may adapt the order-sorted approach [35] where partial functions are allowed by identifying the subdomains in which they give defined values, and require that each application of a partial function is defined. In this approach, the head of an empty list would not be type-correct, but the head of a list suffixed by an element would be type-correct.

Classes and objects

An object-oriented language is now constructed, extending the functional language $\mathcal{F}$. Classes are defined in a traditional way, including declarations of persistent state variables and method definitions.

$$\begin{array}{lcl}
CL & ::= & [\mathbf{class}\ C\ [(\mathit{Param})]^{?}\ [\mathbf{contracts}\ [I]_{,}^{+}]^{?}\ [\mathbf{implements}\ [I]_{,}^{+}]^{?} \\
 & & \quad [\mathbf{inherits}\ [C[(\mathrm{E})]^{?}]_{,}^{+}]^{?}\ \mathbf{begin}\ [\mathbf{var}\ \mathit{Vdecl}]^{?}\ [[\mathbf{with}\ I]^{?}\ \mathit{Mdecls}]^{*}\ \mathbf{end}]^{*} \\
\mathit{Vdecl} & ::= & [v : T[=e]^{?}]_{;}^{+} \\
\mathit{Mdecls} & ::= & [\mathit{Msig} == [\mathbf{var}\ \mathit{Vdecl};]^{?}\ \mathrm{S}]^{+}
\end{array}$$

Figure 7.2: An syntax outline for the abstract representation of classes, excluding expressions e, expression lists E, and statement lists S (which are defined in Figure 7.3).

Definition 6 A *class* is represented by a term

$$\mathit{class}\,(\mathit{Param}, \mathit{Impl}, \mathit{Contract}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd}),$$

where *Param* is a list of typed program variables, *Contract* and *Impl* are lists of interface names, *Inh* is a list of instantiated class names, defining class inheritance, *Var* is a list of typed program variables with initial expressions, and *Mtd* is a set of methods.

Each method is equipped with an element *Co* specifying the cointerface associated with the method (Definition 1). For purely internal methods, the cointerface element contains the special name ς. Notice that *Impl* represents interfaces implemented by the class, whereas *Contract* represents interfaces implemented by the class and all subclasses. Thus *Contract* claims are inherited by subclasses, but *Impl* claims are not. A class C is said to contract an interface I if a subinterface of I appears in the *Contract* clause of C or of a superclass of C. The typing of remote method calls in a class C relies on the fact that the calling object supports the contracted interfaces of C, and these are used to check the cointerface requirements of the calls.

Let $\mathcal{C}$ denote the set of class terms, and τ_C the set of class names with typical element C. In the typing environment, class names are bound to class terms. For convenience, dot notation is used to denote the different elements of a class; e.g., *Cl.Var* denotes the variable list of a class *Cl*. An abstract representation of a class may be given following the syntax of Figure 7.2. Variable declarations are defined as a sequence *Vdecl* of statements $v : T$ or $v : T = e$, where v is the name of the attribute, T its type, and e an optional expression providing an initial value for v. This expression may depend on the actual values of the class parameters. A statement $v : T$ without an initial expression is initialized to the default value d_T. Overloading of methods is allowed. The pseudo-variable *self* is used for self reference in the language; its value cannot be modified. Issues related to inheritance are considered in Section 7.5; until then, we consider classes without explicit inheritance.

An object offers methods to its environment, specified through a number of interfaces. All interaction between objects happens through method calls. In the asynchronous setting method calls can always be emitted, because the receiving object cannot block communication. *Method overtaking* is allowed: if methods offered by an object are invoked in one order, the object may evaluate the corresponding method activations in another order. A method activation is, roughly speaking, a list S of program statements evaluated in the context of a state. Due to the possible interleavings of different method executions, the values of an object's program variables are not entirely controlled by a method activation which suspends itself before completion. However, a method may have local variables supplementing the object attributes. In particular, the values of formal parameters are stored locally, but other local variables may also be created. Among

Syntactic categories.		*Definitions.*
g in Guard	v in Var	$g ::= wait \mid b \mid t? \mid g_1 \wedge g_2 \mid g_1 \vee g_2$
t in Label	s in Stm	$r ::= x.m \mid m$
m in Mtd	r in MtdCall	$\mathrm{S} ::= s \mid s; \mathrm{S}$
e in Expr	b in BoolExpr	$s ::= \mathbf{skip} \mid (\mathrm{S}) \mid \mathrm{V} := \mathrm{E} \mid v := \mathbf{new}\, C(\mathrm{E})$
x in ObjExpr		$\mid r(\mathrm{E};\mathrm{V}) \mid !r(\mathrm{E}) \mid t!r(\mathrm{E}) \mid t?(\mathrm{V}) \mid \mathbf{await}\, g$

Figure 7.3: An outline of the language syntax for program statements, with typical terms for each syntactic category. Capitalized terms such as V, S, and E denote lists, sets, or multisets of the given syntactic categories, depending on the context.

the local variables of a method, certain variables are used to organize inter-object communication; there is read-only access to *caller* and *label*. Assignment to local and object variables is expressed as $\mathrm{V} := \mathrm{E}$ for (the same number of) program variables V and expressions E. In the object creation statement $v := \mathbf{new}\, C(\mathrm{E})$, v must be a variable declared of an interface implemented by C and E are actual values for the class parameters. We refer to $C(\mathrm{E})$ as the *instantiated class name*. The syntax for program statements is given in Figure 7.3.

Let Label denote the type of method call identifiers, partially ordered by $<$ and with least element 1, and let the operation $next$: Label $\rightarrow$ Label be such that $\forall x \in \text{Label}.\, x < next(x)$. A method is *asynchronously* invoked with the statement $t!x.m(\mathrm{E})$, where $t \in$ Label provides a locally unique reference to the call, x is an object expression, m a method name, and E an expression list with the actual in-parameters supplied to the method. The call is *internal* when x is omitted, otherwise the call is *external*. A method with ς as cointerface may only be called internally. Labels are used to identify replies and may be omitted if a reply is not explicitly requested. As no synchronization is involved, process execution can proceed after calling a method until the return value from the method is actually needed by the process.

To fetch the return values from a call, say in a variable list V, we may ask for the reply to our call: $t?(\mathrm{V})$. This statement treats V as a list of future variables. If the reply to the call has arrived, return values may be assigned to V and the execution continues without delay. If the reply has not arrived, process execution is *blocked* at this statement. In order to avoid blocking in the asynchronous case, processor release points are introduced by means of reply guards. In this case, process execution is *suspended* rather than blocked.

Any method may be invoked in a synchronous as well as an asynchronous manner. *Synchronous* (RMI) method calls are given the syntax $x.m(\mathrm{E};\mathrm{V})$, which is defined by $t!x.m(\mathrm{E}); t?(\mathrm{V})$ for some fresh label t, *immediately* blocking the processor while waiting for the reply. This way the call is perceived as synchronous by the caller, although the interaction with the callee is in fact asynchronous. The callee does not distinguish synchronous and asynchronous invocations of its methods. It is clear that in order to reply to local calls, the calling method must eventually suspend its own execution. A local call may be either internal, or external if the callee is equal to *self*. Therefore, the reply statement $t?(\mathrm{V})$ enables execution of the call identified by t when this call is local. The language does not otherwise support monitor reentrance; mutual or cyclic synchronous calls between objects may therefore lead to deadlock.

Potential suspension is a expressed through *processor release points*, a basic programming construct in the language, using guard statements [30]. In Creol, guards influence the control flow between processes inside concurrent objects. A guard g is used to explicitly declare a potential release point for the object's processor with the statement **await** g. Guard statements can be nested within a method body, corresponding to a series of potential suspension points. Let S_1 and S_2 denote statement lists; in $\mathrm{S}_1;$**await** $g;\mathrm{S}_2$ the guard g corresponds to an inner release point. A guard statement is *enabled* if its guard evaluates to true. When an inner guard which is not enabled is encountered during process execution, the process is suspended and the processor released. The *wait* guard is a construct for explicit release of the processor. The reply guard $t?$ is enabled if the reply to the method invocation with label t has arrived. Guards may be composed: $g_1 \wedge g_2$ is enabled if both g_1 and g_2 are enabled. The evaluation of guard statements is atomic. After process suspension, the object's suspended processes compete for the free processor: *any* suspended and enabled process may be selected for execution. For convenience, we introduce the following abbreviations:

await $t?(\mathrm{V}) =$ **await** $t?; t?(\mathrm{V})$
await $r(\mathrm{E};\mathrm{V}) = t!r(\mathrm{E});$ **await** $t?(\mathrm{V})$, where t is a fresh label.

Using reply guards, the object processor need not block while waiting for replies. This approach is more flexible than future variables: suspended processes or new method activations may be evaluated while waiting for a reply. If the called object does not eventually reply, deadlock is avoided in the sense that other activity in the object is possible although the process itself will a priori remain suspended. However, when a reply arrives, the *continuation* of the original process must compete with other enabled suspended processes.

7.3.2 Virtual Binding

Due to the interface typing of object variables, the actual class of the receiver of an external call is not statically known. Consequently, external calls are virtually bound.

Let the function *Sig* give the signature of a method, defined by $Sig(m) = type(m.Inpar) \rightarrow type(m.Outpar)$ where *type* returns the product of the types in a parameter declaration. The static analysis of a synchronous internal call $m(\mathrm{E};\mathrm{V})$ assigns unique types to the in- and out-parameter depending on the textual context, say that the parameters are textually declared as $\mathrm{E} : T_{\mathrm{E}}$ and $\mathrm{V} : T_{\mathrm{V}}$. The call is *type-correct* if there is a method declaration $m : T_1 \rightarrow T_2$ in the class C such that $T_1 \rightarrow T_2 \preceq T_{\mathrm{E}} \rightarrow T_{\mathrm{V}}$. A synchronous external call $o.m(\mathrm{E};\mathrm{V})$ to an object o of interface I is type-correct if it can be bound to a method declaration in I in a similar way. The static analysis of a class verifies that it implements the methods declared in its interfaces. Assuming that any object variable typed by an interface I points to an instance of a class implementing I, method binding will succeed regardless of the actual class of the object. At runtime, the class of the object is dynamically identified and the method is virtually bound. Remark that if the method is overloaded; i.e., there are several methods with the same name in the class, the types of the actual parameter values, including the out-parameter, and the actual cointerface are used to correctly bind the call (see Section 7.5).

Asynchronous calls may be bound in the same way, provided that the type of the actual parameter values and cointerface can be determined. In the operational semantics of Creol, it is assumed that this type information is included at compile-time in both synchronous and

```
class Node (db:DB)
  implements Peer
begin
with Server
  op enquire(out files:List[Str]) == await db.listFiles(ε;files)
  op getLength(in fId:Str out lth:Nat) == await db.getLength(fId;lth)
  op getPacket(in fId:Str; pNbr:Nat out packet:List[Data])==
    var f:List[Data];await db.getFile(fId;f); packet:=f[pNbr]
with Any
  op availFiles (in sList:List[Server] out files:List[Server×Str])==
    var l1:Label; l2:Label; fList:List[Str];
    if (sList = empty) then files:= empty
    else l1!hd(sList).enquire(); l2!this.availFiles(tl(sList));
      await l1?∧l2?; l1?(fList); l2?(files); files:=((hd(sList),fList); files) fi
  op reqFile(in sId:Server out fId:Str)==
    var file:List[Data]; packet:List[Data]; lth:Nat;
    await sId.getLength(fId;lth); while (lth>0) do
      await sId.getPacket(fId, lth; packet); file:=(packet;file); lth:=lth - 1 od;
    !db.storeFile(fId, file)
end
```

Figure 7.4: A class capturing nodes in a peer-to-peer network.

asynchronous method invocations.

7.3.3 Example

A peer-to-peer file sharing system consists of nodes distributed across a network. Peers are equal: each node plays both the role of a server and of a client. In the network, nodes may appear and disappear dynamically. As a client, a node requests a file from a server in the network and downloads it as a series of packet transmissions until the file download is complete. The connection to the server may be blocked, in which case the download automatically resumes if the connection is reestablished. A client may run several downloads concurrently, at different speeds. We assume that every node in the network has an associated database with shared files. Downloaded files are stored in the database, which implements the interface *DB* and is not modeled here:

```
interface DB
begin
with Server
  op getFile(in fId:Str out file:List[List[Data]])
  op getLength(in fId:Str out length:Nat)
  op storeFile(in fId:Str; file:List[Data])
  op listFiles(out fList:List[Str])
end
```

Here, *getFile* returns a list of packets; i.e., a sequence of sequences of data, for transmission over the network, *getLength* returns the number of such sequences, *listFiles* returns the list of available files, and *storeFile* adds a file to the database, possibly overwriting an existing file.

Nodes in the peer-to-peer network which implement the *Peer* interface can be modeled by a class *Node*, given in Figure 7.4. *Node* objects can have several interleaved activities: several downloads may be processed simultaneously as well as uploads to other servers, etc. All method calls are asynchronous: If a server temporarily becomes unavailable, the transaction is suspended and may resume at any time after the server becomes available again. Processor release points ensure that the processor will not be blocked and transactions with other servers not affected. In the class, the method *availFiles* returns a list of pairs where each pair contains a file identifier *fId* and the server identifier *sId* where *fId* may be found, *reqFile* the file associated with *fId*, *enquire* the list of files available from the server, and *getPacket* a particular packet in the transmission of a file. The list constructor is represented by semicolon. For $x : T$ and $\mathrm{s} : \mathsf{List}[T]$, we let $hd(x;\mathrm{s}) = x$, $tl(x;\mathrm{s}) = \mathrm{s}$, and $\mathrm{s}[i]$ denote the i'th element of s, provided $i \leq length(\mathrm{s})$.

7.3.4 Typing

The type analysis of statements and declarations is formalized by a deductive system for judgments of the form

$$\Gamma \vdash_i D \langle \Delta \rangle,$$

where Γ is the typing environment, $i \in \{\mathrm{S}, \mathrm{V}\}$ specifies the syntactic category (Stm or Var, respectively), D is a Creol construct (statement or declaration), and Δ is the *update* of the typing environment. The typing judgment means that D contains no type errors when checked with the environment Γ. The typing environment resulting from the type analysis of D becomes Γ overridden by Δ, denoted $\Gamma + \Delta$. The rule for sequential composition SEQ is captured by

$$\text{(SEQ)} \quad \frac{\Gamma \vdash_i D \langle \Delta \rangle \qquad \Gamma + \Delta \vdash_i D' \langle \Delta' \rangle}{\Gamma \vdash_i D; D' \langle \Delta + \Delta' \rangle}$$

where $+$ is an associative operator on mappings with the identity element $\emptyset$. We abbreviate $\Gamma \vdash_i D \langle \emptyset \rangle$ to $\Gamma \vdash_i D$.

For our purpose, the typing environment Γ is given as a *family of mappings*: $\Gamma_{\mathcal{F}}$ describes the constants and operators of $\mathcal{F}$, Γ_I the binding of interface names to interface terms, Γ_C the binding of class names to class terms, Γ_{V} the binding of program variable names to types, the mapping Γ_P is related to the binding of asynchronous internal and external method calls, and Γ_{Sig} stores derived actual signatures for asynchronous method invocations. Remark that Γ_I and Γ_C correspond to static tables. Declarations may only update Γ_{V} and program statements may not update Γ_{V}. Mapping families are now formally defined.

Definition 7 Let n be a name, d a declaration, $i \in \{I, \mathcal{C}, V, P, Sig\}$ a mapping index, and $[n \overset{i}{\mapsto} d]$ the binding of n to d indexed by i. A *mapping family* Γ is built from the empty mapping family $\emptyset$ and indexed bindings by the constructor $+$. The mapping with index i is extracted from Γ as follows

$$\begin{array}{lcl}\emptyset_i & = & \varepsilon \\ (\Gamma + [n \overset{i'}{\mapsto} d])_i & = & \textbf{if } i = i' \textbf{ then } \Gamma_i + [n \overset{i}{\mapsto} d] \textbf{ else } \Gamma_i.\end{array}$$

For an indexed mapping Γ_i, mapping application is defined by

$$\begin{array}{lcl}\varepsilon(n) & = & \bot \\ (\Gamma_i + [n \overset{i}{\mapsto} d])(n') & = & \textbf{if } n = n' \textbf{ then } d \textbf{ else } \Gamma_i(n').\end{array}$$

Typing of Programs

A class or interface declaration binds a name to a class or interface term, respectively. Class and interface names need not be distinct. A program consists of interface and class declarations, represented by the mappings $\Gamma_{\mathcal{I}} : \tau_{\mathcal{I}} \to \mathcal{I}$ and $\Gamma_{\mathcal{C}} : \tau_{\mathcal{C}} \to \mathcal{C}$, and an initial object creation message **new** $C(\mathrm{E})$. In a nominal type system, each interface and class of a program is type checked in the context of the mappings $\Gamma_{\mathcal{F}}$, $\Gamma_{\mathcal{I}}$, and $\Gamma_{\mathcal{C}}$.

$$\text{(PROG)}\quad \frac{\begin{array}{c}\forall I \in \tau_{\mathcal{I}} \cdot \Gamma_{\mathcal{I}} \vdash \Gamma_{\mathcal{I}}(I) \qquad \Gamma_{\mathcal{F}} + \Gamma_{\mathcal{C}} \vdash_{\mathrm{S}} \textbf{new}\, C(\mathrm{E}) \\ \forall C \in \tau_{\mathcal{C}} \cdot \Gamma_{\mathcal{F}} + \Gamma_{\mathcal{I}} + \Gamma_{\mathcal{C}} + [\mathit{self} \overset{\mathrm{V}}{\mapsto} C] \vdash \Gamma_{\mathcal{C}}(C) \ \langle\, \Delta^c \,\rangle\end{array}}{\Gamma_{\mathcal{F}} \vdash \Gamma_{\mathcal{I}}, \Gamma_{\mathcal{C}}, \textbf{new}\, C(\mathrm{E}) \ \langle\, \bigcup_{C \in \tau_{\mathcal{C}}} \Delta^c_{\mathit{Sig}} \,\rangle}$$

When type checking a class, *self* is bound to the class name. Type checking a program succeeds if all interfaces are well-formed, all classes are type-correct, and the initial object creation message is type-correct in the context of the program's class declarations. In order to focus on the type checking of classes, the method set *Mtd* of interface terms here includes both locally declared and inherited methods. The rule for type checking interface declarations may now be given as follows:

$$\text{(INTERFACE)}\quad \frac{\forall m \in \mathit{Mtd} \cdot \Gamma \vdash_{\mathrm{V}} m.\mathit{Inpar}; m.\mathit{Outpar} \ \langle\, \Delta \,\rangle}{\Gamma \vdash \mathit{interface}(\mathit{Inh}, \mathit{Mtd})}$$

The type checking of classes is now considered in detail. The rule for type checking class declarations is given as follows:

$$\text{(CLASS)}\quad \frac{\begin{array}{l}\Gamma \vdash_{\mathrm{V}} \mathit{Param}; \mathit{Var} \ \langle\, \Delta \,\rangle \qquad \forall m \in \mathit{Mtd} \cdot \Gamma + \Delta \vdash m \ \langle\, \Delta^m \,\rangle \\ \forall I \in (\mathit{Impl}; \mathit{Contract}) \cdot \forall m' \in \Gamma_{\mathcal{I}}(I).\mathit{Mtd} \cdot \exists m \in \mathit{Mtd} \cdot \\ \qquad m.\mathit{Name} = m'.\mathit{Name} \wedge \mathit{Sig}(m) \preceq \mathit{Sig}(m') \wedge m'.\mathit{Co} \preceq m.\mathit{Co}\end{array}}{\Gamma \vdash \mathit{class}(\mathit{Param}, \mathit{Impl}, \mathit{Contract}, \varepsilon, \mathit{Var}, \mathit{Mtd}) \ \langle\, \bigcup_{m \in \mathit{Mtd}} \Delta^m_{\mathit{Sig}} \,\rangle}$$

A class may implement a number of interfaces. For each interface, the class must provide methods with signatures that are correct with respect to the method signatures of the interface. The class' variable declarations are type checked after extending the typing environment with the class parameters, because the variable declarations may include initial expressions that use these parameters. Before type checking the methods, the typing environment is extended with the declared parameters and variables of the class. Method bodies are type checked in the typing environment updated after type checking method parameters and local attributes, including *caller*. At this point Γ_P is empty, since no asynchronous method invocation has been encountered.

$$\text{(METHOD)}\quad \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{V}} (\mathit{caller} : \mathit{Co}); \mathit{Inpar}; \mathit{Outpar}; \mathit{Body.Var} \langle \Delta \rangle \\ \Gamma + \Delta \vdash_{\mathrm{S}} \mathit{Body.Code} \langle \Delta' \rangle\end{array}}{\Gamma \vdash \mathit{method}(\mathit{Name}, \mathit{Co}, \mathit{Inpar}, \mathit{Outpar}, \mathit{Body}) \langle \Delta'_{\mathit{Sig}} \rangle}$$

In order to use self reference in expressions inside classes, we introduce qualified self references by the keyword **qua**, which uniquely controls the typing in case the class has many contracts.

$$\text{(SELF-REF)}\quad \frac{\exists I' \in (\Gamma_C(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Contract}; \mathsf{Any}) \cdot I' \preceq I}{\Gamma \vdash_{\mathrm{F}} \mathit{self}\ \mathbf{qua}\ I : I}$$

Typing of Parameter and Variable Declarations

A method may have local variable declarations preceding the program statements. Thus, both class and local variable declarations may extend the typing environment provided that these are not previously declared in the typing environment. The typing rules for variable and parameter declarations are given below.

$$\text{(VAR-AX)}\quad \Gamma \vdash_{\mathrm{V}} \varepsilon \qquad\qquad \text{(PAR)}\quad \frac{\Gamma_{\mathrm{V}}(v) = \perp \quad T \preceq \mathsf{Data} \quad v \neq \mathit{label}}{\Gamma \vdash_{\mathrm{V}} v : T \langle [v \overset{\mathrm{V}}{\mapsto} T] \rangle}$$

$$\text{(VAR)}\quad \frac{\Gamma \vdash_{\mathrm{V}} v : T \langle \Delta \rangle \quad \Gamma \vdash_{\mathrm{F}} e : T' \quad T' \preceq T}{\Gamma \vdash_{\mathrm{V}} v : T = e \langle \Delta \rangle}$$

Typing of Basic Statements

The typing of basic statements is given in Figure 7.5, as well as a statement for object creation. The typing system for $\mathcal{F}$ is used to type check expressions. The last premise of NEW ensures that the new object implements an interface which is a subtype of the declared interface of the variable v, extending rule NEW2 for initial object creation messages.

$$\text{(SKIP)}\quad \Gamma \vdash_{\mathrm{S}} \mathbf{skip} \qquad\qquad \text{(NEW2)}\quad \frac{\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T \quad T \preceq \mathit{type}(\Gamma_C(C).\mathit{Param})}{\Gamma \vdash_{\mathrm{S}} \mathbf{new}\ C(\mathrm{E})}$$

$$\text{(EMPTY)}\quad \Gamma \vdash_{\mathrm{S}} \varepsilon$$

$$\text{(AWAIT-}b\text{)}\quad \frac{\Gamma \vdash_{\mathrm{F}} b : \mathsf{Bool}}{\Gamma \vdash_{\mathrm{S}} \mathbf{await}\ b} \qquad\qquad \text{(AWAIT-}\wedge\text{)}\quad \frac{\Gamma \vdash_{\mathrm{S}} \mathbf{await}\ g_1 \quad \Gamma \vdash_{\mathrm{S}} \mathbf{await}\ g_2}{\Gamma \vdash_{\mathrm{S}} \mathbf{await}\ g_1 \wedge g_2}$$

$$\text{(AWAIT-}\vee\text{)}\quad \frac{\Gamma \vdash_{\mathrm{S}} \mathbf{await}\ g_1 \quad \Gamma \vdash_{\mathrm{S}} \mathbf{await}\ g_2}{\Gamma \vdash_{\mathrm{S}} \mathbf{await}\ g_1 \vee g_2} \qquad\qquad \text{(ASSIGN)}\quad \frac{\begin{array}{cc}\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T' & T' \preceq \Gamma_{\mathrm{V}}(\mathrm{V}) \\ \Gamma_{\mathrm{V}}(\mathrm{V}) \neq \mathsf{Label} & \mathrm{V} \neq \mathit{caller}\end{array}}{\Gamma \vdash_{\mathrm{S}} \mathrm{V} := \mathrm{E}}$$

$$\text{(NEW)}\quad \frac{\Gamma \vdash_{\mathrm{S}} \mathbf{new}\ C(\mathrm{E}) \qquad \exists I \in (\Gamma_C(C).\mathit{Impl}; \Gamma_C(C).\mathit{Contract}) \cdot I \preceq \Gamma_{\mathrm{V}}(v)}{\Gamma \vdash_{\mathrm{S}} v := \mathbf{new}\ C(\mathrm{E})}$$

Figure 7.5: Typing of basic statements.

Typing of Asynchronous Calls

In order to successfully bind method calls, the types of the formal parameters and cointerface of the declared method must correspond to the types of the actual parameters and cointerface of the call. This is verified by a predicate *match*, defined as follows:

Definition 8 Let T and U be types, I an interface, m a method name, and M a set of methods. Define $match : \tau_{\mathcal{M}} \times \tau \times \tau_I \times \mathsf{Set}[\mathcal{M}] \to \mathsf{Bool}$ by

$$\begin{aligned}&\text{match}(m, T \to U, I, \emptyset) = \mathsf{false}\\ &\text{match}(m, T \to U, I, \{m'\} \cup \text{M}) = (m = m'.Name \wedge I \preceq m'.Co \wedge Sig(m') \preceq T \to U).\end{aligned}$$

If a call has a match in an interface or class, we say that the call is covered:

Definition 9 An external method call is *covered* in an interface if there is a method declaration in the interface which may be type-correctly bound to the call, including the actual parameter types of output variables. An internal method call is *covered* in a class if there is a method declaration in the class, or a superclass, which may be type-correctly bound to the call, including the actual parameter types of output variables.

For synchronous calls, it is straightforward to check whether a call is covered, since the types of the actual in- and out-parameters can be derived directly from the textual invocation. In contrast, checking whether asynchronous calls are covered is more involved, since the type information provided by the textual invocation is not sufficient: the correspondence between in- and out-parameters is controlled by label values. The increased freedom in the language gained from using labels, requires a more sophisticated type analysis. In order to use the type system to derive signatures for asynchronous method invocations, it is assumed in method bodies that every asynchronous invocation using a label t is uniquely indexed; e.g., $t!r(\text{E})$ is transformed by the parser into $t_i!r(\text{E})$ for a fresh index i. The mappings $\Gamma_P : \mathsf{Label} \to \mathsf{Set}[\mathsf{Nat}]$ and $\Gamma_{Sig} : \mathsf{Label}_{\mathsf{Nat}} \to \tau_I \times \tau_{\mathcal{M}} \times \tau_I \times \tau$ are used as an effect system [56, 75] for type checking asynchronous calls. The mapping Γ_P maps labels to *sets* of indices, uniquely identifying the occurrences of calls corresponding to a label. The mapping Γ_{Sig} maps indexed labels to tuples containing the information needed to later refine the type analysis of the method calls associated with the indexed labels. For a label t and any $i \in \Gamma_P(t)$, t is the label associated with the i'th internal or external asynchronous *pending* call for which the types of the out-parameters have yet to be resolved. After the out-parameters have been resolved, $\Gamma_{Sig}(t_i)$ is updated with the refined actual signature of the call associated with the indexed label t_i. Hence, Γ_{Sig} provides the interface of the callee and the actual signature for the asynchronous calls in the code. Both mappings Γ_P and Γ_{Sig} are part of the typing environment used for type checking method bodies.

External and Internal Invocations and Replies

The typing rules for external and internal invocations and replies are given in Figure 7.6. As the types of the actual in- and out-parameters of every synchronous call can be derived immediately, the type system can directly decide if the call is type-correct. Asynchronous calls without labels can also be type checked directly, as the reply values cannot subsequently be requested. Consequently, the type of any formal out-parameter is type checked against the supertype Data. For an external asynchronous invocation with indexed label t_i, the type of the return values is yet unknown. Type checking with the exact type of the out-variables must be *postponed* until the corresponding reply statement for t is eventually analyzed. Therefore, i is added to the

$$\text{(EXT-SYNC)}\quad \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{F}} e : I \qquad \Gamma \vdash_{\mathrm{F}} \mathrm{E} : T \\ Co \in \overline{Contract} \quad \mathrm{match}(m, T \to \Gamma_{\mathrm{V}}(\mathrm{V}), Co, \Gamma_I(I).Mtd)\end{array}}{\Gamma \vdash_{\mathrm{S}} e.m(\mathrm{E};\mathrm{V})}$$

$$\text{(EXT-ASYNC)}\quad \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{F}} e : I \qquad \Gamma \vdash_{\mathrm{F}} \mathrm{E} : T \\ Co \in \overline{Contract} \quad \mathrm{match}(m, T \to \mathsf{Data}, Co, \Gamma_I(I).Mtd)\end{array}}{\Gamma \vdash_{\mathrm{S}} !e.m(\mathrm{E})}$$

$$\text{(EXT-ASYNC-L)}\quad \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{F}} e : I \qquad \Gamma \vdash_{\mathrm{F}} \mathrm{E} : T \qquad \Gamma_{\mathrm{V}}(t) = \mathsf{Label} \\ Co \in \overline{Contract} \quad \mathrm{match}(m, T \to \mathsf{Data}, Co, \Gamma_I(I).Mtd)\end{array}}{\Gamma \vdash_{\mathrm{S}} t_i!e.m(\mathrm{E}) \ \langle\ [t \overset{P}{\mapsto} \{i\}] + [t_i \overset{Sig}{\mapsto} \langle I, m, Co, T \to \mathsf{Data}\rangle]\ \rangle}$$

$$\text{(INT-SYNC)}\quad \frac{\mathrm{match}(m, T \to \Gamma_{\mathrm{V}}(\mathrm{V}), \varsigma, \overline{Mtd}) \qquad \Gamma \vdash_{\mathrm{F}} \mathrm{E} : T}{\Gamma \vdash_{\mathrm{S}} m(\mathrm{E};\mathrm{V})}$$

$$\text{(INT-ASYNC)}\quad \frac{\mathrm{match}(m, T \to \mathsf{Data}, \varsigma, \overline{Mtd}) \qquad \Gamma \vdash_{\mathrm{F}} \mathrm{E} : T}{\Gamma \vdash_{\mathrm{S}} !m(\mathrm{E})}$$

$$\text{(INT-ASYNC-L)}\quad \frac{\Gamma_{\mathrm{V}}(t) = \mathsf{Label} \quad \Gamma \vdash_{\mathrm{F}} \mathrm{E} : T \quad \mathrm{match}(m, T \to \mathsf{Data}, \varsigma, \overline{Mtd})}{\Gamma \vdash_{\mathrm{S}} t_i!m(\mathrm{E}) \ \langle\ [t \overset{P}{\mapsto} \{i\}] + [t_i \overset{Sig}{\mapsto} \langle \varsigma, m, \varsigma, T \to \mathsf{Data}\rangle]\ \rangle}$$

$$\text{(AWAIT-t?)}\quad \frac{\Gamma_P(t) \neq \emptyset \qquad \Gamma_P(t) \neq \perp}{\Gamma \vdash_{\mathrm{S}} \textbf{await}\, t?}$$

$$\text{(REPLY)}\quad \frac{\begin{array}{c}\Gamma_P(t) \neq \emptyset \qquad \Gamma_P(t) \neq \perp \\ \forall i \in \Gamma_P(t) \cdot \Gamma_{Sig}(t_i) = \langle I, m, Co, T_1 \to T_2\rangle \wedge (T_2 \sqcap \Gamma_{\mathrm{V}}(\mathrm{V})) \neq \perp \wedge \\ (I = \varsigma \Rightarrow \mathrm{match}(m, T_1 \to (T_2 \sqcap \Gamma_{\mathrm{V}}(\mathrm{V})), Co, \overline{Mtd})) \wedge \\ (I \neq \varsigma \Rightarrow \mathrm{match}(m, T_1 \to (T_2 \sqcap \Gamma_{\mathrm{V}}(\mathrm{V})), Co, \Gamma_I(I).Mtd))\end{array}}{\Gamma \vdash_{\mathrm{S}} t?(\mathrm{V}) \ \langle\ [t \overset{P}{\mapsto} \emptyset] + \bigcup_{i \in \Gamma_P(t)} [t_i \overset{Sig}{\mapsto} \langle I, m, Co, T_1 \to (T_2 \sqcap \Gamma_{\mathrm{V}}(\mathrm{V}))\rangle]\ \rangle}$$

Figure 7.6: Typing of external and internal invocation and reply statements. For an element F, $\overline{F}$ denotes $\Gamma_C(\Gamma_{\mathrm{V}}(\mathit{self})).F$, i.e., the corresponding element in the current class term.

set of pending calls $\Gamma_P(t)$ and the invocation is recorded in Γ_{Sig} with a mapping from t_i to the callee's interface, the method name, the caller's interface, and a preliminary actual signature for the call. Some erroneous invocations may already be eliminated by type checking against this preliminary signature.

For internal calls, the method must have cointerface ς. For external calls $x.m$, the interface of x must offer a method m with a cointerface *contracted* by the current class (and thereby supported by the actual calling object). This implies that *remote calls to self* are allowed when the class itself contracts an interface allowed as cointerface for the method.

A reply is requested through a reply statement or a guard, if there are pending invocations with the same label. For a reply statement $t?(\mathrm{V})$, the matching invocations must be type checked again as the types of the out-parameters V are now known. If the reply rule succeeds, all pending calls on t have been type checked against the actual type of V. This type checking depends on the interface of the callee for external calls and the class of *self* for internal calls. The pending calls on t are removed from Γ_P and the stored signatures for these calls in Γ_{Sig} are replaced by the refined signatures. Note that the type-correctness of pending calls without a corresponding reply statement is given directly by the typing rules for method invocations; i.e., the preliminary signature was sufficiently precise.

Example. Typing with labeled invocation and reply statements, where the binding depends on out-parameters, is illustrated by the following example.

```
interface A                  interface B                 interface AB
begin with B                 begin with A                inherits A, B
  op m(out x:Bool)             op m(out x:Nat)           begin end
end                          end
```

```
class C contracts AB
begin
  op run == var x:Nat; o:AB; t:Label; o:=new C(); t!o.m(); await t?; t?(x)
with B op m(out x:Bool) == x:=true
with A op m(out x:Nat) == x:=0
end
```

Here type checking succeeds, binding the call to m in *run* to interface B. Let t_1 be the (first) invocation with label t. The result of the type analysis is that $\Gamma_{Sig}(t_1) = \langle AB, m, AB, \varepsilon \rightarrow \mathsf{Nat}\rangle$. This information is passed on to the runtime system to ensure proper binding of the call. At runtime the call is then bound to the last declaration of m in C (see Section 7.3.5). In contrast, if the reply statement were removed from *run*, the result of the type analysis would be $\Gamma_{Sig}(t_1) = \langle AB, m, AB, \varepsilon \rightarrow Data\rangle$ and the call may be bound to either m.

Some initial properties of the type system are now presented.

Lemma 1 *No type checked list of program variable declarations dereference undeclared program variables.*

Proof. The proof is by induction over the length of a type checked list *Var* of program variable declarations. If $Var = \varepsilon$, no variables are dereferenced and the lemma holds. We now assume that no undeclared program variables are dereferenced in *Var* and show that this is also the case for $Var; v : T = e$ by considering the variables in e. Let $\Gamma' = \Gamma + \Delta$ such that $\Gamma \vdash_{\mathrm{V}} Var \langle \Delta \rangle$. Rule VAR requires that $\Gamma' \vdash_{\mathrm{F}} e : T'$. It follows from the type system for $\mathcal{F}$ that $T' \neq \bot$, so all variables in e have been declared in Γ'. ■

Lemma 2 *No type checked methods assign to or dereference undeclared program variables.*

Proof. We consider a method term $method(Name, Co, Inpar, Outpar, Body)$ type checked in the context of some class. The in- and out-parameters *Inpar* and *Outpar*, *label*, and *caller* do not have initial expressions, so they do not dereference any program variables. The method body *Body* consists of a list of local variable declarations *Var* and a list of program statements *Code*. It follows from Lemma 1 that *Var* does not dereference undeclared program variables. The proof proceeds by induction over the length of *Code*. Recall that if $\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T$ for some expression E, then $T \neq \bot$, so for all variables v in E, $\Gamma(v) \neq \bot$ and v is declared in Γ.

If $Code = \varepsilon$, the lemma trivially holds. For the induction step, we now assume that no undeclared program variables are assigned to or dereferenced in *Code* and show that this also holds for $Code; s$ by case analysis of s. Let $\Gamma = \Gamma' + \Delta$ such that $\Gamma' \vdash_{\mathrm{S}} Code \langle \Delta \rangle$.

- No program variables are assigned to or dereferenced by **skip**.
- For $\text{V} := \text{E}$, the ASSIGN rule asserts that $\Gamma \vdash_{\text{F}} \text{E} : T'$ and $T' \preceq \Gamma_{\text{V}}(\text{V})$. Since $T' \preceq \Gamma_{\text{V}}(\text{V})$, $\Gamma_{\text{V}}(\text{V}) \neq \bot$ and V is declared.
- For $x :=$ **new** $C(\text{E})$, the NEW rule asserts that $\Gamma \vdash_{\text{F}} \text{E} : T$ and that there exists an interface $I \in \Gamma_C(C).\mathit{Impl};\Gamma_C(C).\mathit{Contract}$ such that $I \preceq \Gamma_{\text{V}}(x)$. Consequently $\Gamma_{\text{V}}(x) \neq \bot$ and x is declared.
- For $x.m(\text{E};\text{V})$, rule EXT-SYNC asserts that $\Gamma \vdash_{\text{F}} x : I$, $\Gamma \vdash_{\text{F}} \text{E} : T$ and there is a cointerface $Co \in \Gamma_C(\Gamma_{\text{V}}(\mathit{self})).\mathit{Contract}$ such that $\mathit{match}(m, T \to \Gamma_{\text{V}}(\text{V}), Co, \Gamma_I(I).\mathit{Mtd})$. The match is impossible unless $\Gamma_{\text{V}}(\text{V}) \neq \bot$.
- For $!x.m(\text{E})$, rule EXT-ASYNC asserts that $\Gamma \vdash_{\text{F}} x : I$ and $\Gamma \vdash_{\text{F}} \text{E} : T$.
- For $t!x.m(\text{E})$, the additional condition $\Gamma_{\text{V}}(t) = \mathsf{Label}$ of rule EXT-ASYNC-L asserts that also t has been declared.
- For $m(\text{E};\text{V})$, rule INT-SYNC gives us the judgment $\Gamma \vdash_{\text{F}} \text{E} : T$ such that $\mathit{match}(m, T \to \Gamma_{\text{V}}(\text{V}), \varsigma, \Gamma_C(\Gamma_{\text{V}}(\mathit{self})).\mathit{Mtd})$. It follows that $\Gamma_{\text{V}}(\text{V}) \neq \bot$.
- For $!m(\text{E})$, rule INT-ASYNC asserts that $\Gamma \vdash_{\text{F}} \text{E} : T$.
- For $t!m(\text{E})$, the additional condition $\Gamma_{\text{V}}(t) = \mathsf{Label}$ of rule INT-ASYNC-L asserts that also t has been declared.
- For $t?(\text{V})$, rule REPLY asserts that $\Gamma_P(t) \neq \emptyset$, so there are pending calls on label t. Consequently there must be an invocation on t in *Code* which has been type checked by either EXT-ASYNC-L or INT-ASYNC-L. In both cases, the condition $\Gamma_{\text{V}}(t) = \mathsf{Label}$ guarantees that t has been declared. The matches for the pending calls on t imply that $\Gamma_{\text{V}}(\text{V}) \neq \bot$.
- For **await** g, induction over the construction of g shows that all variables are declared. The base cases are handled by AWAIT-b and AWAIT-$t?$.
- $\text{S}_1;\text{S}_2$ follows by the induction hypothesis. ■

Lemma 3 *If a synchronous call* $e.m(\text{E};\text{V})$ *or* $m(\text{E};\text{V})$ *is type-correct, then the corresponding asynchronous invocation* $!e.m(\text{E})$ *or* $!m(\text{E})$ *is also type-correct.*

Proof. Assume that $e.m(\text{E};\text{V})$ is type-correct. The rule EXT-SYNC asserts that there is an interface I, a cointerface $Co \in \Gamma_C(\Gamma_{\text{V}}(\mathit{self})).\mathit{Contract}$ and a type T such that $\Gamma \vdash_{\text{F}} e : I$, $\Gamma \vdash_{\text{F}} \text{E} : T$, and $\text{match}(m, T \to \Gamma_{\text{V}}(\text{V}), Co, \Gamma_I(I).\mathit{Mtd})$. It follows from the match that $\Gamma_{\text{V}}(\text{V}) \neq \bot$, so we have $\Gamma_{\text{V}}(\text{V}) \preceq \mathsf{Data}$ and consequently $\text{match}(m, T \to \mathsf{Data}, Co, \Gamma_I(I).\mathit{Mtd})$ holds. Rule EXT-ASYNC then asserts that the call $!e.m(\text{E})$ is type-correct. The case for the internal calls $m(\text{E};\text{V})$ and $!m(\text{E})$ is similar. ■

It follows from Lemma 3 that a minimal requirement for successful binding is that an invocation can be bound with Data as the type of the actual out-parameter. This is reflected in the typing rules. This minimal requirement is sufficient to show that the call may be bound correctly unless the return values from the asynchronous call are assigned to program variables.

Lemma 4 *Let Γ be a mapping family such that $\Gamma_P = \varepsilon$. For any statement list* S *with a type judgment* $\Gamma \vdash_{\mathrm{S}} \mathrm{S} \langle \Delta \rangle$*, the set* Δ_P *contains exactly the labeled method invocations for which the return values may still be assigned to program variables after* S*.*

Proof. The proof is by induction over the length of S. If $\mathrm{S} = \varepsilon$, we get $\Delta_P = \varepsilon$. Assume as induction hypothesis that for $\Gamma \vdash_{\mathrm{S}} \mathrm{S} \langle \Delta \rangle$, Δ_P contains the invocations for which the return values may still be assigned to program variables after S. For the induction step, we prove that for $\Gamma \vdash_{\mathrm{S}} \mathrm{S}; s \langle \Delta + \Delta' \rangle$, $(\Delta + \Delta')_P$ contains the method invocations for which the return values may still be assigned to program variables after $\mathrm{S}; s$, by case analysis of s.

- For **skip**, $\mathrm{V} := \mathrm{E}$, $x :=$ **new** $C(\mathrm{E})$, and **await** g, there are no new method calls, so $\Delta'_P = \varepsilon$ and $(\Delta + \Delta')_P$ contains exactly the method invocations that may need further analysis by the induction hypothesis.
- For $e.m(\mathrm{E}; \mathrm{V})$, $m(\mathrm{E}; \mathrm{V})$, $!e.m(\mathrm{E})$, and $!m(\mathrm{E})$, the type system allows the type-correctness of these statements to be verified directly. Consequently $\Delta'_P = \varepsilon$ and $(\Delta + \Delta')_P$ contains the method invocations for which the return values may still be assigned to program variables by the induction hypothesis.
- For $t_i!e.m(\mathrm{E})$, an eventual reply statement may later impose a restriction on the type of the out-values. Note that previous calls pending on t are no longer accessible to such a reply statement. The effect of the EXT-ASYNC-L rule records only the new call as pending on t, yielding $\Delta'_P = [t \overset{P}{\mapsto} \{i\}]$. It follows that $(\Delta + \Delta')_P$ contains exactly the method invocations for which the return values may still be assigned to program variables after $\mathrm{S}; t_i!e.m(\mathrm{E})$.
- For $t_i!m(\mathrm{E})$, the case is similar. The effect of the INT-ASYNC-L rule yields $\Delta'_P = [t \overset{P}{\mapsto} \{i\}]$. It follows that $(\Delta + \Delta')_P$ contains exactly the the method invocations for which the return values may still be assigned to program variables after $\mathrm{S}; t_i!m(\mathrm{E})$.
- For $t?(v)$ the REPLY rule states that $\Gamma_P(t) \neq \emptyset$ and $\Gamma_P(t) \neq \perp$, so there are pending calls to t in Δ_P which are type checked with the new out-parameter type and removed from Δ_P. The effect of REPLY is $\Delta'_P = [t \overset{P}{\mapsto} \emptyset]$ and it follows from the induction hypothesis that $(\Delta + \Delta')_P$ contains exactly the invocations for which the return values may be assigned to program variables after $\mathrm{S}; t?(\mathrm{V})$.
- $\mathrm{S}_1; \mathrm{S}_2$ follows from the induction hypothesis. ■

It follows that all asynchronous invocations can be precisely type checked.

Lemma 5 *In a well-typed program, all method invocations have been verified as type-correct by the type analysis.*

Proof. We consider method invocations in the code S of an arbitrary method body in a class of the program, with the type judgment $\Gamma \vdash_{\mathrm{S}} \mathrm{S} \langle \Delta \rangle$. As $\Gamma_P = \varepsilon$, Lemma 4 states that Δ_P contains exactly the method invocations that may need further type checking. As the entire method body S has been type checked, the invocations in Δ_P may be safely bound with the weakest possible type for actual out-variables, which has already been checked. ■

Note that for any typing environment Γ used in type checking a well-typed program of Section 7.3 and for any label t in this program, $\#\Gamma_P(t) \leq 1$ and there can be at most one reply statement in the program corresponding to any label t. Consequently, the following lemma holds for the language and type system considered in this section:

Lemma 6 *In a well-typed method, a signature and cointerface can be derived for every method invocation in the body, such that the invocation is covered.*

Proof. Let S be the code in a well-typed method such that $\Gamma \vdash_{\mathrm{S}} \mathrm{S} \langle \Delta \rangle$. The proof is by induction over S. For $\mathrm{S} = \varepsilon$, the lemma holds trivially. Now let $\mathrm{S} = \mathrm{S}_0;s;\mathrm{S}_1$ and assume as induction hypothesis that well-typed signatures and cointerfaces have been derived for every invocation in S_0, we now show that this is also the case for $\mathrm{S}_0;s$. Let $\Gamma \vdash_{\mathrm{S}} \mathrm{S}_o;s \langle \Delta' \rangle$. The proof is by case analysis of s; only statements that invoke methods are discussed. (The remaining cases follow directly from the induction hypothesis.)

- For $o.m(\mathrm{E};\mathrm{V})$, rule EXT-SYNC provides the signature $T \to \Gamma_{\mathrm{V}}(\mathrm{V})$, where $\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T$, and cointerface *Co*, such that the invocation is covered.

- For $!o.m(\mathrm{E})$, rule EXT-ASYNC provides the signature $T \to \mathsf{Data}$, where $\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T$, and cointerface *Co*, such that the invocation is covered.

- For $t_i!o.m(\mathrm{E})$, rule EXT-ASYNC-L provides the signature $T \to \mathsf{Data}$, where $\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T$, and cointerface *Co*, for which the rule gives a match. This signature and cointerface *Co* are stored in $\Delta'_{Sig}(t_i)$. If there is a reply statement $t?(\mathrm{V})$ in S_1 before an invocation labeled t_j ($i \neq j$), the return values from the call $t_i!o.m(\mathrm{E})$ will be assigned to V. Hence, the REPLY rule refines the signature to $T \to \Gamma_{\mathrm{V}}(\mathrm{V})$ in $\Delta_{Sig}(t_i)$, such that the invocation is covered by the new signature and *Co*. Otherwise, the return values from the call are not accessible and $\Delta_{Sig}(t_i) = \Delta'_{Sig}(t_i)$. In both cases, $\Delta_{Sig}(t_i)$ provides a signature and cointerface such that the invocation is covered.

- For $m(\mathrm{E};\mathrm{V})$, the INT-SYNC rule provides the signature $T \to \Gamma_{\mathrm{V}}(\mathrm{V})$, where $\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T$, and cointerface *Co*.

- For $!m(\mathrm{E})$, the INT-ASYNC rule provides the signature and cointerface.

- For $t_i!m(\mathrm{E})$, the INT-ASYNC rule provides a signature and cointerface. The signature may be refined by a reply statement as for $t_i!o.m(\mathrm{E})$. The call is covered by the signature and cointerface provided by $(\Gamma + \Delta')_{Sig}(t_i)$ ■

7.3.5 Operational Semantics

The operational semantics of the language is defined in rewriting logic [58]. A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$, where the signature Σ defines the function symbols of the language, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. From a computational viewpoint, a rewrite rule $t \longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t'. Each rewrite rule describes how a part of a configuration can evolve in one transition step.

If rewrite rules may be applied to non-overlapping subconfigurations, the transitions may be performed in parallel. Consequently, concurrency is implicit in rewriting logic (RL). A number of concurrency models have been successfully represented in RL [58, 21], including Petri nets, CCS, Actors, and Unity, as well as the ODP computational model [64]. RL also offers its own model of object orientation [21].

Informally, a state configuration in RL is a multiset of terms of given types. Types are specified in (membership) equational logic (Σ, E), the functional sublanguage of RL which supports algebraic specification in the OBJ [35] style. When modeling computational systems, configurations may include the local system states. Different parts of the system are modeled by terms of the different types defined in the equational logic.

RL extends algebraic specification techniques with transition rules: The dynamic behavior of a system is captured by rewrite rules, supplementing the equations which define the term language. Assuming that all terms can be reduced to normal form, rewrite rules transform terms modulo the equations in E. Conditional rewrite rules are allowed, where the condition is formulated as a conjunction of rewrites and equations which must hold for the main rule to apply:

$$\mathit{subconfiguration} \longrightarrow \mathit{subconfiguration}\ \textbf{if}\ \mathit{condition}.$$

Rules in RL may be formulated at a high level of abstraction, closely resembling a structural operational semantics. In fact, structural operational semantics can be uniformly mapped into RL specifications [59].

System Configurations

Synchronous and asynchronous method calls are given a uniform representation in the operational semantics: Objects communicate by sending messages. Messages have the general form *message* **to** *dest* where *dest* is a single object or class, or a list of classes. The actual signature and cointerface of a method call, as derived during type checking (Lemma 6), are now assumed to be *included* as arguments to the method invocations of the runtime system. (After the signatures and cointerfaces of invocations have been included, the label indices for asynchronous method invocations are erased.) If an object o_1 calls a method m of an object o_2, with actual type *Sig*, cointerface *Co*, and actual parameters E, and the execution of $m(Sig, Co, \mathrm{E})$ results in the return values E', the call is reflected by two messages $invoc(m, Sig, Co, (n\ o_1\ \mathrm{E}))$ **to** o_2 and $comp(n, \mathrm{E}')$ **to** o_1, which represent the invocation and completion of the call, respectively. In the asynchronous setting, invocation messages will include the caller's identity, which ensures that completions can be transmitted to the correct destination. Objects may have several pending calls to another object, so the completion message includes a locally unique label value n, generated by the caller. Object activity is organized around a *message queue* which contains incoming messages and a *process queue* which contains suspended processes; i.e., remaining parts of method activations.

A state configuration is a multiset combining Creol objects, classes, and messages. (In order to increase the parallelism in the model, message queues could be external to object bodies [47, 48].) In RL, objects are commonly represented by terms of the type $\langle O : C \,|a_1 : v_1, \ldots, a_n : v_n\rangle$ where O is the object's identifier, C is its class, the a_i's are the names of the object's attributes, and the v_i's are the corresponding values [21]. We adopt this form of presentation and define

Creol objects and classes as RL objects. Let a process be a pair consisting of a sequence of program statements and a local state, given by a *mapping* which binds program variables to values of their declared types. Omitting RL types, a Creol object is represented by an RL object $\langle Ob \,|\, Cl, Att, Pr, PrQ, EvQ, Lab \rangle$, where *Ob* is the object identifier, *Cl* the class name, *Att* the object state, *Pr* the active process, *PrQ* a multiset of suspended processes, *EvQ* a multiset of unprocessed messages, and *Lab* of type Label is the method call identifier, respectively. Thus, the object identifier *Ob* and the generated local label value provide a globally unique identifier for each method call.

At runtime, classes are represented by RL objects $\langle Cl \,|\, Par, Att, Mtds, Tok \rangle$, where *Cl* is the class name, *Par* and *Att* are lists of parameter and attribute declarations, *Mtds* is a set of methods, and *Tok* is an arbitrary term of sort Label. A method has a name, signature, cointerface, in-parameter, and body. When an object needs a method, it is bound to a definition in the *Mtds* set of the appropriate class. In RL's object model [21], classes are not represented explicitly in the system configuration. This leads to ad hoc mechanisms to handle object creation, which we avoid with explicit class representations. The Creol construct **new** $C(\mathrm{E})$ creates a new object with a unique object identifier, attributes as listed in the class parameter list and in *Att*, and places the code from the *run* method in *Pr*. An *initial (state) configuration* consists of class representations and an initial **new** message.

Executions

An *execution* of a program *P* is a sequence of state configurations such that there is a rewrite step in the operational semantics between every two consecutive configurations. The operational semantics is given in Figures 7.7 and 7.8. There are three main kinds of rewrite rules:

- *Rules that execute code from the active process:* For every program statement there is at least one rule. For example, the assignment rule R1 for the statement $\mathrm{V} := \mathrm{E}$ binds the values of the expression list E to the list V of local and object variables.

- *Rule R5 suspends the active process:* When an active process guard evaluates to false, the process and its local variables are suspended, leaving *Pr* empty.

- *Rule R6 activates a suspended process:* If *Pr* is empty, suspended processes may be activated. The rule selects an arbitrary suspended and enabled process for activation.

In addition, *transport rules* (R11 and R17) move messages into the message queues, representing network flow. The rules are now briefly presented. Auxiliary functions are defined in equational logic and are therefore evaluated in between the state transitions [58]; e.g., the equation $(\varepsilon := \varepsilon) = \varepsilon$ removes empty assignments. Irrelevant attributes are ignored in the style of Full Maude [21]. A detailed discussion may be found in [48].

Whitespace is used as the constructor of multisets, such as *PrQ* and *EvQ*, as well as variable and expression lists, whereas semicolon (with ε as left and right identity) is used as the constructor of lists of statements in order to improve readability. The **skip** statement is understood as ε. As before, $+$ is the constructor for mappings. In the assignment rule R1, a list of expressions is evaluated and bound to a list of program variables. The auxiliary function *eval* evaluates an

$$\begin{array}{l}\langle o : \mathit{Ob} \,|\, \mathit{Att} : \textsc{a}, \mathit{Pr} : \langle (v\ \textsc{v} := e\ \textsc{e}); \textsc{s}, \textsc{l} \rangle\ \rangle \\ \longrightarrow\ \textbf{if } v \mathit{\ in\ } \textsc{l} \textbf{ then } \langle o : \mathit{Ob} \,|\, \mathit{Att} : \textsc{a}, \mathit{Pr} : \langle (\textsc{v} := \textsc{e}); \textsc{s}, (\textsc{l} + [v \mapsto \mathit{eval}(e, (\textsc{a}; \textsc{l}))]) \rangle\ \rangle \\ \qquad \textbf{else } \langle o : \mathit{Ob} \,|\, \mathit{Att} : (\textsc{a} + [v \mapsto \mathit{eval}(e, (\textsc{a}; \textsc{l}))]), \mathit{Pr} : \langle (\textsc{v} := \textsc{e}); \textsc{s}, \textsc{l} \rangle\ \rangle \textbf{ fi}\end{array} \tag{R1}$$

$$\begin{array}{l}\mathit{new}\, C(\textsc{e})\ \langle C : \mathit{Cl} \,|\, \mathit{Par} : (\textsc{v} : T), \mathit{Att} : \textsc{a}, \mathit{Tok} : n \rangle \\ \ \longrightarrow\ \langle C : \mathit{Cl} \,|\, \mathit{Par} : (\textsc{v} : T), \mathit{Att} : \textsc{a}, \mathit{Tok} : \mathit{next}(n) \rangle \\ \langle (C;n) : \mathit{Ob} \,|\, \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle ((\mathit{self} : \mathit{Any} = (C;n); \textsc{v} : T := \textsc{e}; \textsc{a}) \downarrow; \mathit{run}), \varepsilon \rangle, \\ \qquad \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 1 \rangle\end{array} \tag{R2}$$

$$\begin{array}{l}\langle o : \mathit{Ob} \,|\, \mathit{Att} : \textsc{a}, \mathit{Pr} : \langle (v := \mathit{new}\, C(\textsc{e}); \textsc{s}), \textsc{l} \rangle\ \rangle \\ \langle C : \mathit{Cl} \,|\, \mathit{Par} : (\textsc{v} : T), \mathit{Att} : \textsc{a}', \mathit{Tok} : n \rangle \\ \ \longrightarrow\ \langle o : \mathit{Ob} \,|\, \mathit{Att} : \textsc{a}, \mathit{Pr} : \langle (v := (C;n); \textsc{s}), \textsc{l} \rangle\ \rangle\ \langle (C;n) : \mathit{Ob} \,|\, \mathit{Cl} : C, \mathit{Att} : \varepsilon, \\ \qquad \mathit{Pr} : \langle ((\mathit{self} : \mathit{Any} = (C;n); \textsc{v} : T = \mathit{eval}(\textsc{e}, (\textsc{a}; \textsc{l})); \textsc{a}') \downarrow; \mathit{run}), \varepsilon \rangle, \mathit{PrQ} : \varepsilon, \\ \qquad \mathit{EvQ} : \varepsilon, \mathit{Lab} : 1 \rangle\ \langle C : \mathit{Cl} \,|\, \mathit{Par} : (\textsc{v} : T), \mathit{Att} : \textsc{a}', \mathit{Tok} : \mathit{next}(n) \rangle\end{array} \tag{R3}$$

$$\begin{array}{l}\langle o : \mathit{Ob} \,|\, \mathit{Att} : \textsc{a}, \mathit{Pr} : \langle \mathit{await}\, g; \textsc{s}, \textsc{l} \rangle, \mathit{EvQ} : \textsc{q} \rangle \\ \ \longrightarrow\ \langle o : \mathit{Ob} \,|\, \mathit{Att} : \textsc{a}, \mathit{Pr} : \langle \textsc{s}, \textsc{l} \rangle, \mathit{EvQ} : \textsc{q} \rangle \textbf{ if } \mathit{enabled}(g, (\textsc{a}, \textsc{l}), \textsc{q})\end{array} \tag{R4}$$

$$\begin{array}{l}\langle o : \mathit{Ob} \,|\, \mathit{Att} : \textsc{a}, \mathit{Pr} : \langle \textsc{s}, \textsc{l} \rangle, \mathit{PrQ} : \textsc{w}, \mathit{EvQ} : \textsc{q} \rangle \\ \ \longrightarrow\ \langle o : \mathit{Ob} \,|\, \mathit{Att} : \textsc{a}, \mathit{Pr} : \langle \varepsilon, \varepsilon \rangle, \mathit{PrQ} : (\textsc{w}\ \langle \mathit{clear}(\textsc{s}), \textsc{l} \rangle), \mathit{EvQ} : \textsc{q} \rangle \textbf{ if } \mathit{not\ enabled}(\textsc{s}, (\textsc{a}; \textsc{l}), \textsc{q})\end{array} \tag{R5}$$

$$\begin{array}{l}\langle o : \mathit{Ob} \,|\, \mathit{Att} : \textsc{a}, \mathit{Pr} : \langle \varepsilon, \textsc{l}' \rangle, \mathit{PrQ} : \langle \textsc{s}, \textsc{l} \rangle\ \textsc{w}, \mathit{EvQ} : \textsc{q} \rangle \\ \ \longrightarrow\ \langle o : \mathit{Ob} \,|\, \mathit{Att} : \textsc{a}, \mathit{Pr} : \langle \textsc{s}, \textsc{l} \rangle, \mathit{PrQ} : \textsc{w}, \mathit{EvQ} : \textsc{q} \rangle \textbf{ if } \mathit{enabled}(\textsc{s}, (\textsc{a}, \textsc{l}), \textsc{q})\end{array} \tag{R6}$$

$$\begin{array}{l}\langle o : \mathit{Ob} \,|\, \mathit{Pr} : \langle (r(\mathit{Sig}, \mathit{Co}, \textsc{e}; \textsc{v}); \textsc{s}), \textsc{l} \rangle, \mathit{Lab} : n \rangle \\ \ \longrightarrow\ \langle o : \mathit{Ob} \,|\, \mathit{Pr} : \langle (!r(\mathit{Sig}, \mathit{Co}, \textsc{e}); n?(\textsc{v}); \textsc{s}), \textsc{l} \rangle, \mathit{Lab} : n \rangle\end{array} \tag{R7}$$

$$\begin{array}{l}\langle o : \mathit{Ob} \,|\, \mathit{Att} : \textsc{a}, \mathit{Pr} : \langle (t!r(\mathit{Sig}, \mathit{Co}, \textsc{e}); \textsc{s}), \textsc{l} \rangle, \mathit{Lab} : n \rangle \\ \ \longrightarrow\ \langle o : \mathit{Ob} \,|\, \mathit{Att} : \textsc{a}, \mathit{Pr} : \langle (t := n; !r(\mathit{Sig}, \mathit{Co}, \textsc{e}); \textsc{s}), \textsc{l} \rangle, \mathit{Lab} : n \rangle\end{array} \tag{R8}$$

$$\begin{array}{l}\langle o : \mathit{Ob} \,|\, \mathit{Att} : \textsc{a}, \mathit{Pr} : \langle (!x.m(\mathit{Sig}, \mathit{Co}, \textsc{e}); \textsc{s}), \textsc{l} \rangle, \mathit{Lab} : n \rangle \\ \ \longrightarrow\ \langle o : \mathit{Ob} \,|\, \mathit{Att} : \textsc{a}, \mathit{Pr} : \langle \textsc{s}, \textsc{l} \rangle, \mathit{Lab} : \mathit{next}(n) \rangle \\ \mathit{invoc}(m, \mathit{Sig}, \mathit{Co}, (o\ n\ \mathit{eval}(\textsc{e}, (\textsc{a}; \textsc{l})))) \textbf{ to } \mathit{eval}(x, (\textsc{a}; \textsc{l}))\end{array} \tag{R9}$$

Figure 7.7: An operational semantics in rewriting logic (1).

expression in a given *state*; the equations for the functional language $\mathcal{F}$ extend state lookup, given by

$$\begin{array}{l}\mathit{eval}(\varepsilon, \textsc{l}) = \varepsilon \\ \mathit{eval}(v, \textsc{l} + [v' \mapsto d]) = \textbf{if } v = v' \textbf{ then } d \textbf{ else } \mathit{eval}(v, \textsc{l}) \textbf{ fi} \\ \mathit{eval}(v\ \textsc{v}, \textsc{l}) = (\mathit{eval}(v, \textsc{l})\ \mathit{eval}(\textsc{v}, \textsc{l}))\end{array}$$

In the object creation rules R2 and R3, an object state is constructed from the class parameters and attribute list, an object identifier for the new object is constructed, and the *run* method is synchronously invoked. Let *self* of type *Any* be the self reference in a runtime object. The object identifier $(C;n)$ is, by the typing rule NEW, typed by an interface I such that $I \preceq \mathit{Any}$. The parameter is represented as a typed list of variables to accommodate the assignment rule. In order to apply R1 to the initialization of the object state, a *type erasure* function on attribute lists is introduced, defined recursively by $(\varepsilon) \downarrow = \varepsilon$ and $(\textsc{v} : T = \textsc{e}; \textsc{s}) \downarrow = \textsc{v} := \textsc{e}; (\textsc{s}) \downarrow$. New object identifiers are created by concatenating tokens n from the unbounded set *Tok* to the class

(R10) $$\begin{array}{l}\langle o : Ob \mid Att : \textsc{a}, Pr : \langle (!m(Sig, Co, \textsc{e}); \textsc{s}), \textsc{l}\rangle, Lab : n\rangle \\ \longrightarrow \langle o : Ob \mid Att : \textsc{a}, Pr : \langle \textsc{s}, \textsc{l}\rangle, Lab : next(n)\rangle \\ invoc(m, Sig, Co, (o\ n\ eval(\textsc{e}, (\textsc{a}; \textsc{l})))) \ \mathbf{to}\ o\end{array}$$

(R11) $$(msg\ \mathbf{to}\ o)\ \langle o : Ob \mid EvQ : \textsc{q}\rangle \longrightarrow \langle o : Ob \mid EvQ : \textsc{q}\ msg\rangle$$

(R12) $$\begin{array}{l}\langle o : Ob \mid Cl : C, EvQ : \textsc{q}\ invoc(m, Sig, Co, \textsc{e})\rangle \\ \longrightarrow \langle o : Ob \mid Cl : C, EvQ : \textsc{q}\rangle\ bind(m, Sig, Co, \textsc{e}, o)\ \mathbf{to}\ C\end{array}$$

(R13) $$\begin{array}{l}\langle o : Ob \mid Pr : \langle (t\,?(\textsc{v}); \textsc{s}), \textsc{l}\rangle, EvQ : \textsc{q}\ comp(n, \textsc{e})\rangle \\ \longrightarrow \langle o : Ob \mid Pr : \langle (\textsc{v} := \textsc{e}; \textsc{s}), \textsc{l}\rangle, EvQ : \textsc{q}\rangle\ \mathbf{if}\ n = eval(t, \textsc{l})\end{array}$$

(R14) $$\begin{array}{l}\langle o : Ob \mid Pr : \langle (t\,?(\textsc{v}); \textsc{s}), \textsc{l}\rangle, PrQ : (\textsc{s}', \textsc{l}')\ \textsc{w}\rangle \\ \longrightarrow \langle o : Ob \mid Pr : \langle \textsc{s}'; cont(eval(t, \textsc{l})), \textsc{l}'\rangle, PrQ : \langle await\ t\,?(\textsc{v}); \textsc{s}, \textsc{l}\rangle\ \textsc{w}\rangle \\ \mathbf{if}\ eval(caller, \textsc{l}') = o \wedge eval(label, \textsc{l}') = eval(t, \textsc{l})\end{array}$$

(R15) $$\begin{array}{l}\langle o : Ob \mid Pr : \langle cont(n), \textsc{l}\rangle, PrQ : \langle (await\ (t'?); \textsc{s}), \textsc{l}'\rangle\ \textsc{w}\rangle \\ \longrightarrow \langle o : Ob \mid Pr : \langle \textsc{s}, \textsc{l}'\rangle, PrQ : \textsc{w}\rangle\ \mathbf{if}\ eval(t', \textsc{l}') = n\end{array}$$

(R16) $$\begin{array}{l}(bind(m, Sig, Co, \textsc{e}, o)\ \mathbf{to}\ C)\ \langle C : Cl \mid Mtds : \textsc{m}\rangle \\ \longrightarrow \mathbf{if}\ match(m, Sig, Co, \textsc{m}) \\ \qquad\qquad \mathbf{then}\ (bound(get(m, \textsc{m}, \textsc{e}))\ \mathbf{to}\ o) \\ \qquad\qquad \mathbf{else}\ (bind(m, Sig, Co, \textsc{e}, o)\ \mathbf{to}\ \varepsilon)\ \mathbf{fi}\ \langle C : Cl \mid Mtds : \textsc{m}\rangle\end{array}$$

(R17) $$(bound(w)\ \mathbf{to}\ o)\ \langle o : Ob \mid PrQ : \textsc{w}\rangle \longrightarrow \langle o : Ob \mid PrQ : \textsc{w}\ w\rangle$$

(R18) $$\begin{array}{l}\langle o : Ob \mid Att : \textsc{a}, Pr : \langle (return(\textsc{v}); \textsc{p}), \textsc{l}\rangle\rangle \\ \longrightarrow \langle o : Ob \mid Att : \textsc{a}, Pr : \langle \textsc{p}, \textsc{l}\rangle\rangle \\ comp(eval(label, \textsc{l}), eval(\textsc{v}, (\textsc{l})))\ \mathbf{to}\ eval(caller, \textsc{l})\end{array}$$

Figure 7.8: An operational semantics in rewriting logic (2).

name. The identifier is returned to the object which initiated the object creation. Before the new object can be activated, its state must be initialized. This is done by assigning actual values to class parameters and then evaluating the attribute list. The rules R4, R5, and R6 for guards depend on an *enabledness* function. Let $\textsc{d}$ denote a state and let the infix function *in* check whether a completion message corresponding to a given label value is in a message queue $\textsc{q}$. The *enabledness* function is defined by induction over the construction of guards:

$$\begin{array}{l}enabled(t?, \textsc{d}, \textsc{q}) = eval(t, \textsc{d})\ in\ \textsc{q} \\ enabled(b, \textsc{d}, \textsc{q}) = eval(b, \textsc{d}) \\ enabled(wait, \textsc{d}, \textsc{q}) = false \\ enabled(g \vee g', \textsc{d}, \textsc{q}) = enabled(g, \textsc{d}, \textsc{q}) \vee enabled(g', \textsc{d}, \textsc{q}) \\ enabled(g \wedge g', \textsc{d}, \textsc{q}) = enabled(g, \textsc{d}, \textsc{q}) \wedge enabled(g', \textsc{d}, \textsc{q})\end{array}$$

When a non-enabled guard is encountered in R5, the active process is suspended on the process queue. In this rule, the auxiliary function *clear* removes occurrences of *wait* from any leading guards. The enabledness predicate is extended to *statements* as follows (where $\textsc{s}$ may match an empty statement list):

$enabled(s;s';\textsc{s},\textsc{d},\textsc{q}) = enabled(s,\textsc{d},\textsc{q})$
$enabled(\mathit{await}\ g,\textsc{d},\textsc{q}) = enabled(g,\textsc{d},\textsc{q})$
$enabled(s,\textsc{d},\textsc{q}) = true \qquad [\textbf{otherwise}]$

The **otherwise** attribute of the last equation states that this equation is taken when no other equation matches.

In R7, a synchronous call $r(Sig,Co,\textsc{e};\textsc{v})$, where $\textsc{v}$ is a list of variables and r is either m or $x.m$, is translated into an *asynchronous call*, $!r(Sig,Co,\textsc{e})$, followed by a blocking *reply statement*, $n?(\textsc{v})$, where n is the label value uniquely identifying the call. In R8 a labeled asynchronous call is translated into a label assignment and an unlabeled asynchronous call, which results in an invocation message in R9. Internal calls are treated as external calls to *self* in R10. Guarded calls are expanded to asynchronous calls and guarded replies, as defined in Section 7.3.1.

When an object calls a method, a message is emitted into the configuration (R9 or R10) and delivered to the callee (R11) where *msg* ranges over invocations and completions. Message overtaking is captured by the nondeterminism inherent in RL: invocation and completion messages sent by an object to another object in one order may arrive in any order. The call is bound by sending a *bind* message to the class of the object (R12). Note that for external calls, the class of the callee is first identified in this rule; consequently external method calls are virtually bound.

The *bind* message is handled by R16, which identifies the method m in the method set $\textsc{m}$ of the class. The auxiliary predicate $match(m,Sig,Co,\textsc{m})$ evaluates to true if m is declared in $\textsc{m}$ with a signature Sig' and cointerface Co' such that $Sig' \preceq Sig$, $Co \preceq Co'$. Note that a *method-not-understood* error is represented by a *bind* message sent to an empty list of classes. Furthermore, the auxiliary function *get* returns a process with the method's code and local state (instantiating the method's in-parameters with the call's actual parameters $\textsc{e}$ and binding local variable declarations to initial expressions) from the method set $\textsc{m}$ of the class, and ensures that a completion message is emitted upon method termination, by appending a special construct $return(\textsc{v})$ to the method code. The values of the actual in-parameters, the caller, and the label value n are stored locally; the caller and label value are stored in the local variables *caller* and *label*. The special construct $return(\textsc{v})$ is used in R18 to return a uniquely labeled completion message to the caller. The process w resulting from the binding is loaded into the internal process queue in R17.

The reply statement fetches the return values corresponding to $\textsc{v}$ from the completion message in the object's queue (R13). In the model, *EvQ* is a multiset; thus the rule matches any occurrence of $comp(n,\textsc{e})$ in the queue. The use of rewrite rules rather than equations mimics distributed and concurrent processing of method lookup. Note the special construct $cont(n)$ in R14 and R15, which is used to control local calls in order to avoid deadlock in the case of self reentrance [48] and impose a LIFO ordering of local calls.

Example. We consider an execution sequence inspired by the example of Section 7.3.4. Let C' be a class similar to class C, but without an active *run* method. The runtime representation of class C' is given as

$$\begin{aligned}\langle C' : Cl \mid Par : \varepsilon, Att : \varepsilon, Mtds = \{ &\langle run, \varepsilon \to \varepsilon, \varsigma, \varepsilon, \langle return(\varepsilon), \varepsilon\rangle\rangle \\ &\langle m, \varepsilon \to \mathsf{Bool}, B, \varepsilon, \langle (x := \mathsf{true}; return(x)), x \mapsto d_{\mathrm{Bool}}\rangle\rangle \\ &\langle m, \varepsilon \to \mathsf{Nat}, A, \varepsilon, \langle\ (x := 0; return(x)), x \mapsto d_{\mathrm{Nat}}\rangle\rangle\}, Tok : 1\rangle.\end{aligned}$$

Figure 7.9 presents an execution sequence in which an object of class C creates an instance of C' and makes an asynchronous call to the new object. The call $t!o.m()$ of the Creol code is expanded to $t!o.m(\varepsilon \to \mathsf{Nat}, AB, \varepsilon)$ after the type analysis. This call causes an invocation to the C' object, of the method m with Nat output, which again causes a completion with the value 0. The object of class C assigns 0 to its local variable x and the execution terminates.

7.3.6 Type Soundness

The soundness of the type system is established in this section. First, we define well-typed runtime objects, configurations, and executions. Then, we show that when applying rewrite rules to the final state of a well-typed execution, the execution remains well-typed. In particular, method-not-understood errors do not occur. Say that a runtime object is of a program P if it is an instance of a class defined in P.

Definition 10 Let $\langle o : Ob \mid Att : \mathrm{A}, Pr : \langle \mathrm{S}, \mathrm{L}\rangle, PrQ : \langle \mathrm{S}_1, \mathrm{L}_1\rangle \ldots \langle \mathrm{S}_n, \mathrm{L}_n\rangle\rangle$ represent a runtime object of a program P. If A, L, $\mathrm{L}_1, \ldots, \mathrm{L}_n$ are well-typed states in the typing environment of P, then the runtime object is *well-typed*.

An object has been *initialized* when the object state has been constructed and the object is ready to call *run*. Recall that method-not-understood errors are captured technically by *bind* messages with no destination address. Say that a configuration is of a program P if all objects in the configuration are of P.

Definition 11 In a *well-typed configuration* of a program P, there are no method-not-understood errors and every object in the configuration is a well-typed runtime object of P with a unique identity. A *well-typed initial configuration* of a program P is an initial configuration of a well-typed program P. A *well-typed execution* of a program P is an execution that starts in an initial configuration and in which every configuration is well-typed.

Note that all objects in a configuration of a well-typed execution follow the naming convention of the object creation rules R2 and R3. Furthermore, all messages in a configuration of a well-typed execution are generated directly or indirectly by a method call in a well-typed object; i.e., all invocation messages are generated from the asynchronous call statement (R9 and R10), all *bind* messages are generated from these invocation messages (R12), all *bound* messages result from *bind* messages (R16), and all completion messages result from method termination (R18).

Lemma 7 *Given an arbitrary Creol program P and a well-typed execution ρ of P. The execution of a statement $x :=$ **new** $C(\mathrm{E})$ in the final configuration of ρ results in well-typed configurations of P while the new object is initialized.*

$\langle C' : \mathit{Cl} \mid \mathit{Par} : \varepsilon, \mathit{Att} : \varepsilon, \mathit{Mtds} = \textsc{m}, \mathit{Tok} : 1\rangle$

$\langle (C;1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle (o := \mathbf{new}\ C'; t!o.m(\varepsilon \to \mathrm{Nat}, AB, \varepsilon); \mathbf{await}\ t?; t?(x)),$
$\quad (x \mapsto d_{\mathrm{Nat}}, o \mapsto \mathit{null}, t \mapsto d_{\mathrm{Label}})\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 2\rangle$

$\longrightarrow$ R2

$\langle C' : \mathit{Cl} \mid \mathit{Par} : \varepsilon, \mathit{Att} : \varepsilon, \mathit{Mtds} = \textsc{m}, \mathit{Tok} : 2\rangle$

$\langle (C;1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle (o := (C';1); t!o.m(\varepsilon \to \mathrm{Nat}, AB, \varepsilon); \mathbf{await}\ t?; t?(x)),$
$\quad (x \mapsto d_{\mathrm{Nat}}, o \mapsto \mathit{null}, t \mapsto d_{\mathrm{Label}})\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 2\rangle$

$\langle (C';1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle \mathit{run}(\varepsilon \to \varepsilon, \varsigma, \varepsilon); \varepsilon\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 1\rangle$

$\longrightarrow$ R1, $\longrightarrow$ R8, we omit the default invocation of the empty *run* method (e.g., **skip**) in $(C';1)$.

$\langle C' : \mathit{Cl} \mid \mathit{Par} : \varepsilon, \mathit{Att} : \varepsilon, \mathit{Mtds} = \textsc{m}, \mathit{Tok} : 2\rangle$

$\langle (C;1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle (t := 2; !o.m(\varepsilon \to \mathrm{Nat}, AB, \varepsilon); \mathbf{await}\ t?; t?(x)),$
$\quad (x \mapsto d_{\mathrm{Nat}}, o \mapsto (C';1), t \mapsto d_{\mathrm{Label}})\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 2\rangle$

$\langle (C';1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle \varepsilon, \varepsilon\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 2\rangle$

$\longrightarrow$ R1, $\longrightarrow$ R9

$\langle C' : \mathit{Cl} \mid \mathit{Par} : \varepsilon, \mathit{Att} : \varepsilon, \mathit{Mtds} = \textsc{m}, \mathit{Tok} : 2\rangle$

$\langle (C;1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle \mathbf{await}\ t?; t?(x), (x \mapsto d_{\mathrm{Nat}}, o \mapsto (C';1), t \mapsto 2)\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 3\rangle$

$\langle (C';1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle 1?(\varepsilon), \varepsilon\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 2\rangle$

$\mathit{invoc}(m, \varepsilon \to \mathrm{Nat}, AB, (C;1)\ 2)$ **to** $(C';1)$

$\longrightarrow$ R11, $\longrightarrow$ R12

$\langle C' : \mathit{Cl} \mid \mathit{Par} : \varepsilon, \mathit{Att} : \varepsilon, \mathit{Mtds} = \textsc{m}, \mathit{Tok} : 2\rangle$

$\langle (C;1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle \mathbf{await}\ t?; t?(x), (x \mapsto d_{\mathrm{Nat}}, o \mapsto (C';1), t \mapsto 2)\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 3\rangle$

$\langle (C';1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle \varepsilon, \varepsilon\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} :, \mathit{Lab} : 2\rangle$

$\mathit{bind}(m, \varepsilon \to \mathrm{Nat}, AB, (C;1)\ 2)$ **to** C'

$\longrightarrow$ R16

$\langle C' : \mathit{Cl} \mid \mathit{Par} : \varepsilon, \mathit{Att} : \varepsilon, \mathit{Mtds} = \textsc{m}, \mathit{Tok} : 2\rangle$

$\langle (C;1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle \mathbf{await}\ t?; t?(x), (x \mapsto d_{\mathrm{Nat}}, o \mapsto (C';1), t \mapsto 2)\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 3\rangle$

$\langle (C';1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle \varepsilon, \varepsilon\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 2\rangle$

$\mathit{bound}(\langle (x := 0; \mathit{return}(x)), (\mathit{caller} \mapsto (C;1), \mathit{label} \mapsto 2, x \mapsto d_{\mathrm{Nat}})\rangle)$ **to** $(C';1)$

$\longrightarrow$ R17, $\longrightarrow$ R6

$\langle C' : \mathit{Cl} \mid \mathit{Par} : \varepsilon, \mathit{Att} : \varepsilon, \mathit{Mtds} = \textsc{m}, \mathit{Tok} : 2\rangle$

$\langle (C;1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle \mathbf{await}\ t?; t?(x), (x \mapsto d_{\mathrm{Nat}}, o \mapsto (C';1), t \mapsto 2)\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 3\rangle$

$\langle (C';1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle (x := 0; \mathit{return}(x)), (\mathit{caller} \mapsto (C;1), \mathit{label} \mapsto 2, x \mapsto d_{\mathrm{Nat}})\rangle,$
$\quad \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 2\rangle$

$\longrightarrow$ R1, $\longrightarrow$ R18

$\langle C' : \mathit{Cl} \mid \mathit{Par} : \varepsilon, \mathit{Att} : \varepsilon, \mathit{Mtds} = \textsc{m}, \mathit{Tok} : 2\rangle$

$\langle (C;1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle \mathbf{await}\ t?; t?(x), (x \mapsto d_{\mathrm{Nat}}, o \mapsto (C';1), t \mapsto 2)\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 3\rangle$

$\langle (C';1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle \varepsilon, (\mathit{caller} \mapsto (C;1), \mathit{label} \mapsto 2, x \mapsto 0)\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 2\rangle$

$\mathit{comp}(2,0)$ **to** $(C;1)$

$\longrightarrow$ R10, $\longrightarrow$ R4

$\langle C' : \mathit{Cl} \mid \mathit{Par} : \varepsilon, \mathit{Att} : \varepsilon, \mathit{Mtds} = \textsc{m}, \mathit{Tok} : 2\rangle$

$\langle (C;1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle t?(x), (x \mapsto d_{\mathrm{Nat}}, o \mapsto (C';1), t \mapsto 2)\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \mathit{comp}(2,0), \mathit{Lab} : 3\rangle$

$\langle (C';1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle \varepsilon, (\mathit{caller} \mapsto (C;1), \mathit{label} \mapsto 2, x \mapsto 0)\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 2\rangle$

$\longrightarrow$ R13, since $t \mapsto 2$

$\langle C' : \mathit{Cl} \mid \mathit{Par} : \varepsilon, \mathit{Att} : \varepsilon, \mathit{Mtds} = \textsc{m}, \mathit{Tok} : 2\rangle$

$\langle (C;1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle x := 0, (x \mapsto d_{\mathrm{Nat}}, o \mapsto (C';1), t \mapsto 2)\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 3\rangle$

$\langle (C';1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle \varepsilon, (\mathit{caller} \mapsto (C;1), \mathit{label} \mapsto 2, x \mapsto 0)\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 2\rangle$

$\longrightarrow$ R1

$\langle C' : \mathit{Cl} \mid \mathit{Par} : \varepsilon, \mathit{Att} : \varepsilon, \mathit{Mtds} = \textsc{m}, \mathit{Tok} : 2\rangle$

$\langle (C;1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle \varepsilon, (x \mapsto 0, o \mapsto (C';1), t \mapsto 2)\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 3\rangle$

$\langle (C';1) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle \varepsilon, (\mathit{caller} \mapsto (C;1), \mathit{label} \mapsto 2, x \mapsto 0)\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 2\rangle$

Figure 7.9: An example of an execution sequence. The representation of C as well as some intermediary states are omitted, $\longrightarrow$ RX denotes the application of rule RX. For convenience, we denote the method set of C' by $\textsc{m}$, $\mathit{next}(n)$ by $n+1$, and ignore equational reduction.

Proof. Let o be a runtime object executing $x := \textbf{new}\ C(\mathrm{E})$ (by applying rule R3) in the final configuration ρ_i of ρ (and let o' denote o after executing the statement). Since ρ is well-typed, o is well-typed. Since P is well-typed, Lemma 2 asserts that $\Gamma_{\mathrm{V}}(x)$ has a type J and the typing rule for object creation ensures that there is an interface $I \in \Gamma_{C}(C).\mathit{Impl}; \Gamma_{C}(C).\mathit{Contract}$ such that $I \preceq J$. Consequently, the new object reference $(C;n)$ may be typed by J and o' is well-typed.

It remains to show that object creation results in a new well-typed initialized runtime object of class C with a unique identifier. Let E' denote E evaluated in o. Given a runtime class representation $\langle C : \mathit{Cl} \,|\, \mathit{Par} : (\mathrm{V} : T), \mathit{Att} : \mathrm{A}, \mathit{Tok} : n \rangle$ in ρ_i, the configuration ρ_{i+1} includes a runtime object

$$\langle (C;n) : \mathit{Ob} \,|\, \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle ((\mathrm{V} : T = \mathrm{E}'; \mathrm{A}) \downarrow; \mathit{run}), \varepsilon \rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 1 \rangle,$$

which is well-typed. All object identifiers in ρ_i have been constructed by applications of R2 and R3, as pairs consisting of a class identifier and an element of type Label. In particular, no identifiers for instances of other classes than C contain the class identifier C, all instances of class C may be ordered by the relation $<$ on Label, and for any instance $(C;n')$ of C in ρ_i we have $n' < n$. Consequently, $(C;n)$ is an unused identifier in ρ_i. As the application of R3 locks the class in the rewrite step from ρ_i to ρ_{i+1}, $(C;n)$ is a unique identifier in ρ_{i+1}. Assuming that other concurrent activity in the configuration preserves well-typedness, ρ_{i+1} is well-typed. We now show that object initialization preserves well-typedness; i.e., for some well-typed state σ the object reduces to

$$\langle (C;n) : \mathit{Ob} \,|\, \mathit{Cl} : C, \mathit{Att} : \sigma, \mathit{Pr} : \langle \mathit{run}, \varepsilon \rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 1 \rangle$$

in a new well-typed configuration ρ_{i+j}. Note that the rewrite steps involved in object initialization, i.e., the repeated application of R1, are internal to $(C;n)$. Consequently, concurrent activity does not influence the initialization and for simplicity we assume that such activity preserves configuration well-typedness. For class C, rule CLASS asserts that $\Gamma \vdash_{\mathrm{V}} \mathit{Param}; \mathit{Var} \langle\, \Delta \,\rangle$. It follows by induction over the length of Var that the assignment of initial expressions to the program variables in the list $(\mathit{self} : \mathit{Any} = (C;n); \mathrm{V} : T = \mathrm{E}'; \mathrm{A}) \downarrow$ is type-correct and results in a well-typed state σ which is defined for self and the variables declared in $\mathit{Param}; \mathit{Var}$.

For $\mathit{Var} = \varepsilon$, we have $\Gamma_{\mathcal{F}}(\mathit{self}) = \mathit{Any}$ and $\Gamma_{\mathcal{F}}((C;n)) = I$ so $I \preceq \mathit{Any}$ and rule NEW asserts that $\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T'$ with $T' \preceq T$. Since E is well-typed, we have $\Gamma \vdash_{\mathrm{F}} \mathrm{E}' : T''$ such that $T'' \preceq T'$. Since $T'' \preceq T' \preceq T$, the assignment $(\mathrm{V} : T = \mathrm{E}') \downarrow$ is type-correct. By induction over the length k of Par, we show that a well-typed state σ is built by the multiple assignment $(\mathrm{V} : T = \mathrm{E}') \downarrow$. For $k = 0$, there are no parameters and $\sigma = \varepsilon$ is well-typed (as is ρ_{i+1}). For the induction step, assume that ρ_{i+k-1} is a well-typed configuration in which σ_{k-1} $(1 < k \leq n)$ is the well-typed state of $(C;n)$ constructed by assigning values to the variables $v_1, \ldots, v_{k-1}$. Rule PAR asserts that the variable name v_k of type T_k is new, so an assignment to v_k does not override a previous variable in σ_{k-1}. By rule NEW, the assignment $v_k := e'_k$ is type-correct, so the application of R1 results in a well-typed state $\sigma_{k-1} \cup \{v_k \mapsto e_k\}$. It follows that the object state $\sigma = \sigma_n$ is well-typed and defined for the variables V, and that ρ_{i+k} is a well-typed configuration.

For the induction step, assume $\Gamma \vdash_{\mathrm{V}} \mathit{Param}; \mathit{Var} \langle\, \Delta \,\rangle$ such that the assignment list $(\mathrm{A}) \downarrow$ is type-correct, resulting in a well-typed state σ, defined for self and the variables declared in *Param*; *Var*. If $\Gamma+\Delta \vdash_{\mathrm{V}} \ v\colon T = e \ \langle \Delta' \rangle$ for some type T, then $(\mathrm{A}; v\colon T = e)\downarrow$ is also a type-correct assignment list, resulting in a well-typed state σ' defined for *self*, the variables declared in *Param*; *Var*, and for v. Since $\Gamma+\Delta \vdash_{\mathrm{V}} \ v\colon T = e \ \langle \Delta' \rangle$, the variable name v is new, $\Gamma+\Delta \vdash_{\mathrm{F}} \ e\colon T'$ such that $T' \preceq T$, and e reduces to a value e' such that $\Gamma+\Delta \vdash_{\mathrm{F}} \ e'\colon T''$ and $T'' \preceq T' \preceq T$. It follows that applying R1 results in a well-typed state $\sigma' = \sigma \cup \{v \mapsto e'\}$. There are no other processes with local state in $(C;n)$, so the initialized runtime representation of $(C;n)$ is well-typed. Assuming that other concurrent rewrites in the transition from ρ_i to ρ_{i+j} preserve well-typedness, $\rho_{i+1},\ldots,\rho_{i+j}$ are well-typed configurations. ■

Using the same argument, we can show that a **new** message in a well-typed initial configuration of a program P creates a well-typed configuration of P (by R2). It follows by Lemma 7 that any program variable typed by an interface I will, if not null, point to an object of a class which implements I.

Lemma 8 *Let P be an arbitrary program. If $\Gamma_{\mathcal{F}} \vdash P$, then every method invocation $!x.m(T_{in} \to T_{out}, Co, \mathrm{E})$ or $!m(T_{in} \to T_{out}, Co, \mathrm{E})$ in a well-typed configuration of P can be type-correctly bound at runtime to a method such that the return values from the method are of type T'_{out} and $T'_{out} \preceq T_{out}$, provided that x is not a null pointer.*

Proof. By Lemma 6, the signature $Sig = T_{in} \to T_{out}$ and cointerface Co of every invocation is derived by the type system. The proof considers the evaluation rule R9 for $!x.m(Sig, Co, \mathrm{E})$ and R10 for $!m(Sig, Co, \mathrm{E})$. Let E, x evaluate to E', x' such that $\Gamma(\mathrm{E}') \preceq \Gamma(\mathrm{E})$ and $\Gamma(x') \preceq \Gamma(x)$. For an external method call $!x.m(Sig, Co, \mathrm{E})$, rule R9 creates an invocation message in the following rewrite step

$$\begin{array}{l} \langle o : Ob \mid Pr\colon \langle (!x.m(Sig, Co, \mathrm{E}); \mathrm{S}) \rangle, Lab\colon n \rangle \\ \longrightarrow \ \langle o : Ob \mid Pr\colon \langle \mathrm{S} \rangle, Lab\colon next(n) \rangle \ invoc(m, Sig, Co, (o \ n \ \mathrm{E}')) \ \textbf{to} \ x'. \end{array}$$

Let C be the runtime class of x. After $invoc(m, Sig, Co, (o \ n \ \mathrm{E}'))$ **to** x' eventually arrives at x' by application of R11, the application of R12 generates a message $bind(m, Sig, Co, (o \ n \ \mathrm{E}'), x')$ **to** C. Lemma 7 asserts that if $\Gamma_{\mathrm{V}}(x') = I$ for some interface I, then C implements I. It follows from the type analysis that there must be a signature Sig_I and cointerface Co_I for m in I and a signature $Sig_C = T'_{in} \to T'_{out}$ and cointerface Co_C for m in C such that

$$Sig_C \preceq Sig_I \preceq Sig \text{ and } Co \preceq Co_I \preceq Co_C,$$

so $T'_{out} \preceq T_{out}$. As the subtype relation is transitive the runtime *match* function succeeds and the application of R16 results in the rewrite step

$$\begin{array}{l} (bind(m, Sig, Co, (o \ n \ \mathrm{E}'), x') \ \textbf{to} \ C) \ \langle C : Cl \mid Mtds : \mathrm{M} \rangle \\ \longrightarrow \ (bound(get(m, \mathrm{M}, (o \ n \ \mathrm{E}'))) \ \textbf{to} \ x') \ \langle C : Cl \mid Mtds : \mathrm{M} \rangle. \end{array}$$

For an internal runtime invocation $!m(Sig, \varsigma, \mathrm{E})$, we get $Sig_C \preceq Sig$ and $\varsigma \preceq \varsigma$ directly from rule INT-ASYNC and the runtime *match* function succeeds. ■

All invocations in Creol are expanded into asynchronous invocations at runtime and all *bind* messages in a well-typed execution are generated from these invocations. Consequently, Lemma 8 implies that for every *bind* message $bind(m, T_1 \rightarrow T_2, Co, \text{E}, o)$ **to** C in a well-typed execution, $\Gamma_{\text{V}}(\text{E}) \preceq T_1$ and the matching of $T_1 \rightarrow T_2$ and Co with the formal declaration of m in C succeeds. Since $\Gamma_{\text{V}}(\text{E}) \preceq T_1$, the instantiation of local variables for the new process results in a well-typed state. Thus, loading a new process into the process queue of a well-typed object by R17 results in a well-typed runtime object.

Lemma 9 *Given an arbitrary Creol program P such that $\Gamma_{\mathcal{F}} \vdash P$ and a well-typed execution ρ of P. The execution of a statement $t?(\text{V})$ in the final configuration of ρ results in a well-typed configuration of P.*

Proof. We consider an object executing the (enabled) statement $t?(\text{V})$ by applying R13 on the final configuration of a well-typed execution ρ of P. Rule REPLY asserts that there is a pending method invocation with label t, say $t_i!o.m(\text{E})$ for some index i, object o, method name m, and data E (where $\Gamma \vdash_{\text{F}} \text{E} : T$). We need to show that the runtime method lookup selects a method body such that the return values are of a subtype of the type $\Gamma_{\text{V}}(\text{V})$, in order to ensure that the object remains well-typed after applying R13.

By Lemma 6, the signature derived by the type analysis for every call is such that the call is covered. This signature $Sig = T \rightarrow \Gamma_{\text{V}}(\text{V})$ and cointerface Co are used by the runtime system, yielding $t_i!o.m(Sig, Co, \text{E})$. By Lemma 8 the signature $Sig' = T_{in} \rightarrow T_{out}$ of the method selected at runtime is such that $Sig' \preceq Sig$, so $T_{out} \preceq \Gamma_{\text{V}}(\text{V})$. Since ρ is well-typed, the return values E' assigned to the out-parameters of the call by 18 are such that $\Gamma(\text{E}') \preceq T_{out}$. Consequently, $\Gamma(\text{E}') \preceq \Gamma_{\text{V}}(\text{V})$ and the result of applying R13 is a well-typed configuration of P. ■

Theorem 10 (Type soundness) *All executions of programs starting in a well-typed initial configuration, are well-typed.*

Proof. We consider a well-typed execution ρ of a program P. The proof is by induction over the length of $\rho = \rho_0, \rho_1, \ldots$. By assumption, ρ_0 is a well-typed initial configuration of P; i.e., ρ_0 consists of class representations and a **new** message. Only R2 is applicable to ρ_0 and, by Lemma 7, ρ_1 is well-typed.

For the induction step we show that, for any well-typed configuration ρ_i, the successor configuration ρ_{i+1} is well-typed by case analysis over the rewrite rules of the operational semantics. We first consider the reduction of an object $\langle o : Ob \,|\, Att : \text{A}, Pr : \langle s; \text{S}, \text{L} \rangle \rangle$ to $\langle o : Ob \,|\, Att : \text{A}', Pr : \langle s'; \text{S}, \text{L}' \rangle \rangle$ and then the remaining rewrite rules.

- For $s = (v \ \text{V} := e \ \text{E})$ and the application of R1, the execution of $(v \ \text{V} := e \ \text{E})$ reduces the statement to $\text{V} := \text{E}$. Since the program is well-typed, we can assume that $\Gamma_{\text{V}}(v) = T_v$, $\Gamma \vdash_{\text{F}} e : T$, $T \preceq T_v$, and the functional expression e reduces to e' with type $\Gamma \vdash_{\text{F}} e' : T'$ such that $T' \preceq T$. By transitivity $T' \preceq T_v$ and ρ_{i+1} is well-typed.

- Consider $s = \textbf{new } C(\text{E})$ and the application of R3. Lemma 7 guarantees that the evaluation of object creation statements does not result in object representations which are not well-typed, so ρ_{i+1} and the successor configurations from the initialization of the new object are well-typed.

- For $s = \textbf{await}\ g$, either R4 or R5 may be applied, depending on the enabledness of g. If g is enabled and R4 is applied, the state variables do not change and ρ_{i+1} is well-typed. If g is not enabled and R5 is applied, the active process with the well-typed state L is moved to *PrQ* without modifying the state variables. The state of the active process is ε and hence also well-typed. Consequently, ρ_{i+1} is well-typed.

- For $s = r(Sig, Co, \mathrm{E}; \mathrm{V})$ and the application of R7, the state variables are not changed and ρ_{i+1} is well-typed.

- For $s = t!r(Sig, Co, \mathrm{E})$ and the application of R8, the statement reduces to $t := n; !r(Sig, Co, \mathrm{E})$. As n is of type Label, $\Gamma \vdash_S t := n$. The state variables are not changed and ρ_{i+1} is well-typed.

- For $s = !r(Sig, Co, \mathrm{E})$ and the application of R9 and R10, the state variables are not changed and ρ_{i+1} is well-typed.

- For $s = t?(\mathrm{V})$ and the application of R13, there is a message $comp(n\ \mathrm{E})$ in *EvQ* such that t is bound to n in o. It follows from Lemma 9 that $\Gamma(\mathrm{E}) \preceq \Gamma_V(\mathrm{V})$. Consequently, ρ_{i+1} is well-typed and the assignment of values E to variables V will preserve well-typedness.

- For $s = t?(\mathrm{V})$ and the application of R14, the active process with state L is suspended and a suspended process with state L' is activated. Since ρ_i is well-typed both L and L' are well-typed, and consequently ρ_{i+1} is well-typed.

- For $s = cont(n)$ and the application of R15, a process with state L' is activated. Since ρ_i is well-typed, L' is well-typed, and ρ_{i+1} is well-typed.

- For $s = return(\mathrm{E})$ and the application of R18, the state variables do not change and ρ_{i+1} is well-typed.

- For sequential composition $s = s_1; \mathrm{S}_1$, all cases are covered except $\varepsilon; \mathrm{S}_1$, which trivially reduces to S_1 by the left identity of sequential composition without modifying state variables. Consequently ρ_{i+1} is well-typed.

We now consider the remaining rewrite rules.

- The application of R6 activates a suspended process with a state L. Since ρ_i is well-typed, L is well-typed and, consequently, ρ_{i+1} is well-typed.

- Applying R11 or R12 does not modify state variables, and ρ_{i+1} is well-typed.

- For R16, observe that since ρ is well-typed a message $bind(m, Sig, Co, \mathrm{E}, o)$ must have been generated from a message $invoc(m, Sig, Co, \mathrm{E})$ **to** o by application of R12 and that the message $invoc(m, Sig, Co, \mathrm{E})$ **to** o must have been generated by R9 or R10 from a statement $!x.m(Sig, Co, \mathrm{E})$ or $!m(Sig, Co, \mathrm{E})$. Lemma 8 asserts that a call $!x.m(Sig, Co, \mathrm{E})$ or $!m(Sig, Co, \mathrm{E})$ in P can be type-correctly bound at runtime. Consequently, the conditional test $match(m, Sig, Co, \mathrm{M})$ will succeed in R16, resulting in a *bound* message. It follows that ρ_{i+1} is well-typed.

Syntactic categories.		*Definitions.*
g in Guard	v in Var	$g ::= wait \mid b \mid t? \mid g_1 \wedge g_2 \mid g_1 \vee g_2$
t in Label	s in Stm	$r ::= x.m \mid m$
m in Mtd	r in MtdCall	$\mathrm{S} ::= s \mid s; \mathrm{S}$
e in Expr	b in BoolExpr	$s ::= \textbf{skip} \mid (\mathrm{S}) \mid \mathrm{V} := \mathrm{E} \mid v := \textbf{new}\ C(\mathrm{E})$
x in ObjExpr		$\mid\ !r(\mathrm{E}) \mid t!r(\mathrm{E}) \mid t?(\mathrm{V}) \mid r(\mathrm{E};\mathrm{V}) \mid \textbf{while}\ b\ \textbf{do}\ \mathrm{S}\ \textbf{od}$
		$\mid \textbf{await}\ g \mid \textbf{await}\ t?(\mathrm{V}) \mid \textbf{await}\ r(\mathrm{E};\mathrm{V})$
		$\mid \mathrm{S}_1 \Box \mathrm{S}_2 \mid \mathrm{S}_1 \vert\vert\vert \mathrm{S}_2 \mid \textbf{if}\ b\ \textbf{then}\ \mathrm{S}_1\ \textbf{else}\ \mathrm{S}_2\ \textbf{fi}$

Figure 7.10: The language extended with constructs for local high-level control.

- For R17, since ρ is well-typed the message *bound*$(\langle \mathrm{S}, \mathrm{L} \rangle)$ **to** o comes from applying R16. Therefore the function *get* has instantiated the formal parameters V of some method m with actual values E such that the call is covered. Consequently $\Gamma(\mathrm{E}) \preceq \Gamma_{\mathrm{V}}(\mathrm{V})$ and L is well-typed. It follows that since o was well-typed in ρ_i, o is also well-typed in ρ_{i+1} and ρ_{i+1} is well-typed. ■

In a well-typed execution, objects only communicate by method calls. When a **new** $C(\mathrm{E})$ statement is evaluated in a well-typed configuration, no other objects of class C can be created in the same rewrite step, Consequently, the above theorem also applies to concurrent processes; i.e., the parallel reduction of a well-typed configuration ρ_i results in a well-typed configuration ρ_{i+1}.

7.4 Flexible High-Level Control Structures for Local Computation

Asynchronous method calls, as introduced in Section 7.3, add flexibility to method calls in the distributed setting because waiting activities may yield processor control to suspended and enabled processes. Further, this allows active and reactive behavior in a concurrent object to be naturally combined. However a more fine-grained control may be desirable, in order to allow different tasks within the same process to be selected depending on the order in which communication with other objects actually occur. For this purpose, additional composition operators between program statements are introduced: conditionals, while-loops, nondeterministic *choice*, and nondeterministic *merge*. The latter operators introduce high-level branching structures which allow the local computation in a concurrent object to take advantage of nondeterministic delays in the environment in a flexible way. By means of these operators, the local computation adapts itself to the distributed environment *without* yielding control to competing processes. For example, nondeterministic choice may be used to encode interrupts for process suspension such as timeout and race conditions between competing asynchronous calls [48].

7.4.1 Syntax

Statements can be composed in different ways, reflecting the requirements to the internal control flow in the objects. Recall that unguarded statements are always enabled, and that reply

$$\text{(MERGE)}\quad \frac{\Gamma \vdash_{\mathrm{S}} \mathrm{s} \langle \Delta \rangle \quad \Gamma \vdash_{\mathrm{S}} \mathrm{s}' \langle \Delta' \rangle \quad \mathrm{Dom}(\Delta_P) \cap \mathrm{Dom}(\Delta'_P) = \emptyset}{\Gamma \vdash_{\mathrm{S}} \mathrm{s} \| \| \mathrm{s}' \langle \Delta + \Delta' \rangle}$$

$$\text{(CHOICE)}\quad \frac{\Delta_{\mathrm{Sig}} \cup \Delta'_{\mathrm{Sig}} \neq \perp \quad \Gamma \vdash_{\mathrm{S}} \mathrm{s} \langle \Delta \rangle \quad \Gamma \vdash_{\mathrm{S}} \mathrm{s}' \langle \Delta' \rangle}{\Gamma \vdash_{\mathrm{S}} \mathrm{s} \Box \mathrm{s}' \langle (\Gamma + \Delta) \cup (\Gamma + \Delta') \rangle}$$

$$\text{(COND)}\quad \frac{\Gamma \vdash_{\mathrm{F}} \phi : \mathsf{Bool} \quad \Gamma \vdash_{\mathrm{S}} \mathrm{s} \Box \mathrm{s}' \langle \Delta \rangle}{\Gamma \vdash_{\mathrm{S}} \textbf{if } \phi \textbf{ then } \mathrm{s} \textbf{ else } \mathrm{s}' \textbf{ fi} \langle \Delta \rangle}$$

$$\text{(WHILE)}\quad \frac{\Gamma \vdash_{\mathrm{F}} \phi : \mathsf{Bool} \quad \Gamma_{\mathcal{F}} + \Gamma_{\mathrm{V}} + \Gamma_{I} + \Gamma_{C} \vdash_{\mathrm{S}} \mathrm{s} \langle \Delta \rangle}{\Gamma \vdash_{\mathrm{S}} \textbf{while } \phi \textbf{ do } \mathrm{s} \textbf{ od} \langle \Delta_{\mathrm{Sig}} \rangle}$$

Figure 7.11: Typing of local high-level control structures.

statements $t?(\mathrm{v})$ may block. Let s_1 and s_2 be statements. Nondeterministic choice between statements s_1 and s_2, written $\mathrm{s}_1 \Box \mathrm{s}_2$, may compute s_1 once s_1 is ready or s_2 once s_2 is ready, and suspends if neither branch is enabled. (Remark that to avoid deadlock the semantics additionally will not commit to a branch which starts with a blocking reply statement.) Nondeterministic merge, written $\mathrm{s}_1 \| \| \mathrm{s}_2$, evaluates the statements s_1 and s_2 in some interleaved and enabled order, and suspends if neither branch is enabled. Control flow without potential processor release uses **if** and **while** constructs. The conditional **if** g **then** s_1 **else** s_2 **fi** selects s_1 if g evaluates to true and otherwise s_2, and the loop **while** g **do** s_1 **od** repeats s_1 until g is not true. Figure 7.10 gives the extended language syntax.

7.4.2 Typing

The type system introduced in Section 7.3 is now extended to account for choice, merge, conditional, and while statements. It is nondeterministic for choice statements which branch will be executed at runtime, and for merge statements the order in which the statements of the different branches are executed. Consequently, the branches must be type checked in the same typing environment. The typing environment resulting from a nondeterministic statement depends on the calls introduced and removed in each branch. The additional rules are given in Figure 7.11. We now define a union operator for typing environments (recall that Γ_I, Γ_C, and $\Gamma_{\mathcal{F}}$ are static tables and that Γ_{V} is not modified by the type checking of program statements).

Definition 12 Let Γ and Δ be typing environments and let N and M be sets. The commutative *union of typing environments* $\Gamma \cup \Delta$ is defined as $\Gamma_{\mathcal{F}} + \Gamma_I + \Gamma_C + \Gamma_{\mathrm{V}} + (\Gamma_{Sig} \cup \Delta_{Sig}) + (\Gamma_P \cup \Delta_P)$. The *union of signature mappings*, $\Gamma_{Sig} \cup \Delta_{Sig}$, is defined as follows:

$$\begin{aligned}
&(\Gamma_{Sig} + [t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \to T_2 \rangle] + \Gamma'_{Sig}) \cup (\Delta_{Sig} + [t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \to T'_2 \rangle] + \Delta'_{Sig})\\
&\quad = (\Gamma_{Sig} + \Gamma'_{Sig} \cup \Delta_{Sig} + \Delta'_{Sig}) + [t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \to T_2 \rangle] \oplus [t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \to T'_2 \rangle]\\
&\quad \text{if} \quad t_i \notin \mathrm{Dom}(\Gamma'_{Sig} + \Delta'_{Sig}),
\end{aligned}$$

$$\begin{aligned}
&(\Gamma_{Sig} + [t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \to T_2 \rangle] + \Gamma'_{Sig}) \cup \Delta_{Sig}\\
&\qquad = (\Gamma_{Sig} + \Gamma'_{Sig} \cup \Delta_{Sig}) + [t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \to T_2 \rangle] \quad \text{if} \quad \Delta_{Sig}(t_i) = \perp,
\end{aligned}$$

$$\emptyset_{Sig} \cup \Delta_{Sig} = \Delta_{Sig}.$$

Let $\bot + \Gamma_{Sig} = \bot$. The operator $\oplus$ on signature mappings is defined as

$$\begin{array}{l} [t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \rightarrow T_2 \rangle] \\ \quad \oplus [t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \rightarrow T_2' \rangle] \end{array} = \begin{cases} [t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \rightarrow (T_2 \cap T_2') \rangle] & \text{if } \; T_2 \cap T_2' \neq \bot, \\ \bot & [\textbf{otherwise}] \end{cases}$$

The *union of pending mappings*, $\Gamma_P \cup \Delta_P$, is defined as follows:

$$\begin{array}{l} (\Gamma_P + [t \overset{P}{\mapsto} N] + \Gamma_P') \cup (\Delta_P + [t \overset{P}{\mapsto} M] + \Delta_P') \\ \qquad = (\Gamma_P + \Gamma_P' \cup \Delta_P + \Delta_P') + [t \overset{P}{\mapsto} N] \otimes [t \overset{P}{\mapsto} M] \quad \text{if} \quad t \notin \mathrm{Dom}(\Gamma_P' + \Delta_P'), \end{array}$$

$$(\Gamma_P + [t \overset{P}{\mapsto} N] + \Gamma_P') \cup \Delta_P = [t \overset{P}{\mapsto} \emptyset] + (\Gamma_P + \Gamma_P' \cup \Delta_P) \quad \text{if} \quad \Delta_P(t) \in \{\bot, \emptyset\}.$$

The operator $\otimes$ on pending mappings is defined as

$$[t \overset{P}{\mapsto} N] \otimes [t \overset{P}{\mapsto} M] = \begin{cases} [t \overset{P}{\mapsto} N \cup M] & \text{if} \quad N \neq \emptyset \wedge M \neq \emptyset, \\ [t \overset{P}{\mapsto} \emptyset] & [\textbf{otherwise}]. \end{cases}$$

In a merge statement $s_1 ||| s_2$, both statement lists will be evaluated. The MERGE rule ensures that reply statements in the two branches do not correspond to the same pending call. Furthermore all asynchronous invocations introduced in a merge statement must have unique labels to avoid interference between calls in the two branches, caused by the interleaved execution of s_1 and s_2. We then have that for each labeled invocation, the invocation cannot be matched by a reply statement in another branch of the merge. For this purpose, the MERGE rule ensures that $\mathrm{Dom}(\Delta_P) \cap \mathrm{Dom}(\Delta_P') = \emptyset$.

For nondeterministic choice, at most one of the two branches is evaluated. If only one branch has a reply statement which corresponds to a pending call in Γ then a reply statement corresponding to the same call is not allowed in statements succeeding the nondeterministic choice, although the branch with the reply statement need not be chosen at runtime. Moreover, if there is a reply statement in each branch that corresponds to the same pending call in Γ, say $t?(v)$ and $t?(v')$, the CHOICE rule asserts that the types of the out-parameters have a common subtype (i.e., $\Gamma_V(v) \cap \Gamma_V(v') \neq \bot$) in order to ensure a deterministic signature for the invocation. This property is ensured by $\Delta_{\mathrm{Sig}} \cup \Delta_{\mathrm{Sig}}' \neq \bot$, which compares the types of the actual out-parameters of the reply statements in the branches when they correspond to the same pending call in Γ. The subtype $\Gamma_V(v) \cap \Gamma_V(v')$ is then selected as the type of the out-parameter of the call. Note that if a label is reused in a new asynchronous invocation, the reply statement in this branch will refer to the new invocation and Data is used as the type for the out-parameters of the previous call in this branch. Furthermore, given two branches s and s' in a nondeterministic choice, the type system ensures that a reply statement with label t can occur after the nondeterministic choice statement only if a call with label t is pending after the evaluation of either s or s'. Consequently, each reply statement corresponds to a call independent of the branch selection.

The WHILE rule ensures that reply statements in the body s of the while-loop must correspond to invocations in the same traversal of s. Similarly, calls initiated in s must have their corresponding reply statements within the same traversal of s. This is guaranteed by the fact that the WHILE rule does not update the pending mapping Γ_P.

Lemma 11 *Let Γ be a mapping family such that $\Gamma_P = \varepsilon$. For any statement list* s *in the code of an arbitrary method body with a type judgment $\Gamma \vdash_S \mathrm{s} \langle \Delta \rangle$, the set Δ_P contains exactly the labeled method invocations for which the return value may still be assigned to program variables after* s.

Proof. The proof is by induction over the length of s, similar to the proof of Lemma 4. For the old cases, the signatures and cointerfaces in a single branch of a statement list follow from Lemma 6. The new cases are now considered for the induction step. We assume given $\Gamma \vdash_S \mathrm{s} \langle \Delta \rangle$ where Δ_P contains the labeled method invocations for which the return values may still be assigned to program variables after s. We prove that for a judgment $\Gamma \vdash_S \mathrm{s};s \langle \Delta + \Delta' \rangle$, where s is a nondeterministic choice, merge, conditional, or while statement, $(\Delta + \Delta')_P$ contains the labeled method invocations for which the return values may still be assigned to program variables after $\mathrm{s};s$.

- For the while statement **while** ϕ **do** s' **od**, the type system does not allow any updates on the pending mapping Γ_P, so $(\Delta + \Delta')_P = \Delta_P$.

- For nondeterministic choice $\mathrm{s}_1 \Box \mathrm{s}_2$, the induction hypothesis gives us $\Gamma + \Delta \vdash_S \mathrm{s}_1 \langle \Delta_1 \rangle$ and $\Gamma + \Delta \vdash_S \mathrm{s}_2 \langle \Delta_2 \rangle$ such that the lemma holds. There are two cases:

  Case 1: s_1 contains a reply statement $t?(\mathrm{v}_1)$ with a label t corresponding to a call on t_i such that $i \in \Delta_P(t)$. There are two possibilities. First, a reply statement with label t does not occur in s_2. The pending call is type checked with the new out-parameter type in s_1, and the update of the typing environment removes the pending calls; i.e., $(\Delta_1 \cup \Delta_2)_P(t) = \emptyset$. Signature and cointerface uniqueness is here immediate. Second, a reply statement $t?(\mathrm{v}_2)$ occurs in s_2. We must check that the reply statements yield a unique signature and cointerface. By the induction hypothesis, there is a unique signature candidate in each branch. $(\Delta_1 \cup \Delta_2)_{\mathrm{Sig}} \neq \bot$ evaluates to true if a compatible minimal type can be given for the two reply statements, which gives us a unique signature and cointerface. Note that if the reply statement in one branch refers to a new method invocation with the same label t in that branch, Data is used as the out-parameter type for the first call in that branch. The update of the typing environment removes the pending calls.

  Case 2: s_1 introduces a new invocation with label t. The effect of the CHOICE rule records the pending call if there is also a pending call to t if s_2 is chosen. (This pending call after s_2 may either correspond to an invocation in s_2 or a pending call in Δ which has been overwritten in s_1.) The invocations from both branches are captured in the effect of the CHOICE rule $(\Delta + \Delta_1) \cup (\Delta + \Delta_2)$. It follows from the induction hypothesis that $\Delta + (\Delta + \Delta_1)_P \cup (\Delta + \Delta_2)_P$ contains the labeled invocations for which the return values may be assigned to program variables after $\mathrm{s};\mathrm{s}_1 \Box \mathrm{s}_2$.

- The conditional statement follows from the case for nondeterministic choice.

- For nondeterministic merge $\mathrm{s}_1 ||| \mathrm{s}_2$ the induction hypothesis is that for $\Gamma + \Delta \vdash_S \mathrm{s}_1 \langle \Delta_1 \rangle$ and $\Gamma + \Delta \vdash_S \mathrm{s}_2 \langle \Delta_2 \rangle$, $(\Delta_1)_P$ and $(\Delta_2)_P$ contain exactly the labeled method invocations for which the return value may still be assigned to program variables after s_1 and s_2. The condition $\mathrm{Dom}((\Delta_1)_P) \cap \mathrm{Dom}((\Delta_2)_P) = \emptyset$, ensures that labels used in branches s_1 and s_2

of the merge statement are non-overlapping, so a call with label t cannot be matched by a reply statement in another branch. Consequently, Δ_1 and Δ_2 are disjoint, so $\Delta_1+\Delta_2 = \Delta_2+\Delta_1$ and the order in which the updates are applied to Δ is insignificant. It then follows directly from the induction hypothesis that $(\Delta+\Delta_1+\Delta_2)_P$ contains exactly the labeled method invocations for which the return value may still be assigned to program variables after $\mathrm{s};\mathrm{s}_1|||\mathrm{s}_2$. ■

We show that every call is covered for the derived signature and cointerface.

Lemma 12 *In a well-typed method, a signature and cointerface can be derived for every method invocation in the body, such that the invocation is covered.*

Proof. The proof extends the proof of Lemma 6. Let s be the code of an arbitrary well-typed method, such that $\Gamma \vdash_S \mathrm{s} \langle \Delta \rangle$. The proof is by induction over the length of s. Let $\mathrm{s} = \mathrm{s}_0; s$ and assume as induction hypothesis that well-typed signatures and cointerfaces have been derived for every invocation in $\Gamma \vdash_S \mathrm{s}_0 \langle \Delta_{\mathrm{s}_0} \rangle$. We show that well-typed signatures and cointerfaces have been derived for every invocation in $\Gamma \vdash_S \mathrm{s}_0; s \langle \Delta \rangle$. Here, only the cases for choice, merge, conditional, and while statements are considered.

- For $s = $ **while** ϕ **do** s' **od**, by the induction hypotheses every invocation introduced in s_0 and s' has a well-typed signature and cointerface. Traversing s' does not modify the pending mapping, signatures in $(\Delta_{\mathrm{s}_0})_{Sig}$ are not refined after $\mathrm{s}_0; s$. Well-typed signatures and cointerfaces for invocations in $\mathrm{s}_0; s$ follow from the induction hypotheses and the invocations are covered.

- For $s = \mathrm{s}_1 \Box \mathrm{s}_2$, by the induction hypothesis, the signature and cointerface of every invocation in s_1 and s_2 are derived. By rule CHOICE, if one of the two branches contains a reply statement to an invocation in s_0, the signature of the invocation in $(\Delta_{\mathrm{s}_0})_{Sig}$ will be refined in Δ_{Sig}. If both branches contain such reply statements, then the signature is refined in Δ_{Sig} with a common subtype, such that the invocation is covered with the new signature and the cointerface, independent of the branch being evaluated.

- The conditional statement follows from the case for nondeterministic choice.

- For $s = \mathrm{s}_1|||\mathrm{s}_2$, the signature and cointerface for every invocation in s_1 and s_2 is derived by the induction hypothesis. The signatures in $(\Delta_{\mathrm{s}_0})_{Sig}$ may be refined if the corresponding reply statement is contained in the merge statement. By rule MERGE, reply statements in the two branches do not correspond to the same pending call, so the sets of labels used in the two branches are disjoint. Consequently, the signature can only be refined once in Δ_{Sig}, and the invocation is covered by the new signature. Regardless of the order of the evaluation of branches, every call in $\mathrm{s}_0; s$ is covered. ■

7.4.3 Operational Semantics

The operational semantics of Section 7.3.5 is extended with rules for the local control structures in Figure 7.12. The selection of a branch s_1 in a nondeterministic choice statement $\mathrm{s}_1 \Box \mathrm{s}_2$ is modeled by R19. Combined with the associativity and commutativity of the $\Box$ operator, this

(R19) $\langle o : Ob \,|\, Att : \mathrm{A}, Pr : \langle (\mathrm{S}_1 \Box \mathrm{S}_2); \mathrm{S}_3, \mathrm{L}\rangle, EvQ : \mathrm{Q}\rangle$
$\longrightarrow \langle o : Ob \,|\, Att : \mathrm{A}, Pr : \langle \mathrm{S}_1; \mathrm{S}_3, \mathrm{L}\rangle, EvQ : \mathrm{Q}\rangle$ **if** $ready(\mathrm{S}_1, (\mathrm{A};\mathrm{L}), \mathrm{Q})$

(R20) $\langle o : Ob \,|\, Att : \mathrm{A}, Pr : \langle (\mathrm{S}_1 ||| \mathrm{S}_2); \mathrm{S}_3, \mathrm{L}\rangle, EvQ : \mathrm{Q}\rangle$
$\longrightarrow \langle o : Ob \,|\, Att : \mathrm{A}, Pr : \langle (\mathrm{S}_1 /// \mathrm{S}_2); \mathrm{S}_3, \mathrm{L}\rangle, EvQ : \mathrm{Q}\rangle$ **if** $ready(\mathrm{S}_1, (\mathrm{A};\mathrm{L}), \mathrm{Q})$

(R21) $\langle o : Ob \,|\, Att : \mathrm{A}, Pr : \langle ((s; \mathrm{S}_1) /// \mathrm{S}_2); \mathrm{S}_3, \mathrm{L}\rangle, EvQ : \mathrm{Q}\rangle$
$\longrightarrow$ **if** $enabled(s, (\mathrm{A};\mathrm{L}), \mathrm{Q})$
then $\langle o : Ob \,|\, Att : \mathrm{A}, Pr : \langle s; (\mathrm{S}_1 /// \mathrm{S}_2); \mathrm{S}_3, \mathrm{L}\rangle, EvQ : \mathrm{Q}\rangle$
else $\langle o : Ob \,|\, Att : \mathrm{A}, Pr : \langle ((s; \mathrm{S}_1) ||| \mathrm{S}_2); \mathrm{S}_3, \mathrm{L}\rangle, EvQ : \mathrm{Q}\rangle$ **fi**

(R22) $\langle o : Ob \,|\, Att : \mathrm{A}, Pr : \langle (\mathit{if}\; b\; \mathit{then}\; \mathrm{S}_1\; \mathit{else}\; \mathrm{S}_2\; \mathit{fi}; \mathrm{S}_3), \mathrm{L}\rangle\rangle$
$\longrightarrow$ **if** $eval(b, (\mathrm{A};\mathrm{L}))$ **then** $\langle o : Ob \,|\, Att : \mathrm{A}, Pr : \langle (\mathrm{S}_1; \mathrm{S}_3), \mathrm{L}\rangle \rangle$
else $\langle o : Ob \,|\, Att : \mathrm{A}, Pr : \langle (\mathrm{S}_2; \mathrm{S}_3), \mathrm{L}\rangle \rangle$ **fi**

(R23) $\langle o : Ob \,|\, Att : \mathrm{A}, Pr : \langle (\mathit{while}\; b\; \mathit{do}\; \mathrm{S}_1\; \mathit{od}; \mathrm{S}_2), \mathrm{L}\rangle\rangle$
$\longrightarrow \langle o : Ob \,|\, Att : \mathrm{A}, Pr : \langle (\mathit{if}\; b\; \mathit{then}\; (\mathrm{S}_1; \mathit{while}\; b\; \mathit{do}\; \mathrm{S}_1\; \mathit{od})\; \mathit{else}\; \varepsilon\; \mathit{fi}); \mathrm{S}_2, \mathrm{L}\rangle \rangle$ **fi**

Figure 7.12: An operational semantics for local high-level control structures.

rule covers the selection of any branch in a compound nondeterministic choice statement. Here, the *ready* predicate tests if a process is ready to execute; i.e., the process does not immediately need to wait for a guard to become true or for a completion message. The *ready* predicate is defined as follows:

$$ready(s; \mathrm{S}, \mathrm{D}, \mathrm{Q}) = ready(s, \mathrm{D}, \mathrm{Q})$$
$$ready(t?(\mathrm{V}), \mathrm{D}, \mathrm{Q}) = eval(t, \mathrm{D})\; in\, \mathrm{Q}$$
$$ready(s, \mathrm{D}, \mathrm{Q}) = enabled(s, \mathrm{D}, \mathrm{Q}) \qquad [\textbf{otherwise}].$$

As long as neither S_1 nor S_2 is ready, the active process is blocked if enabled and suspended if not enabled. Consequently, selecting a branch which immediately blocks or suspends execution is avoided if possible.

The merge statement $\mathrm{S}_1 ||| \mathrm{S}_2$ interleaves the execution of two statement lists S_1 and S_2. A naive approach is to define merge in terms of the nondeterministic choice $\mathrm{S}_1; \mathrm{S}_2 \Box \mathrm{S}_2; \mathrm{S}_1$. To improve efficiency, a more fine-grained interleaving is preferred. However, in order to comply with the suspension technique of the language, interleaving is only allowed at processor release points in the branches. An associative but not commutative auxiliary operator $///$ is introduced in R20 and R21. The latter rule has the following property: Whenever evaluation of the selected (left) branch leads to non-enabledness, execution has arrived at a suspension point and it is safe to pass control back to the $|||$ operator. Rule R20 for merge decides whether to block or select a branch. (Suspension is handled by R5.) The $|||$ operator is associative, commutative, and has identity element ε (i.e., $\varepsilon ||| \mathrm{S} = \mathrm{S}$). The operational semantics for conditionals (R22) and while-loops (R23) are as expected. Finally, the enabledness predicate is extended to nondeterministic choice and merge as follows:

$$enabled(\mathrm{S} \Box \mathrm{S}', \mathrm{D}, \mathrm{Q}) = enabled(\mathrm{S}, \mathrm{D}, \mathrm{Q}) \vee enabled(\mathrm{S}', \mathrm{D}, \mathrm{Q})$$
$$enabled(\mathrm{S} ||| \mathrm{S}', \mathrm{D}, \mathrm{Q}) = enabled(\mathrm{S}, \mathrm{D}, \mathrm{Q}) \vee enabled(\mathrm{S}', \mathrm{D}, \mathrm{Q})$$

7.4.4 Type Soundness

The soundness of the type system for the extended language is established in this section. By Lemma 12, the signature and cointerface of every method invocation is deterministically given by the extended type system. Consequently, Lemmas 8 and 9 hold for the extended language. We first show a property of the execution sequences of nondeterministic merge.

Lemma 13 *Let* $\mathrm{s}_1|||\mathrm{s}_2$ *be well-typed in a typing environment* Γ *and let* s *be an interleaving of* s_1 *and* s_2*. Then* s *is well-typed in* Γ*.*

Proof. Let $\Gamma \vdash_{\mathrm{S}} \mathrm{s}_1 \langle \Delta_1 \rangle$ and $\Gamma \vdash_{\mathrm{S}} \mathrm{s}_2 \langle \Delta_2 \rangle$ such that $\Gamma \vdash_{\mathrm{S}} \mathrm{s}_1|||\mathrm{s}_2 \langle \Delta_1{+}\Delta_2 \rangle$. The proof is by induction over the length of s. For $\mathrm{s} = \varepsilon$, s is trivially well-typed. For the induction step, let $\mathrm{s};s_i;\ldots;s_1$ be an interleaving of s_1 and s_2 such that $\Gamma \vdash_{\mathrm{S}} \mathrm{s};s_i;\ldots;s_1 \langle \Delta \rangle$. We prove that $\mathrm{s};s;s_i;\ldots;s_1$ is well-typed if $(\mathrm{s}_1;s)|||\mathrm{s}_2$ is well-typed. Let $\Gamma \vdash_{\mathrm{S}} \mathrm{s}_1;s \langle \Delta_1' \rangle$ such that $\Gamma \vdash_{\mathrm{S}} (\mathrm{s}_1;s)|||\mathrm{s}_2 \langle \Delta_1'{+}\Delta_2 \rangle$. (The case for $\mathrm{s}_2;s$ is similar.) The proof proceeds by induction over i and by case analysis over s. Observe that $s_i;\ldots;s_1$ $(i \geq 0)$ is a tail sequence from s_2 and that for well-typedness, only the statements $t!p(\mathrm{E})$, $t?(\mathrm{V})$, and **await** $t?$ actually depend on the dynamically decided typing environment. We consider $s = t?(\mathrm{V})$.

- For $i = 0$, we have the interleaving $\mathrm{s};s$. Since $\Gamma \vdash_{\mathrm{S}} \mathrm{s}_1;s \langle \Delta_1' \rangle$, it follows that $(\Gamma + \Delta_1)_{\mathrm{V}}(\mathrm{V}) \neq \bot$ and $(\Gamma+\Delta_1)_P(t) \neq \emptyset$, and hence $(\Gamma+\Delta_1)_{\mathrm{V}}(t) = \mathsf{Label}$. Since $(\Gamma+\Delta_1)_{\mathrm{V}} = (\Gamma+\Delta)_{\mathrm{V}}$, it follows that t and V are declared variables in $\Gamma+\Delta$. Since $\Gamma \vdash_{\mathrm{S}} \mathrm{s}_1|||\mathrm{s}_2 \langle \Delta_1 + \Delta_2 \rangle$, $\mathrm{Dom}((\Delta_1)_P) \cap \mathrm{Dom}((\Delta_2)_P) = \emptyset$. Consequently, $(\Gamma + \Delta_1)_P(t) = (\Gamma + \Delta_1 + \Delta_2)_P(t) = (\Gamma+\Delta)_P(t)$ and $\mathrm{s};s$ is well-typed in Γ.

- For $i+1$, the induction hypothesis is that $\mathrm{s};s_{i+1};s;s_i;\ldots;s_1$ is a well-typed interleaving and s_{i+1} is a statement from s_2. Since s_{i+1} is a statement from s_2, $\mathrm{Dom}((\Delta_1)_P)(t) \neq \bot$, and $\mathrm{Dom}((\Delta_1)_P) \cap \mathrm{Dom}((\Delta_2)_P) = \emptyset$, exchanging s and s_{i+1} does not affect well-typedness and $\mathrm{s};s;s_{i+1};s_i;\ldots;s_1$ is well-typed.

The cases for $t!p(\mathrm{E})$ and **await** $t?$ are similar, the other cases are straightforward. ■

It follows from Lemma 13 that if $(\mathrm{s}_1;\mathrm{s}_2)|||\mathrm{s}_3$ is well-typed in a typing environment Γ, then so is $\mathrm{s}_1;(\mathrm{s}_2|||\mathrm{s}_3)$.

Theorem 14 (Type soundness) *All executions of programs starting in a well-typed initial configuration, are well-typed.*

Proof. We consider a well-typed execution ρ of a program P. The proof is by the induction over the length of $\rho = \rho_0, \rho_1, \ldots$ and extends the proof of Theorem 10. We show that for any well-typed configuration ρ_i, the successor configuration ρ_{i+1} is also well-typed, by case analysis over the rewrite rules. Here, only the new rewrite rules are considered, i.e., the reduction of a well-typed runtime object $\langle o : Ob|\, Att\colon \mathrm{A}, Pr\colon \langle s; \mathrm{S}, \mathrm{L}\rangle\rangle$ to $\langle o : Ob|\, Att\colon \mathrm{A}', Pr\colon \langle s'; \mathrm{S}, \mathrm{L}\rangle\rangle$ where s is a nondeterministic choice, merge, conditional, or while statement. (The remaining cases are covered by the proof of Theorem 10.) For the induction hypothesis, we assume that the type soundness property holds for well-typed branches s_1 and s_2 of s.

- For $s = \mathrm{s}_1 \Box \mathrm{s}_2$, the application of R19 reduces s to either s_1 or s_2. As the program is well-typed, $\Gamma \vdash_{\mathrm{S}} \mathrm{s}_1$ and $\Gamma \vdash_{\mathrm{S}} \mathrm{s}_2$. The state variables do not change and ρ_{i+1} is well-typed.

- Consider $s = \mathrm{S}_1 \mathbin{|||} \mathrm{S}_2$. If $\mathrm{S}_1 = \varepsilon$ then $\mathrm{S}_1 \mathbin{|||} \mathrm{S}_2 = \mathrm{S}_2$, for which type soundness holds by assumption. Now assume that S_1 is nonempty. By R20, $\mathrm{S}_1 \mathbin{|||} \mathrm{S}_2$ reduces to $\mathrm{S}_1 \mathbin{///} \mathrm{S}_2$, so ρ_{i+1} is well-typed since state variables do not change.

- For $\mathrm{S}_1 \mathbin{///} \mathrm{S}_2$, we proceed by induction over the number n of rewrite steps using R21 immediately preceding ρ_i. If $n = 0$, ρ_i must have been obtained by application of R20. Let $\mathrm{S}_1 = \mathrm{S}_1'; \mathrm{S}_1''$. By the induction hypothesis $\mathrm{S}_1 \mathbin{|||} \mathrm{S}_2$ is well-typed and by Lemma 13 any $\mathrm{S}_1'; (\mathrm{S}_1'' \mathbin{|||} \mathrm{S}_2)$ is well-typed. For the induction step, observe that the repeated application of R21 must eventually result in a statement $\mathrm{S}_1'' \mathbin{|||} \mathrm{S}_2$ in a state $\rho_{i+n'}$ $(n' \geq n)$ after executing S_1'. It follows from the induction hypothesis and Lemma 13 that all states $\rho_i, \ldots, \rho_{i+n'}$ are well-typed.

- For $s =$ **if** ϕ **then** S_1 **else** S_2 **fi**, the application of R22 reduces s to either S_1 or S_2. Since the program is well-typed, we know that $\Gamma \vdash_{\mathrm{F}} \phi : \mathsf{Bool}$, $\Gamma \vdash_{\mathrm{S}} \mathrm{S}_1$, and $\Gamma \vdash_{\mathrm{S}} \mathrm{S}_2$. The functional expression ϕ will be successfully reduced to a Boolean value, as expected by the conditional test. The state variables do not change and ρ_{i+1} is well-typed.

- For $s =$ **while** E **do** S_1 **od**, the application of R23 reduces s to $s' =$ **if** E **then** $(\mathrm{S}_1;$**while** E **do** S_1 **od**$)$ **else** ε **fi**. The typing rule WHILE ensures that a traversal of S_1 does not modify the pending mapping Γ_P. Consequently, $\mathrm{S}_1;$**while** E **do** S_1 **od** is well-typed. Moreover, since there is no interference between the different traversals of S_1, the conditions of rule CHOICE are satisfied and s' is also well-typed. The state variables do not change and ρ_{i+1} is well-typed. ■

7.5 An Extension with Multiple Inheritance

Many languages identify the subclass and subtype relations, in particular for parameter passing, although several authors argue that inheritance relations for code and for behavior should be distinct [23, 5, 14, 73]. From a pragmatic point of view, combining these relations leads to severe restrictions on code reuse which seem unattractive to programmers. From a reasoning perspective, the separation of these relations allows greater expressiveness while providing type-safety. In order to solve the conflict between unrestricted code reuse in subclasses, and behavioral subtyping and incremental reasoning control [55, 73], we use interfaces to type object variables and external calls. Multiple inheritance is allowed for both interfaces and classes. Whereas subinterfacing is restricted to a form of behavioral subtyping, subclassing is unrestricted in the sense that implementation claims (and class invariants) are not in general inherited. However, the mutual dependencies introduced by cointerfaces makes the inheritance of contracts necessary in subclasses.

A class describes a collection of objects with similar internal structure; i.e., attributes and method definitions. Class inheritance is a powerful mechanism for defining, specializing, and understanding the imperative class structures through code reuse and modification. Class extension and method redefinition are convenient both for the development and understanding of code. Calling superclass methods in a subclass method enables *reuse in redefined methods*, while method redefinition allows *specialization* in subclasses.

With distinct inheritance and subtyping hierarchies, class inheritance could allow a subset of the attributes and methods of a class to be inherited. However, this would require considerable work establishing invariants for parts of the superclass that appear desirable for inheritance, either anticipating future needs or while designing subclasses. The *encapsulation principle* for class inheritance states that it should suffice to work at the subclass level to ensure that the subclass is well-behaved when inheriting from a superclass: Code design as well as new proof obligations should occur in the subclass only. Situations that break this principle are called inheritance anomalies [57, 62] (see also the fragile base class problem [61]). Reasoning considerations therefore suggest that all attributes and methods of a superclass are inherited, but method redefinition may violate the requirements of the interfaces of the superclass.

7.5.1 Syntax

A mechanism for multiple inheritance at the class level is now considered, where all attributes and methods of a superclass are inherited by the subclass, and where superclass methods may be redefined. In the syntax the keyword **inherits** is introduced followed by a list of instantiated class names $C(\mathrm{E})$, where E provides the actual class parameters.

Let a class hierarchy be a directed acyclic graph of classes. Each class consists of lists of class parameters and instantiated class names (for superclasses), a set of attributes, and method definitions. Let a class C be *below* a class C' if C is C', or if C is a direct or indirect subclass of C' and *above* C' if C is C', or if C is a direct or indirect superclass of C'. The encapsulation provided by interfaces suggests that external calls to an object of class C are virtually bound to the closest method definition above C. However, the object may internally invoke methods of its superclasses. In the setting of multiple inheritance and overloading, ambiguities may occur when attributes or methods are accessed. A name conflict is *vertical* if a name occurs in a class and in one of its ancestors, and *horizontal* if the name occurs in distinct branches of the graph. Vertical name conflicts for method names are resolved in a standard way: the first definition matching the types of the actual parameters is chosen while ascending a branch of the inheritance tree. Horizontal name conflicts are resolved dynamically depending on the class of the object and the context of the call.

Qualified Names

Qualified names may be used to internally refer to an attribute or method in a class in a unique way. For this purpose, we adapt the **qua** construct of Simula [26] to the setting of multiple inheritance with virtual binding. For an attribute v or a method m declared in a class C, we denote by $v@C$ and $m@C$ the qualified names which provide static references to v and m. By extension, if v or m is *not* declared in C, but inherited from the superclasses of C, the qualified reference $m@C$ binds as an unqualified reference m above C.

Attribute names are not visible through an object's external interfaces. Consequently, attribute names should not be merged if inheritance leads to name conflicts and attributes of the same name should be allowed in different classes of the inheritance hierarchy [72]. In order to allow the reuse of attribute names, these are always expanded into qualified names. This is desirable in order to avoid runtime errors that may occur if methods of superclasses assign

Syntactic categories.		*Definitions.*
g in Guard	v in Var	$g ::= \mathit{wait} \,\vert\, b \,\vert\, t? \,\vert\, g_1 \wedge g_2 \,\vert\, g_1 \vee g_2$
t in Label	s in Stm	$r ::= x.m \,\vert\, m \,\vert\, m@C \,\vert\, m < C$
m in Mtd	r in MtdCall	$\mathrm{S} ::= s \,\vert\, s;\mathrm{S}$
e in Expr	b in BoolExpr	$s ::= \mathbf{skip} \,\vert\, (\mathrm{S}) \,\vert\, \mathrm{V} := \mathrm{E} \,\vert\, v := \mathbf{new}\ C(\mathrm{E})$
x in ObjExpr		$\vert\ !r(\mathrm{E}) \,\vert\, t!r(\mathrm{E}) \,\vert\, t?(\mathrm{V}) \,\vert\, r(\mathrm{E};\mathrm{V}) \,\vert\, \mathbf{while}\ b\ \mathbf{do}\ \mathrm{S}\ \mathbf{od}$
		$\vert\ \mathbf{await}\ g \,\vert\, \mathbf{await}\ t?(\mathrm{V}) \,\vert\, \mathbf{await}\ r(\mathrm{E};\mathrm{V})$
		$\vert\ \mathrm{S}_1 \Box \mathrm{S}_2 \,\vert\, \mathrm{S}_1 \vert\vert\vert \mathrm{S}_2 \,\vert\, \mathbf{if}\ b\ \mathbf{then}\ \mathrm{S}_1\ \mathbf{else}\ \mathrm{S}_2\ \mathbf{fi}$

Figure 7.13: A language extension for static and virtual internal calls.

to overloaded attributes. This convention has the following consequence: unlike C++, there is no duplication of attributes when branches in the inheritance graph have a common superclass. Consequently, if multiple copies of the superclass' attributes are needed, one has to rely on delegation techniques.

Instantiation of Attributes

At object creation time, attributes are collected from the object's class and superclasses. Recall that an attribute in a class C is declared by $x : T = e$, where x is the name of the attribute, T its type, and e its initial value. The expression e may refer to the values of inherited attributes by means of qualified references, in addition to the values of the class parameter variables V. The initial state values of an object of class C then depend on the actual parameter values bound to V. These may be passed as actual class parameter values to inherited classes in order to derive values for the inherited attributes, which in turn may be used to instantiate the locally declared attributes.

Accessing Inherited Attributes and Methods

If C is a superclass of C' we introduce the syntax $m@C(\mathrm{E};\mathrm{V})$ for synchronous internal invocation of a method above C in the inheritance graph, and similarly for external and asynchronous invocations. These calls may be bound without knowing the exact class of *self*, so they are called *static*, in contrast to calls without @, called *virtual*. We assume that attributes have unique names in the inheritance graph; this may be enforced at compile-time by extending each attribute name x with the name of the class C in which it is declared, which implies that attributes are bound statically. Consequently, a method declared in a class C may only access attributes declared above C. In a subclass, an attribute x of a superclass C is accessed by the qualified reference $x@C$. The extended language syntax is given in Figure 7.13.

7.5.2 Virtual Binding

When multiple inheritance is included in the language, it is necessary to reconsider the mechanism for virtual binding. A method declaration in a class C is *constrained* by C' if C is required to be below C'. The virtual binding of method calls is now explained. At runtime, a call to a

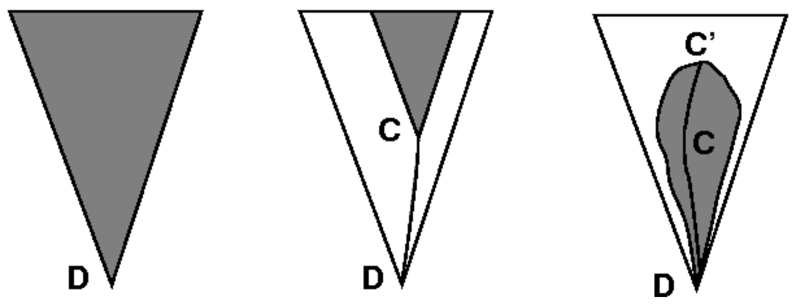

Figure 7.14: Binding calls to m, $m@C$, and $m < C$ in an object of class D.

method of an object o is always bound above the class of o. Let m be a method declared in an interface I and let o be an instance of a class C implementing I. There are two cases:

1. m is called *externally*, in which case C is not statically known. In this case, C is dynamically identified as the class of o.

2. m is called *internally* from C', a class above the actual class C of o. In this case static analysis identifies the call with a declaration of m above C', say in C''. Consequently we let the call be constrained by C'', and compilation replaces the reference to m with a reference to $m < C''$.

The dynamically decided context of a call may eliminate parts of the inheritance graph above the actual class of the callee with respect to the binding of a specific call. If a method name is ambiguous within the dynamic constraint, we assume that any solution is acceptable. For a natural and simple model of priority, the call is bound to the first matching method definition above C, in a left-first depth-first order as given by the textual declarations of the instantiated class names. (An arbitrary order may be obtained by replacing the list of instantiated class names by a multiset.) The three forms of method binding are illustrated in Figure 7.14.

7.5.3 Example: Combining Authorization Policies

In a database containing sensitive information and different authorization policies, the information returned for a request depends on the clearance level of the agent making the request. Let *Agent* denote the interface of arbitrary agents and *Auth* an authorization interface with methods *grant*(x), *revoke*(x), and *auth*(x) for agents x. The two classes *SAuth* and *MAuth*, which both implement *Auth*, provide single and multiple authorization policies, respectively. *SAuth* authorizes one agent at a time and *MAuth* authorizes multiple agents. The method *grant*(x) returns when x becomes authorized, and authorization is removed by *revoke*(x). The method *auth*(x) suspends until x is authorized, and *delay* returns once no agent is authorized.

```
class SAuth implements Auth
begin
 var gr: Agent = null
 op delay == await (gr = null)
 op grant(in x:Agent)== delay(); gr:=x
 op auth(in x:Agent)== await (gr=x)
 op revoke(in x:Agent) ==
  if gr = x then gr := null else skip fi
with Agent
 op grant == grant(caller)
 op revoke == revoke(caller)
 op auth == auth(caller)
end
```

```
class MAuth implements Auth
begin
 var gr: Set[Agent] = ∅
 op delay == await (gr = ∅)
 op grant(in x:Agent) == gr := gr ∪ {x}
 op auth(in x:Agent) == await (x ∈ gr)
 op revoke(in x:Agent) == gr := gr \ {x}

with Agent
 op grant == grant(caller)
 op revoke == revoke(caller)
 op auth == auth(caller)
end
```

Authorization Levels

We now consider concurrent access to the database. *Low clearance* agents may share access to unclassified data while *high clearance* agents have unique access to (classified) data. Proper usage is defined by two interfaces, defining open and close operations at both access levels:

```
interface High
begin
with Agent
 op openH(out ok:Bool)
 op access(in k:Key out y:Data)
 op closeH
end
```

```
interface Low
begin
with Agent
 op openL
 op access(in k:Key out y:Data)
 op closeL
end
```

Not all agents are entitled to high authorization, so *openH* returns a Boolean.

Let a class *DB* provide the actual operations on the database. We assume given the following internal operations: *access*(**in** k:Key, level:Bool **out** y:Data), where *level* defines the access level (high or low), and *clear*(**in** x:*Agent* **out** b:Bool) to give clearance to sensitive data for agent *x*. Any agent may get low access rights, while only agents cleared by the database may be granted exclusive high access. Consequently, the *MAuth* class authorizes low clearance and *SAuth* authorizes high clearance. Since the attribute *gr* in *SAuth* is implemented as an object identifier, only one agent is authorized full access at a time.

```
class HAuth implements High
 inherits SAuth, DB
begin
 op access(in x:Agent;k:Key out y:Data)
 == auth(x); await access@DB(k,high;y)
with Agent
 op openH(out ok:Bool) ==
      await clear(caller;ok);
      if ok then grant(caller) else skip fi
 op access(in k:Key out y:Data) ==
      access(caller,k; y)
 op closeH == revoke(caller)
end
```

```
class LAuth implements Low
 inherits MAuth, DB
begin
 op access(in x:Agent;k:Key out y:Data)
 == auth(x); await access@DB(k,low;y)
with Agent
 op openL == grant(caller)
 op access(in k:Key out y:Data) ==
      access(caller,k; y)
 op closeL == revoke(caller)
end
```

The code given here uses asynchronous calls whenever an internal deadlock would be possible. Thus, objects of the four classes above may respond to new requests even when used improperly, for instance when agent access is not initiated by open.

The database itself has no interface containing *access*, therefore all database access is through the *High* and *Low* interfaces. Notice also that objects of the *HAuth* and *LAuth* classes may not be used through the *Auth* interface. This would have been harmful for the authorization provided in the example. For instance, an external call to the *grant* method of a *HAuth* object could result in high *access* without clearance of the calling agent! This supports the approach not to inherit implementation clauses.

Combining Authorization Levels

High and low authorization policies may be combined in a subclass *HLAuth* which implements both interfaces, inheriting *LAuth* and *HAuth*.

```
class HLAuth implements High, Low
 inherits LAuth, HAuth
begin
with Agent
 op access(in k:Key out y:Data) == if caller=gr@SAuth
  then access@HAuth(caller,k; y) else access@LAuth(caller,k; y) fi
end
```

Although the *DB* class is inherited twice, for both *High* and *Low* interaction, *HLAuth* gets only one copy (see Section 7.5.1).

The example demonstrates natural usage of classes and multiple inheritance. Nevertheless, it reveals problems with the combination of inheritance and *statically ordered* virtual binding: Objects of the classes *LAuth* and *HAuth* work well, in the sense that agents opening access through the *Low* and *High* interfaces get the appropriate access, but the addition of the common subclass *HLAuth* is detrimental: When used through the *High* interface, this class allows multiple high access to data! Calls to the *High* operations of *HLAuth* trigger calls to the *HAuth* methods. From these methods the virtual internal calls to *grant*, *revoke*, and *auth* now binds to those of the *MAuth* class, if selected in a left-first depth-first traversal of the inheritance tree of the actual class *HLAuth*. If the inheritance ordering in *HLAuth* were reversed, similar problems occur with the binding of *Low* interaction.

The *pruned* virtual binding strategy ensures that the virtual internal calls constrained by classes *HAuth* and *LAuth* are bound in classes *SAuth* and *MAuth*, respectively, regardless of the actual class of the caller (*HAuth*, *LAuth*, or *HLAuth*), and of the inheritance ordering in *HLAuth*. In an object of class *HLAuth*, the local calls to *grant*, *revoke*, and *auth* in code from class *HAuth* is understood as *grant<Sauth*, *revoke<Sauth*, and *auth<Sauth*. These may not be bound in the *Mauth* class since *Mauth* is not a subclass of *Sauth*.

7.5.4 Typing

In order to extend the type system to classes with inheritance we revise the rules for classes, internal calls, and replies, and add rules for the new method notations $m@C$ and $m < C$. These

$$\text{(CLASS-INH)}\quad \frac{\begin{array}{c} \Gamma \vdash_{\mathrm{V}} \mathit{Param} \,\langle\, \Delta \,\rangle \qquad \Gamma + \Delta \vdash_{\mathrm{V}} \mathit{InhAttr}(\mathit{Inh}, \Gamma_C); \mathit{Var} \,\langle\, \Delta' \,\rangle \\ \mathit{matchparam}\,(\Gamma + \Delta, \mathit{Inh}) \qquad \forall m \in \mathit{Mtd} \cdot \Gamma + \Delta + \Delta' \vdash m \,\langle\, \Delta^m \,\rangle \\ \forall I \in (\mathit{Impl}; \mathit{Contract}) \cdot \forall m' \in \Gamma_I(I).\mathit{Mtd} \cdot \exists\, C' \in \Gamma_C \cdot \\ \mathit{match}(m'.\mathit{Name}, \mathit{sig}(m'), m'.\mathit{Co}, \Gamma_C(C').\mathit{Mtd}) \wedge \Gamma_{\mathrm{V}}(\mathit{self}) \sqsubseteq C' \end{array}}{\Gamma \vdash \mathit{class}\,(\mathit{Param}, \mathit{Impl}, \mathit{Contract}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd}) \,\langle\, \bigcup_{m \in \mathit{Mtd}} \Delta^m_{\mathit{Sig}} \,\rangle}$$

$$\text{(INT-SYNC)}\quad \frac{\begin{array}{c} \Gamma \vdash_{\mathrm{F}} \textsc{e} : T \qquad \Gamma_{\mathrm{V}}(\mathit{self}) \sqsubseteq C \\ \exists\, C' \in \Gamma_C \cdot \mathit{match}(m, T \rightarrow \Gamma_{\mathrm{V}}(\textsc{v}), \varsigma, \Gamma_C(C').\mathit{Mtd}) \wedge C \sqsubseteq C' \end{array}}{\Gamma \vdash_{\mathrm{S}} m@C(\textsc{e}; \textsc{v})}$$

$$\text{(INT-ASYNC)}\quad \frac{\begin{array}{c} \Gamma \vdash_{\mathrm{F}} \textsc{e} : T \qquad \Gamma_{\mathrm{V}}(\mathit{self}) \sqsubseteq C \\ \exists\, C' \in \Gamma_C \cdot \mathit{match}(m, T \rightarrow \mathsf{Data}, \varsigma, \Gamma_C(C').\mathit{Mtd}) \wedge C \sqsubseteq C' \end{array}}{\Gamma \vdash_{\mathrm{S}} !m@C(\textsc{e})}$$

$$\text{(INT-ASYNC-L)}\quad \frac{\begin{array}{c} \Gamma \vdash_{\mathrm{F}} \textsc{e} : T \qquad \Gamma_{\mathrm{V}}(t) = \mathsf{Label} \qquad \Gamma_{\mathrm{V}}(\mathit{self}) \sqsubseteq C \\ \exists\, C' \in \Gamma_C \cdot \mathit{match}(m, T \rightarrow \mathsf{Data}, \varsigma, \Gamma_C(C').\mathit{Mtd}) \ \wedge\ C \sqsubseteq C' \end{array}}{\Gamma \vdash_{\mathrm{S}} t_i!m@C(\textsc{e}) \,\langle\, [t \overset{P}{\mapsto} \{i\}] + [t_i \overset{\mathit{Sig}}{\mapsto} \langle \varsigma, m_{C'}, \varsigma, T \rightarrow \mathsf{Data} \rangle] \,\rangle}$$

Figure 7.15: Typing of multiple inheritance. Here $\mathit{sig}(\mathrm{m})$ returns the signature of m.

are given in Figures 7.15 and 7.16. Let $\sqsubseteq$ be the reflexive and transitive closure of the subclass relation; $C \sqsubseteq C'$ expresses that C is a direct or indirect subclass of C', or is the same class as C'.

Definition 13 Let Γ be a typing environment, C be a class name, $\textsc{e}$ a list of expressions, and $\textsc{c}$ a list of instantiated class names. Define

$$\begin{array}{lcl} \mathit{matchparam}(\Gamma, \varepsilon) & = & \mathsf{true} \\ \mathit{matchparam}(\Gamma, C(\textsc{e}); \textsc{c}) & = & \Gamma \vdash_{\mathrm{F}} \textsc{e} : T \ \wedge\ T \preceq \mathit{type}(\Gamma_C(C).\mathit{Param}) \ \wedge\ \mathit{matchparam}(\Gamma, \textsc{c}). \end{array}$$

For a class C, the formal parameters of C may be instantiated with values passed to C from its subclasses. Thus, to ensure type-correct instantiations of superclasses, the type of the actual parameters of the instantiated superclass names must be type checked with respect to the type of the formal parameters of the superclasses. This is done by the auxiliary function *matchparam* in the typing rule CLASS-INH, which takes the typing environment and a list of instantiated class names, and compares the actual and formal parameters.

The initial expressions in variable declarations and the program statements in method bodies of a class C may refer to variables declared in its superclasses. Consequently, the typing environment Γ must be extended with inherited attributes before type checking variables and methods of class C. This extension is obtained by traversing the instantiated class names *Inh* of C depth first and using the mapping Γ_C to gather *Param* and *Var* from each superclass. The function *InhAttr* in the typing rule for classes returns the list of all typed variables inherited from the classes above C. (It follows that Lemma 2 holds for the extended language.) Furthermore, for each interface that C implements C must provide at least one type-correct method body for each method in the interface, either by inheritance or by local declaration.

Method calls. The typing of external calls is controlled by interfaces and is not affected by class inheritance. The rules for internal invocations ressemble those in Section 7.3.4, but

$$(\text{BND-SYNC})\quad \frac{\begin{array}{c}\Gamma \vdash_{\text{F}} \text{E} : T \\ \exists C' \in \Gamma_C \cdot (\Gamma_{\text{V}}(\mathit{self}) \sqsubseteq C' \sqsubseteq C) \\ \wedge\ \mathit{match}(m, T \to \Gamma_{\text{V}}(\text{V}), \varsigma, \Gamma_C(C').\mathit{Mtd})\end{array}}{\Gamma \vdash_{\text{S}}\ m < C(\text{E}; \text{V})}$$

$$(\text{BND-ASYNC})\quad \frac{\begin{array}{c}\Gamma \vdash_{\text{F}} \text{E} : T \\ \exists C' \in \Gamma_C \cdot (\Gamma_{\text{V}}(\mathit{self}) \sqsubseteq C' \sqsubseteq C) \\ \wedge\ \mathit{match}(m, T \to \mathsf{Data}, \varsigma, \Gamma_C(C').\mathit{Mtd})\end{array}}{\Gamma \vdash_{\text{S}}\ !m < C(\text{E})}$$

$$(\text{BND-ASYNC-L})\quad \frac{\begin{array}{c}\Gamma \vdash_{\text{F}} \text{E} : T \qquad \Gamma_{\text{V}}(t) = \mathsf{Label} \\ \exists C' \in \Gamma_C \cdot (\Gamma_{\text{V}}(\mathit{self}) \sqsubseteq C' \sqsubseteq C) \\ \wedge\ \mathit{match}(m, T \to \mathsf{Data}, \varsigma, \Gamma_C(C').\mathit{Mtd})\end{array}}{\Gamma \vdash_{\text{S}}\ t_i!m < C(\text{E})\ \langle\ [t \overset{P}{\mapsto} \{i\}] + [t_i \overset{Sig}{\mapsto} \langle \varsigma, m_{C'}, \varsigma, T \to \mathsf{Data}\rangle]\ \rangle}$$

$$(\text{REPLY})\quad \frac{\begin{array}{c}\Gamma_P(t) \neq \emptyset \qquad \Gamma_P(t) \neq \bot \\ \forall i \in \Gamma_P(t) \cdot \Gamma_{Sig}(t_i) = \langle I, m_s, Co, T_1 \to T_2\rangle \wedge (T_2 \sqcap \Gamma_{\text{V}}(\text{V})) \neq \bot \\ \wedge\ (I = \varsigma \wedge s \neq \_) \Rightarrow \mathit{match}(m, T_1 \to (T_2 \sqcap \Gamma_{\text{V}}(\text{V})), Co, \Gamma_C(s).\mathit{Mtd}) \\ \wedge\ (I \neq \varsigma \wedge s = \_) \Rightarrow \mathit{match}(m, T_1 \to (T_2 \sqcap \Gamma_{\text{V}}(\text{V})), Co, \Gamma_I(I).\mathit{Mtd})\end{array}}{\Gamma \vdash_{\text{S}}\ t?(\text{V})\ \langle\ [t \overset{P}{\mapsto} \emptyset] + \bigcup_{i \in \Gamma_P(t)} [t_i \overset{Sig}{\mapsto} \langle I, m_s, Co, T_1 \to (T_2 \sqcap \Gamma_{\text{V}}(\text{V}))\rangle]\ \rangle}$$

Figure 7.16: Typing of bounded method calls. The subscript "_" in the REPLY rule denotes an empty class subscript, representing an external invocation.

the analysis may now depend on the inheritance tree above *self*. The typing of internal calls inspects the inheritance graph, choosing a class such that the invocation is covered. As before, the signature of a call may be refined by a reply statement, but the signature is fixed to a class which need not be the class of *self*. For static calls $m@C$ the match starts from C, and not from the class of *self*. For bounded calls $m < C$, the match must be found below C in the inheritance tree above the class of *self*.

7.5.5 Operational Semantics

The operational semantics is adapted to incorporate multiple inheritance in Figure 7.17. Creol classes are extended to include the instantiated class names of inherited classes and are given as RL objects $\langle \mathit{Cl} \,|\, \mathit{Par}, \mathit{Inh}, \mathit{Att}, \mathit{Mtds}, \mathit{Tok}\rangle$, where *Cl* is the class name, *Par* a list of parameters, *Inh* is a list of instantiated class names, *Att* a list of attributes, *Mtds* a set of methods, and *Tok* is an arbitrary term of sort Label. When an object needs a method, it is bound to a definition in the *Mtds* set of its class or of a superclass. Previous class definitions (without inheritance) can be extended with an empty list of instantiated class names to be valid in the extended semantics.

Virtual and Static Binding of Method Calls

Qualified external method invocations are syntactically excluded; external invocations cannot access the internal structure of the callee. Internal calls give rise to invocation messages, but in R10$'$ the qualified method name *mq* may be of the form $m@C$ or $m < C$, where the constraint

$$\begin{aligned}
&\mathit{new}\, C(\mathrm{E})\ \langle C : \mathit{Cl} \mid \mathit{Tok} : n\rangle\\
&\longrightarrow\ \langle (C;n) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle \varepsilon, \varepsilon\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 1\rangle\\
&\qquad \langle C : \mathit{Cl} \mid \mathit{Tok} : \mathit{next}(n)\rangle\\
&\qquad \mathit{inherit}((C;n), \mathit{self} : \mathit{Any} = (C;n), \varepsilon)\ \mathbf{to}\ C(\mathit{eval}(\mathrm{E}, (\mathrm{A};\mathrm{L})))
\end{aligned}\tag{R2$'$}$$

$$\begin{aligned}
&\langle o : \mathit{Ob} \mid \mathit{Att} : \mathrm{A}, \mathit{Pr} : \langle (v := \mathit{new}\, C(\mathrm{E}); \mathrm{S}), \mathrm{L}\rangle\rangle\ \langle C : \mathit{Cl} \mid \mathit{Tok} : n\rangle\\
&\longrightarrow\ \langle o : \mathit{Ob} \mid \mathit{Att} : \mathrm{A}, \mathit{Pr} : \langle (v := (C;n); \mathrm{S}), \mathrm{L}\rangle\rangle\ \langle C : \mathit{Cl} \mid \mathit{Tok} : \mathit{next}(n)\rangle\\
&\qquad \langle (C;n) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \varepsilon, \mathit{Pr} : \langle \varepsilon, \varepsilon\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 1\rangle\\
&\qquad \mathit{inherit}((C;n), \mathit{self} : \mathit{Any} = (C;n), \varepsilon)\ \mathbf{to}\ C(\mathit{eval}(\mathrm{E}, (\mathrm{A};\mathrm{L})))
\end{aligned}\tag{R3$'$}$$

$$\begin{aligned}
&\langle o : \mathit{Ob} \mid \mathit{Att} : \mathrm{A}, \mathit{Pr} : \langle\, !\mathit{mq}(\mathit{Sig}, \mathit{Co}, \mathrm{E}); \mathrm{S}, \mathrm{L}\rangle, \mathit{Lab} : n\rangle\\
&\longrightarrow\ \langle o : \mathit{Ob} \mid \mathit{Att} : \mathrm{A}, \mathit{Pr} : \langle \mathrm{S}, \mathrm{L}\rangle, \mathit{Lab} : \mathit{next}(n)\rangle\\
&\qquad \mathit{invoc}(\mathit{mq}, \mathit{Sig}, \mathit{Co}, (n\ o\ \mathit{eval}(\mathrm{E}, (\mathrm{A};\mathrm{L}))))\ \mathbf{to}\ o
\end{aligned}\tag{R10$'$}$$

$$\begin{aligned}
&\langle o : \mathit{Ob} \mid \mathit{EvQ} : \mathit{invoc}(m@C, \mathit{Sig}, \mathit{Co}, \mathrm{E})\ \mathrm{Q}\rangle\\
&\longrightarrow \langle o : \mathit{Ob} \mid \mathit{EvQ} : \mathrm{Q}\rangle\ (\mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ C)
\end{aligned}\tag{R24}$$

$$\begin{aligned}
&\langle o : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{EvQ} : \mathit{invoc}(m < C', \mathit{Sig}, \mathit{Co}, \mathrm{E})\ \mathrm{Q}\rangle\\
&\longrightarrow \langle o : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{EvQ} : \mathrm{Q}\rangle\ (\mathit{bind}(m < C', \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ C)
\end{aligned}\tag{R25}$$

$$\begin{aligned}
&(\mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ C\ \mathrm{I})\ \langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}\rangle\\
&\longrightarrow\ \mathbf{if}\ \mathit{match}(m, \mathit{Sig}, \mathit{Co}, \mathrm{M})\\
&\qquad \mathbf{then}\ \mathit{bound}(\mathit{get}(m, \mathrm{M}, \mathrm{E}))\ \mathbf{to}\ o\\
&\qquad \mathbf{else}\ \mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ (\mathrm{I}'\ \mathrm{I})\ \mathbf{fi}\ \langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}\rangle
\end{aligned}\tag{R16$'$}$$

$$\begin{aligned}
&(\mathit{bind}(m < C', \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ C\ \mathrm{I})\ \langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}, \mathit{Tok} : n\rangle\\
&\longrightarrow\ \langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}, \mathit{Tok} : \mathit{next}(n)\rangle\ (\mathbf{if}\ \mathit{match}(m, \mathit{Sig}, \mathit{Co}, \mathrm{M})\\
&\qquad \mathbf{then}\ (\mathit{find}(n, C', C)\ \mathbf{to}\ C)\ (\mathit{stopbind}(n, m < C', \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ C\ \mathrm{I})\\
&\qquad \mathbf{else}\ \mathit{bind}(m < C', \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ (\mathrm{I}'\ \mathrm{I})\ \mathbf{fi})
\end{aligned}\tag{R26}$$

$$\begin{aligned}
&(\mathit{found}(n, b, C')\ \mathbf{to}\ C)\ (\mathit{stopbind}(n, m, \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ C\ \mathrm{I})\\
&\langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}\rangle\\
&\longrightarrow\ \mathbf{if}\ b\ \mathbf{then}\ \mathit{bound}(\mathit{get}(m, \mathrm{M}, \mathrm{E}))\ \mathbf{to}\ o\ \mathbf{else}\ \mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ \mathrm{I}\ \mathbf{fi}\\
&\qquad \langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}\rangle
\end{aligned}\tag{R27}$$

$$\mathit{find}(n, C, C'')\ \mathbf{to}\ \varepsilon\ \longrightarrow\ \mathit{found}(n, \mathit{false}, C)\ \mathbf{to}\ C''\tag{R28}$$

$$\mathit{find}(n, C, C'')\ \mathbf{to}\ \mathrm{I}\ C\ \mathrm{I}'\ \longrightarrow\ \mathit{found}(n, \mathit{true}, C)\ \mathbf{to}\ C''\tag{R29}$$

$$\begin{aligned}
&(\mathit{find}(n, C, C'')\ \mathbf{to}\ C'\ \mathrm{I})\ \langle C' : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}'\rangle\\
&\longrightarrow\ (\mathit{find}(n, C, C'')\ \mathbf{to}\ \mathrm{I}\ \mathrm{I}')\ \langle C' : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}'\rangle\ \mathbf{if}\ (C \neq C')
\end{aligned}\tag{R30}$$

Figure 7.17: An operational semantics with multiple inheritance. Note that the rules R2$'$, R3$'$, R10$'$, and R16$'$ redefine the previous rules R2, R3, R10, and R16. In R10$'$, mq denotes either $m@C$ or $m < C$.

C is used in the binding.

In order to allow concurrent and dynamic execution, the inheritance graph is not statically given. Rather, the binding mechanism dynamically inspects the class hierarchy in the configuration. Our approach to virtual binding uses a *bind* message, which is sent from a class to its superclasses, resulting in a *bound* message returned to the object requesting the method binding. This way, the inheritance graph is explored dynamically and only as far as necessary. When the external invocation of a method m is found in the message queue of an object o, a message $\mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)$ **to** C is sent in R12, after retrieving the class C of the object. For internal static calls $m@C$, the *bind* message is sent by R24 without inspecting the *actual* class of the callee, thus surpassing local definitions. If a suitable m is defined locally in C, a process with

$$\mathit{inherit}(o, \text{S}, \text{A})\ \textbf{to}\ \mathit{nil} = \mathit{inherited}(\text{S}; \text{A})\ \textbf{to}\ o$$

$$\mathit{inherit}(o, \text{S}, \text{A})\ \textbf{to}\ (\text{I}\ C(\mathit{In}))\ \ \langle C : \mathit{Cl} \,|\, \mathit{Par} : (\text{V} : T), \mathit{Inh} : \text{I}', \mathit{Att} : \text{A}' \rangle$$
$$= \mathit{inherit}(o, (\text{S}; \text{V} : T = \mathit{In}), \text{A}'; \text{A}))\ \textbf{to}\ (\text{I}\ \text{I}')\ \langle C : \mathit{Cl} \,|\, \mathit{Par} : (\text{V} : T), \mathit{Inh} : \text{I}', \mathit{Att} : \text{A}' \rangle$$

$$(\mathit{inherited}(\text{A})\ \textbf{to}\ o)\ \langle o : \mathit{Ob} \,|\, \mathit{Pr} : \langle \varepsilon, \varepsilon \rangle \rangle = \langle o : \mathit{Ob} \,|\, \mathit{Pr} : \langle (\text{A}) \downarrow; \mathit{run}, \varepsilon \rangle \rangle$$

Figure 7.18: State instantiation equations for multiple inheritance.

the method code and local state is returned in a *bound* message. Otherwise, the *bind* message is *retransmitted* to the superclasses of C in a left-first depth-first order by application of R16$'$. In order to facilitate the traversal of the inheritance graph, a list of instantiated class names is used as the destination of the *bind* message. The process resulting from the binding is loaded into the internal process queue of the callee as before.

Pruned Virtual Binding

The binding of an internal virtual call $m < C'$ is more involved. When a match in a class C is found in the application of R26, the inheritance graph of C is inspected to ensure that $C \sqsubseteq C'$, otherwise the binding must resume. Note the additional *stopbind* message with token n, which suspends binding while checking that $C \sqsubseteq C'$. This is done by two auxiliary messages, captured in R27 – R30: The message $\mathit{find}(n, C', C)$ **to** I represents that C is asking a list I of instantiated class names (ignoring the actual parameter values for readability) if C' may be found in I or further up in the hierarchy. The corresponding message $\mathit{found}(n, b, C')$ **to** C returns an answer to C. In this message the Boolean b is true if the request was successful and n matches the token of the *stopbind* message, identifying the call being bound. This search corresponds to left-first breadth-first traversal of the inheritance graph.

Object Creation and Attribute Instantiation

The object creation rules R2 and R3 are redefined to address the dynamic inheritance graph (see R2$'$ and R3$'$). In order to initialize the state of the new object the inheritance graph is traversed and inherited state variables collected, using the equations given in Figure 7.18. Recall that using equations enables object creation and attribute collection in one rewrite step. The equations convert class parameters and actual parameter values to attribute declarations which textually precede the attribute list of each class. Inherited parameters and attribute lists textually precede the attribute list of a subclass. The collected attribute declarations are evaluated in the new object in order to initialize the object state. Class parameters and inherited attributes provide a mechanism to pass values to the initial expressions of the inheritance list in a class. The order in which parameters are collected (Figure 7.18) ensures that multi-inheritance of the same class is the same as inheriting the class once, keeping the leftmost instantiation of the state variables.

7.5.6 Type soundness

The soundness of the full language is established in this section. We first show that the class hierarchy is correctly unfolded when initializing the object state.

Lemma 15 *If inherit(o, S, A) **to** I is a message in a configuration of a well-typed execution of a program P, then* $\Gamma_{\mathcal{F}} \vdash_{\text{V}} \text{S} \langle \Delta \rangle$.

Proof. Let $\text{S} = \text{V}_0 : T_0 = \text{E}_0; \dots; \text{V}_n : T_n = \text{E}_n$ be an attribute list and let Γ^i_{V} be a mapping such that $\Gamma^i_{\text{V}}(\text{V}_j) = T_j$ for $1 \leq j < i$. We show that $\Gamma_{\mathcal{F}} + \Gamma^n_{\text{V}} \vdash_{\text{F}} \text{E}_n \langle \Delta \rangle$. The proof is by induction over n. For $n = 0$, $\text{S} = \mathit{self}\colon \mathit{Any} = (C;n)$, $\Gamma_{\mathcal{F}}(\mathit{self}) = \mathit{Any}$, and $\Gamma_{\mathcal{F}}((C;n)) = I$, so $I \preceq \mathit{Any}$ and $\Gamma_{\mathcal{F}} \vdash_{\text{V}} \text{S} \langle \Delta \rangle$. For $n = 1$, the message must have been caused by application of R2$'$ or R3$'$. By Lemma 2, the evaluation of an expression in a well-typed method successfully dereferences all program variables in the expression. Since E_1 is an expression which has been evaluated either in R2$'$ or R3$'$, E_1 does not refer to program variables and consequently $\Gamma_{\mathcal{F}} \vdash_{\text{F}} \text{E}_1$. For the induction step we assume that the lemma holds for $n+1$ and show that it also holds for $n+2$, in which case V_{n+2} is a class parameter to some class C' which is inherited by a class C such that V_i $(i \leq n+1)$ is the class parameter of C. Since C is well-typed, CLASS-INH asserts that $\Gamma_{\mathcal{F}} + \Gamma^i_{\text{V}} \vdash_{\text{F}} \text{E}_{n+2}$ through the *matchparam* predicate. It follows that $\Gamma_{\mathcal{F}} \vdash_{\text{V}} \text{S} \langle \Delta \rangle$. ■

Lemma 16 *Given an arbitrary Creol program P and a well-typed execution* ρ *of P. The execution of a statement* $x := \textbf{new}\ C(\text{E})$ *in the final configuration of* ρ *results in well-typed configurations of P while the new object is initialized.*

Proof. Let o' be a runtime object executing $x := \textbf{new}\ C(\text{E})$ (by the application of R3$'$) in the final configuration ρ_i of ρ. Let o be the new object reference. The well-typedness of o' before and after the execution of $x := \textbf{new}\ C(\text{E})$ and the uniqueness of reference o are covered in the proof of Lemma 7. The configuration ρ_{i+1} includes a runtime object

$$\langle o\colon Ob \,|\, Cl\colon C, Att\colon \varepsilon, Pr\colon \langle ((\text{A}_0)\downarrow; run), \varepsilon \rangle, PrQ\colon \varepsilon, EvQ\colon \varepsilon, Lab\colon 1 \rangle$$

which is well-typed. Let E' be the result of evaluating E in o'. We need to show that the object remains well-typed while the assignment list $(\text{A}_0)\downarrow$ is executed and that the assignment list instantiates all program variables in the object state. This is done by showing that the attribute list A_0 is well-typed, so that object instantiation (by the repeated application of R1) results in well-typed configurations. The proof is by induction over the depth of the inheritance tree above C. At each step of the induction we assume that the parameter and attribute lists are type-correctly collected from the inheritance tree above the currently considered class, and show that when type-checking the parameter and attribute list from left to right, all variables occurring in the initial expressions have been instantiated.

For the basis step, let C' be a leaf above C with formal class parameters V of type T' and let *In* be the actual parameter values passed to C'. For well-typed programs, we know that $\Gamma \vdash_{\text{F}} In : T$ such that $T \prec T'$ (this is checked by the *matchparam* function). Consequently, *In* can be type-correctly assigned to V. Let S and A be accumulated class parameter and class attribute assignments, and I be a list of superclasses of C. (If we are instantiating a leaf class directly, S, A, and I are empty.) The operational semantics gives us

$(\mathit{inherit}(o, \text{S}, \text{A})\ \textbf{to}\ \text{I}\ C'(In))\ \langle C' \colon Cl \,|\, Par\colon (\text{V} : T'), Inh\colon nil, Att\colon \text{A}' \rangle$
$= (\mathit{inherit}(o, (\text{S}; \text{V} : T' = In), (\text{A}'; \text{A}))\ \ \textbf{to}\ \text{I})\ \langle C' \colon Cl \,|\, Par\colon (\text{V} : T'), Inh\colon nil, Att\colon \text{A}' \rangle.$

By Lemma 15, $\Gamma \vdash_{\mathrm{V}} \textsc{s};\textsc{v}:T' = In \langle \Delta \rangle$. We need to show that $\Gamma \vdash_{\mathrm{V}} \textsc{s};\textsc{v}:T' = In;\textsc{a}'$. Since C' is a leaf the function $InhAttr(Inh,\Gamma_C) = \varepsilon$ and, by the typing rule CLASS-INH, $\Gamma + \Delta \vdash_{\mathrm{V}} \textsc{a}'$. It follows that $\Gamma \vdash_{\mathrm{V}} \textsc{s};\textsc{v}:T' = In;\textsc{a}'$. It is immediate that $\textsc{s};\textsc{v}:T' = In;\textsc{a}'$ contains all state variables declared in C'. Note that the attribute assignments of C' precede the accumulated attribute assignment list $\textsc{a}$, so initial expressions in $\textsc{a}$ may safely refer to variables in $\textsc{s};\textsc{v}:T' = In;\textsc{a}'$.

For the induction step, we consider a class C' and assume that the assignment list from each of its superclasses is well-typed (with respect to the accumulated parameter assignment list) and contains all state variables declared in the superclasses of C'. Due to the qualified name convention, the concatenation of these assignments lists is also well-typed with respect to the accumulated parameter assignment list. The following equation applies:

$$\begin{aligned} &inherit(o,\textsc{s},\textsc{a})\ \mathbf{to}\ (\textsc{i}\ C'(In))\ \ \langle C' : Cl \,|\, Par : (\textsc{v}:T), Inh : \textsc{i}', Att : \textsc{a}'\rangle \\ &= inherit(o,(\textsc{s};\textsc{v}:T = In),\textsc{a}';\textsc{a}))\ \mathbf{to}\ (\textsc{i}\ \textsc{i}')\ \langle C' : Cl \,|\, Par : (\textsc{v}:T), Inh : \textsc{i}', Att : \textsc{a}'\rangle. \end{aligned}$$

The message $inherit(o,(\textsc{s};\textsc{v}: T = In),\textsc{a}';\textsc{a}))$ **to** $(\textsc{i}\ \textsc{i}')$ is further reduced to $inherit(o,(\textsc{s};\textsc{v}: T = In;\textsc{s}'),\textsc{a}'';\textsc{a}';\textsc{a}))$ **to** $\textsc{i}$, where the superclasses of C' have been expanded. Here, $\textsc{s}'$ and $\textsc{a}''$ are the accumulated parameter and attribute assignment lists from the classes in $\textsc{i}'$. By Lemma 15 and the induction hypothesis, $\Gamma \vdash_{\mathrm{V}} \textsc{s};\textsc{v}: T = In;\textsc{s}';\textsc{a}'' \langle \Delta \rangle$. We need to show that $\Gamma \vdash_{\mathrm{V}} \textsc{s};\textsc{v}: T = In;\textsc{s}';\textsc{a}'';\textsc{a}'$. By typing rule CLASS-INH, $\Gamma + \Delta \vdash_{\mathrm{V}} \textsc{a}'$, since Δ contains all class parameters and inherited attributes. It follows that $\Gamma \vdash_{\mathrm{V}} \textsc{s};\textsc{v}: T = In;\textsc{s}';\textsc{a}'';\textsc{a}'$ and $\textsc{s};\textsc{v}: T = In;\textsc{s}';\textsc{a}'';\textsc{a}'$ contains all state variables declared above C'.

Finally, if C' is the actual class C of the object, the actual class parameters passed to C' come from the statement **new** $C(\textsc{e})$. Lemma 15 asserts that $\textsc{e}$ can be type-correctly assigned to the formal parameters of C', $\textsc{s} = self : Any = (C;n)$, and $\textsc{a} = \varepsilon$. It follows that the attribute list $self : Any = (C;n);\textsc{v}: T = \textsc{e};\textsc{s}';\textsc{a}'';\textsc{a}'$ is well-typed and declares all state variables of C'. Since the attribute list is well-typed, every application of R1 to $(\textsc{a}_0)\downarrow$ will result in a well-typed object. Consequently the evaluation of $(\textsc{a}_0)\downarrow$ constructs a well-typed state σ containing all object variables, and the object reduces to

$$\langle o : Ob \,|\, Cl : C, Att : \sigma, Pr : \langle run, \varepsilon\rangle, PrQ : \varepsilon, EvQ : \varepsilon, Lab : 1\rangle.$$

Object initialization preserves well-typedness. ■

Lemma 17 *Let P be an arbitrary Creol program. If $\Gamma_{\mathcal{F}} \vdash P$, then every method invocation $!x.m(T_{in} \to T_{out}, Co, \textsc{e})$, $!m@C(T_{in} \to T_{out}, Co, \textsc{e})$, or $!m < C(T_{in} \to T_{out}, Co, \textsc{e})$ in a well-typed configuration of P can be type-correctly bound at runtime to a method such that the return values from the method are of type T'_{out} and $T'_{out} \preceq T_{out}$, provided that x is not a null pointer.*

Proof. We consider how the invocation message is bound in the evaluation rules for $!x.m(Sig,Co,\textsc{e})$, $!m@C(Sig,Co,\textsc{e})$, and $!m < C(Sig,Co,\textsc{e})$. The proof is similar to the proof of Lemma 8, but accounts for the dynamic traversal of the inheritance graph. As the call to m is well-typed, we may assume that x is typed by an interface I and that x is an instance of a class C which implements I. Consequently, there is at least a class C' above C in which m is declared with an appropriate signature and cointerface. Let $\textsc{e}$ evaluate to $\textsc{e}'$ such that $\Gamma(\textsc{e}') \preceq \Gamma(\textsc{e})$. Applying R9 to an external method call $!x.m(Sig, Co, \text{E})$ creates a message $bind(m, Sig, Co, \text{E}', x)$ **to** C, if C is the dynamically identified class of x. Method lookup proceeds by R16′:

$$\begin{aligned}
&(bind(m, Sig, Co, \text{E}', x)\ \textbf{to}\ C\ \text{I}')\langle C : Cl \mid Inh : \text{I}, Mtds : \text{M}\rangle \\
&\longrightarrow \textbf{if}\ match(m, Sig, Co, \text{M})\ \textbf{then}\ bound(get(m, \text{M}, \text{E}'))\ \textbf{to}\ x \\
&\qquad\qquad\qquad\qquad \textbf{else}\ bind(m, Sig, Co, \text{E}', x)\ \textbf{to}\ (\text{I}\ \text{I}')\ \textbf{fi}\ \langle C : Cl \mid Inh : \text{I}, Mtds : \text{M}\rangle
\end{aligned}$$

where I' is initially empty. The method lookup function may potentially traverse the entire inheritance tree above C, including C'. Method binding is guaranteed to succeed at C'.

An internal method call $!m@C(Sig, Co, \text{E})$ in an object o resembles the previous case. By R10′ the call results in a message $invoc(m@C, Sig, Co, \text{E}')$ **to** o which, by applying R24, generates a message $bind(m, Sig, Co, \text{E}', o)$ **to** C where C is the specified class. The call is correctly bound if there is a method m declared above C with signature $Sig' = T'_{in} \rightarrow T'_{out}$ and cointerface Co' such that $Sig' \preceq Sig$ and $Co \preceq Co'$, so $T'_{out} \preceq T_{out}$. The type analysis guarantees that C inherits a method m with a matching signature and cointerface. Consequently, the *bind* message succeeds in binding the call to a method declared above C when traversing the inheritance tree above C, by repeated applications of R16′.

Let o be an object of class C'. An internal bounded call $m < C(Sig, Co, \text{E})$ in o results in the message $invoc(m < C, Sig, Co, \text{E}')$ **to** o, which by R25 generates a message $bind(m < C, Sig, Co, \text{E}', o)$ **to** C'. The (repeated) application of R26 inspects the inheritance graph above C', searching for a matching method declaration located below C. For every match in the inheritance graph, say in a class C'', a message $find(n, C, C'')$ **to** C'' is generated, which means that the class C'' is a candidate for binding m. By application of rules R28–R30 this message returns true with token n if C is above C'' and false otherwise, and the token identifies the corresponding *stopbind* message. If the result is false, rule R27 continues the search. Well-typedness guarantees that there is at least one match in a class below C with signature $T'_{in} \rightarrow T'_{out}$ and Co' such that $T'_{out} \preceq T_{out}$, so the search eventually succeeds. ■

Theorem 18 (Type soundness) *All executions of Creol programs starting in a well-typed initial configuration, are well-typed.*

Proof. We consider a well-typed execution ρ of a program P. The proof is by the induction over the length of $\rho = \rho_0, \rho_1, \ldots$ and extends the proofs of Theorems 10 and 14. By assumption, ρ_0 is a well-typed initial configuration of P. As only R2′ is applicable to ρ_0, it follows from Lemma 16 that object creation results in a well-typed successor configuration. For the induction step we show that for any well-typed configuration ρ_i, the successor configuration ρ_{i+1} is also well-typed, by case analysis of the rewrite rules. Only the new rules of Figure 7.17 are discussed, the other rules are covered by the proofs of Theorems 10 and 14. We first consider object reductions; i.e., rules that reduce an object $\langle o : Ob \mid Att : \text{A}, Pr : \langle s; \text{S}, \text{L}\rangle\rangle$ to $\langle o : Ob \mid Att : \text{A'}, Pr : \langle s'; \text{S}, \text{L}\rangle\rangle$, and then the remaining rewrite rules.

- Consider $s = v := \textbf{new}\ C(\text{E})$ and the application of R3′. By Lemma 16, the evaluation of an object creation statement gives a well-typed successor configuration ρ_{i+1}. Moreover, the successor configurations from the initialization of the new object are also well-typed.

- For $s = mq(Sig, Co, \mathrm{E})$ and the application of R10′, the state variables are not changed and ρ_{i+1} is well-typed.

- Applying R24 and R25 does not modify the state, so ρ_{i+1} is well-typed.

We now consider the remaining new rewrite rules.

- Rules R16′ and R26. Let I be a list of instantiated class names. As ρ_i is a configuration of the well-typed execution ρ, a message $bind(m, Sig, Co, \mathrm{E}, o)$ **to** I must have been generated from a message $invoc(m@C, Sig, Co, \mathrm{E})$ by application of R24 or from a message $invoc(m, Sig, Co, \mathrm{E})$ by application of R12 for external calls. Similarly, a message $bind(m < C', Sig, Co, \mathrm{E}, o)$ **to** I must have been generated from a message $invoc(m < C', Sig, Co, \mathrm{E})$ by application of R25. These *invoc* messages must have been generated by applying R10′ to a statement $!mq(Sig, Co, \mathrm{E})$ or R9 to a statement $!x.m(Sig, Co, \mathrm{E})$ for external calls. By Lemma 17, any such call $m@C(Sig, Co, \mathrm{E})$, $m < C'(Sig, Co, \mathrm{E})$, or $!x.m(Sig, Co, \mathrm{E})$ in P can be type-correctly bound at runtime. Consequently, the *match* function in R16′ and R26 will eventually succeed before $\mathrm{I} = \varepsilon$, resulting in a *bound* or a *find* message, and method-not-understood errors cannot occur in ρ_{i+1}. It follows that ρ_{i+1} is well-typed.

- Rule R27. As ρ is well-typed, the *found* and *stopbind* messages are originated from the application of R26 to a message $bind(m < C', Sig, Co, \mathrm{E}, o)$ **to** I, in a well-typed configuration. By Lemma 17, the invocation can be bound by traversing the inheritance tree above C and below C'. Consequently, the binding will succeed before $\mathrm{I} = \varepsilon$ generating a *bound* message. It follows that method-not-understood errors do not occur in ρ_{i+1}, so ρ_{i+1} is well-typed.

- Rules R28, R29, and R30 do not modify the state, so ρ_{i+1} is well-typed. ■

7.6 Related Work

Many object-oriented languages offer constructs for concurrency; a survey is given in [67]. A common approach is to rely on the tight synchronization of RPC, separating activity (threads) and objects, as done in Hybrid [65] and Java [37], or on the rendezvous concept in concurrent objects languages such as Ada and POOL [5]. These approaches seem less desirable for distributed systems, with potential delays and communication loss. Hybrid offers *delegation* to (temporarily) branch an activity thread. Asynchronous method calls may be seen as a form of delegation and can be implemented in, e.g., Java by explicitly creating new threads to handle calls [24]. In Creol, polling for replies to asynchronous calls is handled by the operational semantics: no new threads and active loops are needed to poll for replies to delegated activity. UML offers asynchronous event communication and synchronous method invocation but does not integrate these, resulting in significantly more complex formalizations [27] than ours. To facilitate the programmer's task and reduce the risk of errors, implicit control structures based on asynchronous method calls seem more attractive, allowing a higher level of abstraction in the language.

The internal concurrency model of concurrent objects in Creol may be compared to monitors [39] or to thread pools executing on a single processor, with a shared state space given by the object attributes. In contrast to monitors, explicit signaling is avoided. Sufficient signaling is ensured by the semantics, which significantly simplifies reasoning [25]. However, general monitors may be encoded in the language [48]. In contrast to thread pools, processor release is explicit. In Creol, the activation of suspended processes is nondeterministically handled by an unspecified scheduler. Consequently, intra-object concurrency is similar to the interleaving semantics of concurrent process languages [30, 6], where each Creol process resembles a series of guarded atomic actions (discarding local process variables). Internal reasoning control is facilitated by the explicit declaration of release points, at which class invariants should hold [32].

Languages based on the Actor model [4, 3] take asynchronous messages as the communication primitive, focussing on loosely coupled processes with less synchronization. This makes Actor languages conceptually attractive for distributed programming. The interpretation of method calls as asynchronous messages has lead to the notion of future variables which may be found in languages such as ABCL [82], Argus [54], ConcurrentSmalltalk [81], Eiffel// [17], CJava [24], and in the Join calculus [34] based languages Polyphonic C$^\sharp$ [9] and Join Java [44]. Our communication model is also based on asynchronous messages and the proposed asynchronous method calls resemble programming with future variables, but Creol's processor release points further extend this approach to asynchrony with additional flexibility.

Languages supporting asynchronous methods generally either disallow inheritance [82, 44] or impose redefinition of asynchronous methods [17]. Multiple inheritance is supported in languages such as C++ [74], CLOS [28], Eiffel [60], POOL [5], and Self [20]. Horizontal name conflicts in C++, POOL, and Eiffel are removed by explicit resolution, after which the inheritance graph may be linearized. A natural semantics for virtual binding in Eiffel is proposed in [7]. This work is similar in spirit to ours and models the binding mechanism at the abstraction level of the program, capturing Eiffel's renaming mechanism. Mixin-based inheritance [11] and traits [71, 66] depend upon linearization to be merged correctly into the single inheritance chain. Linearization changes the parent-child relationship between classes in the inheritance hierarchy [72], and understanding method binding quickly becomes difficult.

Maude's inherent object concept [58, 21] represents an object's state as a subconfiguration, as we have done here, but in contrast to our approach object behavior is captured directly by rewrite rules. Both Actor-style asynchronous messages and synchronous transitions (rewrite rules involving several objects) are allowed, which makes Maude's object model very flexible. However, asynchronous method calls and processor release points as proposed in this paper are hard to represent within this model. Both Maude and the Join-calculus capture multiple inheritance by this disjoint union of methods. Name ambiguity lets method definitions compete for selection. The definition selected when an ambiguously named method is called, is nondeterministically chosen. In Polyphonic C$^\sharp$ this nondeterminism is supplemented by a substitution mechanism for inherited code. CJava, restricted to outer guards and single inheritance, allows separate redefinition of synchronization code and bodies in subclasses. Programmer control may be improved if inherited classes are ordered [28, 20], resulting in deterministic binding. However, the ordering of superclasses may result in surprising but "correct" behavior. The example of Section 7.5.3 displays such surprising behavior regardless of how the inherited classes are ordered.

The statements for high-level control of local computation in Creol are inspired by notions from process algebra [63,40]. Process algebra is usually based on synchronous communication. In contrast to, e.g., the asynchronous π-calculus [41], which encodes asynchronous communication in a synchronous framework by dummy processes, our communication model is truly asynchronous and without channels: message overtaking may occur. Further, Creol differs from process algebra in its integration of processes in an object-oriented setting using methods, including active and passive object behavior, and self reference rather than channels. In formalisms based on process algebra the operation of returning a result is not directly supported, but typically encoded as sending a message on a return channel [69, 78, 79]. Finally, Creol's high-level integration of asynchronous and synchronous communication and the organization of pending processes and interleaving at release points within class objects seem hard to capture naturally in process algebra.

Formal models clarify the intricacies of object orientation and may thus contribute to better programming languages in the future, making programs easier to understand, maintain, and analyze. Object calculi such as the ς-calculus [1] and its concurrent extension [36] aim at a direct expression of object-oriented features such as self-reference, encapsulation, and method calls, but asynchronous invocation of methods is not addressed. This also applies to Obliq [16], a programming language based on similar primitives which targets distributed concurrent objects. The concurrent object calculus of Di Blasio and Fisher [29] provides both synchronous and asynchronous invocation of methods. In contrast to Creol, return values are discarded when methods are invoked asynchronously and the two ways of invoking a method have different semantics. Class inheritance is not addressed in [1, 36, 29].

In the concurrent object calculus with single inheritance studied by Laneve [53], methods of superclasses are accessible and virtual binding is addressed by a careful renaming discipline. A denotational semantics for single inheritance with similar features is studied by Cook and Palsberg [22]. Multiple inheritance is not addressed in these works. Formalizations of multiple inheritance are usually based on the *objects-as-records* paradigm and focusses on subtyping issues related to subclassing. Issues related to method binding are not easily captured in this approach: Even access to superclass' methods is not addressed in Cardelli's denotational semantics of multiple inheritance [15]. Rossi, Friedman, and Wand [70] propose a formalization of multiple inheritance based on *subobjects*, a runtime data structure used for virtual pointer tables [52, 74]. Their work focusses on compile time issues and does not clarify multiple inheritance at the abstraction level of the programming language.

The dynamically typed prototype-based language Self [20] proposes an elegant *prioritized binding strategy* to solve horizontal name conflicts, although a formal semantics is not given. The strategy is based on combining ordered and unordered multiple inheritance. Each superclass is annotated with a priority, and many superclasses may have the same priority. A name is only ambiguous if it occurs in two superclasses with the same priority, in which case a class related to the actual class is preferred. However, explicit class priorities may have surprising effects in large class hierarchies: names may become ambiguous through inheritance. If neither class is related to the caller the binding does not succeed, resulting in a method-not-understood error.

The *pruned binding strategy* proposed in this paper solves these issues without the need for manually declaring (equal) class priorities and without the possibility of method-not-understood errors: Calls are only bound to intended method redefinitions. This binding strategy seems particularly useful during system maintenance to avoid introducing unintentional errors in evolving class hierarchies, supported in Creol [50]. In particular, Creol's operational semantics is based on the dynamic and distributed traversal of the class hierarchy, rather than on virtual pointer tables. Our approach may therefore be combined with dynamic constructs for changing the class inheritance structure, such as adding a class C and enriching an existing class with C as a new superclass.

The type system presented in this paper resembles that of Featherweight Java [42], a core calculus for Java, because of its nominal approach. Featherweight Java is class based and uses a class table to represent class information in its type system. Subtyping is the reflexive and transitive closure of the subclass relation. In contrast the type system of Creol cleanly distinguishes classes and types, which results in both a class and an interface table. Furthermore, Featherweight Java does not address issues related to assignment, overloading, and interfaces. A subtype discipline is required for method overriding, which allows significantly simpler definitions of method lookup (virtual binding is trivial in this setting). Multiple inheritance, interfaces and cointerfaces, nondeterministic merge and choice, and asynchronous method calls are not found in (Featherweight) Java. PolyToil [14] separates subtyping and (single) inheritance. Object types resemble Creol's interfaces, but there is only one type per class and no notion of cointerface. Scala [66] uses a nominal type system for mixin-based traits, extending a single inheritance relation. Asynchronous method calls, interfaces, and cointerfaces in Creol necessitate a more refined type system, including an effect system [56]. A type and effect system provides an elegant way of adding context information to the type analysis [75]. Type and effect systems have been used to ensure that, e.g., guards controlling method availability do not have side effects [29] and to estimate the effects of a reclassification primitive [33]. For Creol, the effect system derives type-correct signatures for asynchronous method calls. The system may be extended to ensure the absence of null pointers, using initialization restrictions [31] and checks on remote calls guaranteeing that called objects are not null.

7.7 Conclusion

This paper has presented the Creol model of distributed concurrent objects communicating by means of asynchronous method calls. The approach emphasizes flexibility with respect to the possible delays and instabilities of distributed computing but also with respect to code reuse through a liberal notion of multiple inheritance. The model makes a clear distinction between inheritance and subtyping, in particular subtyping is not required for method redefinition. Object variables are typed by interface, abstracting from the actual class of external objects. An object may be typed by many interfaces, expressing different roles of the object. Interfaces may require cointerfaces, expressing dependencies which facilitate protocol sessions in the distributed environment. The concept of *contracts* is used to statically control the typing of mutually dependent classes in presence of inheritance. Creol is formalized with an operational semantics defined in rewriting logic, providing a detailed account of, e.g., asynchronous method calls, object creation, and late binding. A type system for Creol has been introduced in this paper, distinguishing data types, interfaces, and classes. Type checking asynchronous method calls is based on a type and effect system. It is shown that runtime type errors do not occur for well-typed programs, including asynchronous method calls, nondeterministic choice and merge, multiple inheritance, object creation, and late binding of internal methods using the *pruned binding strategy*.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, NY, 1996.

[2] E. Ábrahám-Mumm, F. S. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In *International Conference on Foundations of Software Science and Computation Structures (FOSSACS'02)*, volume 2303 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, Apr. 2002.

[3] G. A. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In E. Najm and J.-B. Stefani, editors, *Proc. 1st IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'96)*, pages 135–153, Paris, 1996. Chapman & Hall.

[4] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, Jan. 1997.

[5] P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In N. Meyrowitz, editor, *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 25(10), pages 161–168, New York, NY, Oct. 1990. ACM Press.

[6] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[7] I. Attali, D. Caromel, and S. O. Ehmety. A natural semantics for Eiffel dynamic binding. *ACM Transactions on Programming Languages and Systems*, 18(6):711–729, 1996.

[8] D. A. Basin and M. Rusinowitch, editors. *Proceedings of the Second International Joint Conference on Automated Reasoning (IJCAR 2004)*, volume 3097 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[9] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C$^\sharp$. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, Sept. 2004.

[10] L. Bettini, V. Bono, R. D. Nicola, G. L. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The Klaim project: Theory and practice. In C. Priami, editor, *Global Computing. Programming Environments, Languages, Security, and Analysis of Systems*, volume 2874 of *Lecture Notes in Computer Science*, pages 88–150. Springer-Verlag, 2003.

[11] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[12] P. Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, Apr. 1999.

[13] K. B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. The MIT Press, Cambridge, Mass., 2002.

[14] K. B. Bruce, A. Schuett, R. van Gent, and A. Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems*, 25(2):225–290, 2003.

[15] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2-3):138–164, 1988.

[16] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[17] D. Caromel and Y. Roudier. Reactive programming in Eiffel//. In J.-P. Briot, J. M. Geib, and A. Yonezawa, editors, *Proceedings of the Conference on Object-Based Parallel and Distributed Computation*, volume 1107 of *Lecture Notes in Computer Science*, pages 125–147. Springer-Verlag, Berlin, 1996.

[18] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[19] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 157–200. Springer-Verlag, 1999.

[20] C. Chambers, D. Ungar, B.-W. Chang, and U. Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *Lisp and Symbolic Computation*, 4(3):207–222, 1991.

[21] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.

[22] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, Nov. 1994.

[23] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 125–135. ACM Press, Jan. 1990.

[24] G. Cugola and C. Ghezzi. CJava: Introducing concurrent objects in Java. In M. E. Orlowska and R. Zicari, editors, *4th International Conference on Object Oriented Information Systems (OOIS'97)*, pages 504–514, London, 1997. Springer-Verlag.

[25] O.-J. Dahl. Monitors revisited. In A. W. Roscoe, editor, *A Classical Mind, Essays in Honour of C.A.R. Hoare*, pages 93–103. Prentice Hall, 1994.

[26] O.-J. Dahl, B. Myrhaug, and K. Nygaard. (Simula 67) Common Base Language. Technical Report S-2, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, May 1968.

[27] W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A formal semantics of concurrency and communication in Real-Time UML. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *First International Symposium on Formal Methods for Components and Objects (FMCO 2002), Revised Lectures*, volume 2852 of *Lecture Notes in Computer Science*, pages 71–98. Springer-Verlag, 2003.

[28] L. G. DeMichiel and R. P. Gabriel. The common lisp object system: An overview. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proc. European Conference on Object-Oriented Programming (ECOOP'87)*, volume 276 of *Lecture Notes in Computer Science*, pages 151–170. Springer-Verlag, 1987.

[29] P. Di Blasio and K. Fisher. A calculus for concurrent objects. In U. Montanari and V. Sassone, editors, *7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 655–670. Springer-Verlag, Aug. 1996.

[30] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, Aug. 1975.

[31] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[32] J. Dovland, E. B. Johnsen, and O. Owe. Verification of concurrent objects with asynchronous method calls. In *Proceedings of the IEEE International Conference on Software - Science, Technology & Engineering (SwSTE'05)*, pages 141–150. IEEE Computer Society Press, Feb. 2005.

[33] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: Fickle$_{\mathrm{II}}$. *ACM Transactions on Programming Languages and Systems*, 24(2):153–191, 2002.

[34] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the Join-calculus. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 372–385, 1996.

[35] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, Advances in Formal Methods, chapter 1, pages 3–167. Kluwer Academic Publishers, 2000.

[36] A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In U. Nestmann and B. C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages (Nice, France, September 12, 1998)*, volume 16(3) of *Electr. Notes Theor. Comput. Sci.* Elsevier Science Publishers, 1998.

[37] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java language specification*. Java series. Addison-Wesley, Reading, Mass., 2nd edition, 2000.

[38] B. Hailpern and H. Ossher. Extending objects to support multiple interfaces and access control. *IEEE Transactions on Software Engineering*, 16(11):1247–1257, 1990.

[39] C. A. R. Hoare. Monitors: an operating systems structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[40] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ., 1985.

[41] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proc. European Conf. on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 1991.

[42] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[43] International Telecommunication Union. Open Distributed Processing - Reference Model parts 1–4. Technical report, ISO/IEC, Geneva, July 1995.

[44] G. S. Itzstein and M. Jasiunas. On implementing high level concurrency in Java. In A. Omondi and S. Sedukhin, editors, *Proc. 8th Asia-Pacific Computer Systems Architecture Conference (ACSAC 2003)*, volume 2823 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, 2003.

[45] E. B. Johnsen and O. Owe. A compositional formalism for object viewpoints. In B. Jacobs and A. Rensink, editors, *Proc. 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)*, pages 45–60. Klüwer Academic Publishers, Mar. 2002.

[46] E. B. Johnsen and O. Owe. Object-oriented specification and open distributed systems. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science*, pages 137–164. Springer-Verlag, 2004.

[47] E. B. Johnsen and O. Owe. Inheritance in the presence of asynchronous method calls. In *Proc. 38th Hawaii International Conference on System Sciences (HICSS'05)*. IEEE Computer Society Press, Jan. 2005.

[48] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 2006. To appear. A short version appeared in the proceedings of *SEFM 2004*.

[49] E. B. Johnsen, O. Owe, and E. W. Axelsen. A run-time environment for concurrent objects with asynchronous method calls. In N. Martí-Oliet, editor, *Proc. 5th International Workshop on Rewriting Logic and its Applications (WRLA'04), Mar. 2004*, volume 117 of *Electr. Notes Theor. Comput. Sci.*, pages 375–392. Elsevier Science Publishers, Jan. 2005.

[50] E. B. Johnsen, O. Owe, and I. Simplot-Ryl. A dynamic class construct for asynchronous concurrent objects. In M. Steffen and G. Zavattaro, editors, *Proc. 7th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, volume 3535 of *Lecture Notes in Computer Science*, pages 15–30. Springer-Verlag, June 2005.

[51] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proc. 11th European Conf. on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[52] S. Krogdahl. Multiple inheritance in Simula-like languages. *BIT*, 25(2):318–326, 1985.

[53] C. Laneve. Inheritance in concurrent objects. In H. Bowman and J. Derrick, editors, *Formal methods for distributed processing – a survey of object-oriented approaches*, pages 326–353. Cambridge University Press, 2001.

[54] B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In D. S. Wise, editor, *Proceedings of the SIGPLAN Conference on Programming Lanugage Design and Implementation (PLDI'88)*, pages 260–267, Atlanta, GE, USA, June 1988. ACM Press.

[55] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.

[56] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th Symposium on Principles of Programming Languages (POPL'88)*, pages 47–57. ACM Press, 1988.

[57] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. The MIT Press, Cambridge, Mass., 1993.

[58] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[59] J. Meseguer and G. Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In Basin and Rusinowitch [8], pages 1–44.

[60] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ., 2 edition, 1997.

[61] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In E. Jul, editor, *Proc. 12th European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382. Springer-Verlag, 1998.

[62] G. Milicia and V. Sassone. The inheritance anomaly: ten years after. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1267–1274. ACM Press, 2004.

[63] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.

[64] E. Najm and J.-B. Stefani. A formal semantics for the ODP computational model. *Computer Networks and ISDN Systems*, 27:1305–1329, 1995.

[65] O. Nierstrasz. A tour of Hybrid – A language for programming with active objects. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 167–182. Prentice Hall, 1992.

[66] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In L. Cardelli, editor, *Proc. 17th European Conf. on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer-Verlag, 2003.

[67] M. Philippsen. A survey on concurrent object-oriented languages. *Concurrency: Practice and Experience*, 12(10):917–980, Aug. 2000.

[68] B. C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Mass., 2002.

[69] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press, 1998.

[70] J. G. Rossie Jr., D. P. Friedman, and M. Wand. Modeling subobject-based inheritance. In P. Cointe, editor, *Proc. 10th European Conf. on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, pages 248–274. Springer-Verlag, July 1996.

[71] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In L. Cardelli, editor, *Proc. 17th European Conf. on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer-Verlag, 2003.

[72] A. Snyder. Inheritance and the development of encapsulated software systems. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 165–188. The MIT Press, 1987.

[73] N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Proc. 5th International Conference on Software Reuse (ICSR5)*, pages 206–215. IEEE Computer Society Press, 1998.

[74] B. Stroustrup. Multiple inheritance for C++. *Computing Systems*, 2(4):367–395, Dec. 1989.

[75] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[76] M. Tokoro and R. Pareschi, editors. volume 821 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[77] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, pages 359–369. ACM Press, 1996.

[78] V. T. Vasconcelos. Typed concurrent objects. In Tokoro and Pareschi [76], pages 100–117.

[79] D. Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, Feb. 1995.

[80] R. J. Wirfs-Brock and R. E. Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, 1990.

[81] Y. Yokote and M. Tokoro. Concurrent programming in ConcurrentSmalltalk. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. The MIT Press, Cambridge, Mass., 1987.

[82] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System.* Series in Computer Systems. The MIT Press, 1990.

Chapter 8

Paper 2: Backwards Type Analysis of Asynchronous Method Calls

Einar Broch Johnsen and Ingrid Chieh Yu

Publication.
Journal of Logic and Algebraic Programming, 2008.

Abstract.
Asynchronous method calls have been proposed to better integrate object orientation with distribution. In the Creol language, asynchronous method calls are combined with so-called processor release points in order to allow concurrent objects to adapt local scheduling to network delays in a very flexible way. However, asynchronous method calls complicate the type analysis by decoupling input and output information for method calls, which can be tracked by a type and effect system. Interestingly, backwards type analysis simplifies the effect system considerably and allows analysis in a single pass. This paper presents a kernel language with asynchronous method calls and processor release points, a novel mechanism for local memory deallocation related to asynchronous method calls, an operational semantics in rewriting logic for the language, and a type and effect system for backwards analysis. Source code is translated into runtime code as an effect of the type analysis, automatically inserting inferred type information in method invocations and operations for local memory deallocation in the process. We establish a subject reduction property, showing in particular that method lookup errors do not occur at runtime and that the inserted deallocation operations are safe.

8.1 Introduction

In the distributed setting, the object-oriented programming model may be criticized for its tight coupling of communication and synchronization, as found in, for example, remote procedure calls. Creol is a novel object-oriented language which targets distributed systems by combining *asynchronous method calls* with *processor release points* inside concurrent objects [19]. The language is class-based, supports inheritance, and object interaction happens through method calls only. Processor release points influence the implicit control flow inside concurrent objects. This way, the object may temporarily suspend its current activity and proceed with another enabled activity. This allows the execution to adapt to network delays in a flexible way, and objects to dynamically change between active and reactive behavior (client and server). Asynchronous method calls decouple the invocation and the return of method calls as seen from the caller. As a result the execution may continue after an external method is invoked, introducing a notion of concurrency similar to that of *futures* [40, 23, 6, 11, 13]. This decoupling is *linear*; i.e., exactly one method invocation is required for every return and at most one return is allowed for every invocation. For asynchronous method calls in Creol, processor release points may be associated with method returns, extending this notion of concurrency. Baker and Hewitt observed that futures need not be used in the scope in which they are bound, leading to memory leakage [3]. A similar issue arises with returns to asynchronous calls in Creol, as a caller may ignore method replies in some or all execution branches after the method invocation.

A type system for Creol is presented in [20]. However asynchronous calls complicate the type analysis by decoupling the call's input and output information, leading to a complex type and effect system [31] to track information during type analysis. Furthermore, the memory leakage due to method replies is not addressed. The type system can be significantly simplified for backwards type analysis, allowing each asynchronous call to be analyzed directly at the call site. This paper develops a type based translation of source programs into runtime code, based on backwards type analysis. The main contributions of this paper are:

- a novel light-weight mechanism for local garbage collection, addressing redundant method replies by explicit deallocations in the runtime code. The mechanism is integrated in a simplified kernel language which focuses on Creol's asynchronous method call mechanism, and in an executable operational semantics for this kernel language in rewriting logic;
- a type and effect system for the language, designed for backwards analysis, which directly translates source code into runtime code during type analysis;
- the exploitation of the backwards analysis to simplify the extraction of runtime signatures for asynchronous method calls;
- the exploitation of type analysis to automatically insert deallocation operations for local garbage collection, addressing the memory leakage problem caused by redundant method replies; and
- subject reduction showing the correctness of late binding and of the deallocation mechanism.

Whereas a traditional tracing garbage collector cannot free memory space that is reachable from the state of an object [21], we exploit type and effect analysis to identify method replies that are semantically garbage in specific execution paths and insert operations to explicitly free memory in those paths. The backwards analysis allows signature extraction and operations for garbage collection to be inserted in the code as it is type checked, in a single pass. The type system is given an algorithmic formulation and is directly implementable.

Paper overview. The remainder of this paper is structured as follows. Section 8.2 defines the kernel language with asynchronous method calls and Section 8.3 defines a type based translation into runtime code. Section 8.4 presents the operational semantics and Section 8.5 the runtime type system and its properties, Section 8.6 discusses related work, and Section 8.7 concludes the paper.

8.2 Creol: A Language for Distributed Concurrent Objects

An object in Creol has an encapsulated state on which various processes execute. A process corresponds to the activation of one of the object's methods. Objects are concurrent in the sense that each object has a processor dedicated to executing the processes of that object, so processes in different objects execute in parallel. The state is encapsulated in the sense that external manipulation of the object state is indirect by means of calls to the object's methods. Only one process may be active in an object at a time; the other processes in the object are *suspended*. We distinguish between *blocking* a process and *releasing* a process. Blocking causes process execution to stop, but does not hand control over to a suspended process. Releasing a process suspends the execution of that process and reschedules control to another (suspended) process. Thus, if a process is blocked there is no execution in the object, whereas if a process is released another process in the object may execute. Although processes need not terminate, the object may combine the execution of several processes using release points within method bodies. At a release point, the active process may be released and a suspended process may be activated.

A process which makes a remote method call must *wait* for the return of the call before proceeding with its activity. In a distributed setting this limitation is severe; delays and instabilities may cause much unnecessary waiting. In an unreliable environment where communication can disappear, the process can even be permanently blocked (and the object deadlocked). Asynchronous method calls allow the process to be either blocked or released by introducing a processor release point between the invocation of the method and the access to its return value. Release points, expressed using Boolean guards, influence the implicit control flow inside objects by allowing process release when the guard evaluates to `false`. This way, processes may choose between blocking and releasing control while waiting for the reply to a method call. Remark that the use of release points makes it straightforward to combine active (e.g., nonterminating) and reactive processes in an object. Thus, an object may behave both as a client and as a server for other objects without the need for an active loop to control the interleaving of these different roles.

Syntax. A kernel subset of Creol is given in Figure 8.1, adapting the syntax of Featherweight Java [17]. For a syntactic construct c, we denote by $\overline{c}$ a list of such constructs. We assume given a simple functional language of expressions e without side effects. A program

$$
\begin{array}{l}
P ::= \overline{L}\ \{\overline{T\ x}; s_p\} \\
L ::= \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{T\ f}; \overline{M}\} \\
M ::= T\ m\ (\overline{T\ x})\{\overline{T\ x}; s_p; \texttt{return}\ e\} \\
s_p ::= s \mid s_p; s_p \mid s_p \Box s_p \mid s_p \,|||\, s_p \mid t!e.m(\overline{e}) \\
s\ ::= v := e \mid v := \texttt{new}\ C() \mid \texttt{await}\ g \mid \texttt{release} \mid \texttt{skip} \mid t?(v) \\
g\ ::= b \mid t? \mid g \wedge g
\end{array}
$$

Figure 8.1: The language syntax. Here, program variables v may be fields (f) or local variables (x), a type T may be a class name C, a `Bool`, or a `Label`, b is an expression of type `Bool` and t is a variable of type `Label`.

P is a list of class definitions followed by a method body. A class extends a superclass, which may be `Object`, with additional fields $\overline{f}$, of types $\overline{T}$, and methods $\overline{M}$. A method's formal parameters and local variables $\overline{x}$ of types $\overline{T}$, are declared at the start of the method body. The self reference `this` provides access to the object running the current method and `return` e dispatches the value of e to the method's caller. The statement `release` is an unconditional process release point. Conditional release points are written `await` g, where g may be a conjunction of Boolean guards b and method reply guards $t?$ (for some label variable t). An asynchronous call is written $t!e.m(\overline{e})$, where t provides a reference to the call, e is an object expression (when e evaluates to `this`, the call is local), m a method name, and $\overline{e}$ an expression list with the actual in-parameters supplied to the method. A blocking reply assignment is written $t?(v)$; here, the result from the call is assigned to v. A nonblocking call combines reply guards with a reply assignment; e.g., in $t!e.m(\overline{e})$; `await` $t?; t?(v)$ the processor may be released while waiting for the reply, allowing other processes to execute. A synchronous call is obtained by immediately succeeding an invocation by a (blocking) reply assignment on the same label, e.g., $t!e.m(\overline{e}); t?(v)$. Reply assignments can be omitted if return values are not needed. Finally, $\overline{s}_1 \Box \overline{s}_2$ is the nondeterministic choice and $\overline{s}_1 \,|||\, \overline{s}_2$ is nondeterministic merge between $\overline{s}_1$ and $\overline{s}_2$, defined as $\overline{s}_1; \overline{s}_2 \Box \overline{s}_2; \overline{s}_1$. Informal explanations of the language constructs are given in Examples 1 and 2 below.

The correspondence between the label variables associated with method invocations, reply guards, and reply assignments can be made more precise by adapting the notion of live variables [27] to labels. Reply guards and assignments are read operations on a label variable t, and invocations are write operations. Say that t is *live* before a statement list $\overline{s}$ if there exists an execution path through $\overline{s}$ where a read operation on t precedes the next write operation on t. Note that reply assignments are *destructive* read operations; for a label t, only a write operation on t may succeed a reply assignment on t.

Definition 1 Let $\overline{s}$ be a list of program statements and t a label. Define $live(t, \overline{s})$ by induction

```
class Node{
 DB $db$;
 List[Str] $enquire()$ {...}
 List[Data] $getPacket$(Str $fId$; Nat $pNbr$) {...}
 Str $reqFile$(Node $nId$; Str $fId$) {...}

 List[Node × List[Str]] $availFiles$(List[Node] $nList$) {
   Label $t_1$; Label $t_2$; List[Str] $fList$; List[Node × List[Str]] $files$;
   if ($nList = empty$) then $files := empty$
   else $t_1!hd(nList).enquire()$; $t_2!this.availFiles(tl(nList))$;
    await $t_1? \wedge t_2?$; $t_1?(fList)$; $t_2?(files)$; $files := ((hd(nList), fList); files)$ fi;
   return $files$}
}
```

Figure 8.2: A class capturing nodes in a peer-to-peer network

over statement lists as follows:

$$\begin{aligned}
&live(t,\varepsilon) = \texttt{false}\\
&live(t,t?(v);\overline{s}) = \texttt{true}\\
&live(t,\texttt{await}\ g;\overline{s}) = \texttt{true} \qquad \text{if } g \text{ contains } t?\\
&live(t,t!o.m(\overline{e});\overline{s}) = \texttt{false}\\
&live(t,\overline{s}_1 \Box \overline{s}_2;\overline{s}) = live(t,\overline{s}_1;\overline{s}) \vee live(t,\overline{s}_2;\overline{s})\\
&live(t,s;\overline{s}') = live(t,\overline{s}') \qquad \text{otherwise}
\end{aligned}$$

Here and throughout the paper, "otherwise" denotes the remaining cases for an inductive definition. Given a statement list $t!e.m(\overline{e});\overline{s}$, the return values from the method call $t!e.m(\overline{e})$ will never be used if $\neg live(t,\overline{s})$. In this case the asynchronous method call gives rise to memory leakage through passive storage of the future containing the method reply, as observed by Baker and Hewitt [3]. This information is exploited in Section 8.3 to automatically insert an explicit instruction `free`(t) for the *runtime deallocation* of the future when this memory leakage can be statically detected be the type analysis.

Example 1 We consider a peer-to-peer file sharing system consisting of nodes distributed across a network. In the example a node plays both the role of a server and of a client. A node may request a file from another node in the network and download it as a series of packets until the file download is complete. As nodes may appear and disappear in the unstable network, the connection to a node may be blocked, in which case the download should automatically resume if the connection is reestablished.

In the `Node` class (Figure 8.2), the methods *reqFile*, *enquire*, and *getPacket* requests the file associated with a file name from a node, enquires which files are available from a node, and fetches a particular packet in a file transmission, respectively. To motivate the use of asynchronous method calls and release points, we consider the method *availFiles* in detail. The method takes as formal parameter a list *nList* of nodes and, for each node, finds the files that may be downloaded from that node. The method returns a list of pairs, where each pair contains a node name and a list of (available) files. All method calls are *asynchronous*. Therefore, after the first call, identified by label t_1, the rest of the node list may be explored recursively, without waiting for the reply to the first call. *Process release points* ensure that if a node temporarily becomes unavailable, the process is suspended and may resume at any time after the node becomes available again. Thus, if the statement `await` $t_1? \wedge t_2?$ evaluates to false, the processor will not be blocked, and other processes may execute. A node may have several interleaved activities: several downloads may be processed simultaneously with uploads to other nodes, etc. If `await` $t_1? \wedge t_2?$ is enabled, the return value *files* is assigned to the out parameter of the caller (and retrieved by $t_2?(\mathit{files})$), which completes the process.

While asynchronous invocations and process release points allow objects to dynamically change between active and reactive behavior, nondeterministic choice and merge statements allow different tasks within a process to be selected based on how communication with other objects actually occur. A process may take advantage of delays and adapt to the environment without yielding control to other processes. A typical computational pattern in many wide-area applications is to acquire data from one or more remote services, manipulate these data, and invoke other remote services with the results. We show how Creol's high-level branching structures can be applied in wide-area computing through the following examples, adapted from [25].

Example 2 Consider services which provide news at request. These may be modeled by a class `News` offering a method *news* to the environment, which for a given date returns an XML document with the news of that day as presented by a "site". Let *CNN* and *BBC* be two such sites; i.e., two variables bound to instances of the class `News`. By calling *CNN.news*(d) or *BBC.news*(d) the news for a specified date d will be downloaded. Another service distributes emails to clients at request, modeled by a class `Email` with a method $\mathit{send}(m,a)$ where m is some message content and a is the address of a client. We define a class

```
class NewsService {
 News CNN; Email email;
 Bool newsRelay(Date d, Client a){
   XML v, Label t; CNN := new News(); t!CNN.news(d); await t?; t?(v);
   email := new Email(); !email.send(v,a); return true }
}
```

In this class, a method relays news from CNN for a given date d to a given client a by invoking the *send* service of the *email* object with the result returned from the call to *CNN.news*, if *CNN* responds. For simplicity, the method returns `true` upon termination. Note that an object executing a *newsRelay* process is not blocked while waiting for the object to respond. Instead the process is suspended until the response has arrived. Once the object responds, the process becomes enabled and may resume its execution, forwarding the news by email. If the object never responds the object may proceed with its other activity, only the suspended process will never become enabled.

Now consider the case where a client wants news from both objects *CNN* and *BBC*. Because there may be significant delays until an object responds, the client desires to have the news from each object relayed whenever it arrives. This is naturally modeled by the merge operator, as in the following method:

```
Bool $newsRelay2$(Date $d$, Client $a$) {
  XML $v$, Label $t$, Label $t'$;
  $t!CNN.news(d)$; $t'!BBC.news(d)$; (await $t?$; $t?(v)$; $t!email.send(v,a)$)
  $|||$ (await $t'?$; $t'?(v)$; $t'!email.send(v,a)$); return true }
```

The merge operator allows the news pages from *BBC* and *CNN* to be forwarded to the *email.send* service once they are received. If neither service responds, the whole process is suspended and can be activated again when a response is received. By executing the method, at most two emails are sent.

If the first news page available from either *CNN* or *BBC* is desired, the nondeterministic choice operator may be used to invoke the *email.send* service with the result received from either *CNN* or *BBC*, as in the following method:

```
Bool $newsRelay3$(Date $d$, Client $a$){
  XML $v$, Label $t$, Label $t'$; $t!CNN.news(d)$; $t'!BBC.news(d)$;
  ((await $t?$; $t?(v)$) $\Box$ (await $t'?$; $t'?(v)$)); $t!email.send(v,a)$; return true }
```

Here news is requested from both news objects, but only the first response will be relayed. The latest arriving response is ignored.

Asynchronous method calls add flexibility to execution in the distributed setting but complicate the type analysis. For asynchronous calls the type information provided at the call site is not sufficient to ensure the correctness of method binding as the out-parameter type is unavailable. In Creol, the correspondence between in- and out-parameters is controlled by label values. The combination of branching and asynchronous calls further complicates the type analysis as there may be several potential invocation and reply pairs associated with a label value at a point in the execution. For example, consider the statement list $(t!o.m(\overline{e});\ldots\ \Box\ t!o'.m'(\overline{e}');\ldots);\ldots;$ $(t?(v)\,\Box\, t?(v'))$ and assume that '$\ldots$' contains no invocations nor replies on the label t. In this case we say that each of the two invocations on t *reaches* each of the two reply assignments, depending on the execution path. Consequently, this use of nondeterministic choice gives us four possible invocation and reply pairs and it is nondeterministic which pair will be evaluated at runtime. In contrast, in $t!o.m(\overline{e})$; $t!o'.m'(\overline{e}')$; $t?(v)$ the second call *redefines* the binding of label t. Here only $t!o'.m'(\overline{e}')$ reaches $t?(v)$, and the reply to the first call cannot be accessed. (It would result in memory leakage unless the first method reply is deallocated). The type system developed in this paper derives a signature for each call such that runtime configurations are well-typed independent of the selected execution branch, and inserts this signature in the runtime code.

8.3 Type Based Translation

Type analysis transforms source programs P into runtime code P_r. For this purpose a type and effect system is used to statically derive signatures for asynchronous method invocations,

$$
\begin{array}{ll}
P_r & ::= \overline{L}_r \; \langle s_r, sub \rangle \\
L_r & ::= \texttt{class}\; C \;\texttt{extends}\; D \;\{sub, \overline{M}_r\} \\
M_r & ::= T\; m\; (\overline{T\; x}) \langle s_r, sub \rangle \\
sub & ::= \overline{v \mapsto val} \\
s_r & ::= s \mid s_r;\; s_r \mid s_r \Box s_r \mid t!e.m(T \to T, \overline{e}) \mid \texttt{free}(\overline{t}) \mid \texttt{return}\; e \\
s & ::= v := e \mid v := \texttt{new}\; C() \mid \texttt{await}\; g \mid \texttt{release} \mid \texttt{skip} \mid t?(v) \\
val & ::= oid \mid \texttt{default}(T) \mid b \mid mid
\end{array}
$$

Figure 8.3: The syntax for runtime code. The syntax for s is as given in Figure 8.1.

inferring the type of the out-parameter from constraints imposed by reply assignments, to ensure that late binding succeeds at runtime. Furthermore in order to efficiently remove redundant method returns, the type system ensures that

- reply assignments $t?(v)$ are type safe;
- reply guards `await` $t?$ are type safe;
- method returns are not deallocated if they can be read later; and
- there is no terminating execution path in which a method return is neither deallocated nor destructively read

By type safety for $t?(v)$, we mean that there is an invocation on t which reaches this reply assignment in every execution path and that v can be type correctly assigned the return values from any such invocation $t!e.m(\overline{e})$. Recall that reply assignments are destructive in the sense that a reply assignment on t consumes an invocation on t. Given a well-typed statement list $t!e.m(\overline{e});\; t?(v);\; \overline{s}$, the first invocation $t!e.m(\overline{e})$ does not reach any statement in $\overline{s}$. By type safety for `await` $t?$, we mean that there is an invocation on t which reaches this reply guard in every execution path. Finally, the deallocation instruction $\texttt{free}(t)$ may only occur at a program point where t is not live but where this instruction is reachable by an invocation on t in every execution path.

The syntax for runtime code is given in Figure 8.3. A state *sub* is a ground substitution mapping variable v to values *val*, which include identifiers *oid* for objects and *mid* for method calls. Let $\texttt{default}(T)$ denote some value of type T. In contrast to source programs, the invocation $t!e.m(T \to T, \overline{e})$ has been expanded with type information and there is an explicit statement $\texttt{free}(\overline{t})$ which allows the deallocation of messages corresponding to the labels $\overline{t}$.

Let $(\mathcal{T}, \preceq)$ be a poset of nominal types with `Data` as its top and `Error` as its bottom element; $\mathcal{T}$ includes `Bool`, `Label`, and class names, and for all types $T \in \mathcal{T}$, $\texttt{Error} \preceq T \preceq \texttt{Data}$. The reflexive and transitive partial order $\preceq$ expresses subtyping and restricts a structural subtype relation which ensures substitutability; If $T \preceq T'$, then any value of type T can masquerade as a value of T'. The subtype relation is derived from the class hierarchy: If a class C extends a class C', then $C \preceq C'$. However, if the methods of C include those of C' (with signatures obeying the subclass relation for function types, defined below), we may safely extend $\preceq$ with $C \preceq C'$ since the internal state of other objects is inaccessible in the language. For product types R and R', $R \preceq R'$ is the point-wise extension of the subtype relation. For the typing and binding of methods, $\preceq$ is extended to function spaces $A \rightarrow B$, where A is a (possibly empty) product type: $A \rightarrow B \preceq A' \rightarrow B' \Leftrightarrow A' \preceq A \wedge B \preceq B'$. Let $T \cap T'$ denote a (largest) common subtype of T and T'. (In fact there may be more than one, in which case any can be selected.) Assume that $\perp \notin \mathcal{T}$ and let $\mathcal{T}_\perp = \mathcal{T} \cup \{\perp\}$. We lift type intersection to $\mathcal{T}_\perp$ by defining $\perp \cap T = T$ for all $T \in \mathcal{T}_\perp$ and lift $\preceq$ to $\mathcal{T}_\perp$ by defining $\perp \not\preceq T$ and $T \not\preceq \perp$ for all $T \in \mathcal{T}$. Conceptually, $\perp$ represents missing type information.

Mappings bind variables and constants to types in $\mathcal{T}_\perp$ and are built from the empty mapping ε by means of bindings $[v \mapsto e]$. For a mapping σ and variables v and v', *mapping application* is defined by $\varepsilon(v) = \perp$, $\sigma[v \mapsto e](v) = e$, and $\sigma[v \mapsto e](v') = \sigma(v')$ if $v' \neq v$. A mapping σ is *well-defined* if $\sigma(v) \neq \texttt{Error}$ for every v. Let $\sigma \cdot \sigma'$ denote the *concatenation* of mappings σ and σ', defined by $\sigma \cdot \varepsilon = \sigma$ and $\sigma \cdot \sigma'[v \mapsto e] = (\sigma \cdot \sigma')[v \mapsto e]$. The domain and range of a mapping are defined by induction; $\text{dom}(\varepsilon) = \text{ran}(\varepsilon) = \emptyset$, $\text{dom}(\sigma[v \mapsto e]) = \{v\} \cup \text{dom}(\sigma)$, and $\text{ran}(\sigma[v \mapsto e]) = \{e\} \cup \text{ran}(\sigma)$. Let $\sigma \subseteq \sigma'$ if $\sigma(x) = \sigma'(x)$ for all $x \in \text{dom}(\sigma)$. For convenience, we let $[x_1 \mapsto T_1, x_2 \mapsto T_2]$ denote $\varepsilon[x_1 \mapsto T_1][x_2 \mapsto T_2]$. Note that a variable may be explicitly bound to $\perp$. Consequently we get $\text{dom}(\varepsilon[v \mapsto \perp]) \neq \text{dom}(\varepsilon)$, although $\varepsilon[v \mapsto \perp](v') = \varepsilon(v')$ for all v'. For mappings σ and σ', we define their *union* by $\sigma \cup \sigma'(v) = \sigma(v) \cap \sigma'(v)$; i.e., the mapping union includes bindings which only occur in one mapping and takes a (least) common subtype if a variable is bound in both mappings.

The type analysis uses a type and effect system, given in Figure 8.4. Let Γ and Δ be well-defined mappings from variable and constant to type names. Judgments $\Gamma, \Delta \vdash \bar{s} \rhd \bar{s}_r, \Delta'$ express that $\bar{s}$ is well-typed in the typing context Γ of program and Δ of label variables, $\bar{s}_r$ is an *expansion* of $\bar{s}$ into runtime code, and Δ' is an *update* of Δ. (To improve readability, $\bar{s}_r$ is omitted if there is no expansion and Δ' is omitted if there is no update.) The analysis is *from right to left*, as apparent from (SEQ): $\bar{s}_2$ is analyzed first, and $\bar{s}_1$ is analyzed in the context updated by the effect of the analysis of $\bar{s}_2$. The rule is not symmetric since mapping composition is not commutative. In the type rules, let T, T', and T_0 range over types in $\mathcal{T} \setminus \{\texttt{Error}\}$.

Rule (SEQ) illustrates the backwards analysis approach. In the analysis of $\bar{s}_1$; $\bar{s}_2$, the statements $\bar{s}_2$ are analyzed first, deriving a context update Δ_2. The statements $\bar{s}_1$ are analyzed subsequently and depend on the effect of the analysis of $\bar{s}_2$. Rules (ASSIGN), (NEW), (RELEASE), and (SKIP) are as expected and have no effect. However, assignment to label variables is not allowed. For simplicity, we identify the empty list ε of program statements with `skip`. The analysis of `await` statements decomposes guards with ($\wedge$-GUARDS). Here, the effects of the analysis of the two branches are composed into the joint effect of the conjunction. Rule (B-GUARDS) for Boolean guards has no effect. However, for a reply guard $t?$, (t-GUARDS) records an effect which places a type constraint on t. The type constraint may be understood as follows: If $\Delta(t) = \perp$, there is no previous constraint on t and the effect records the weakest constraint possible; i.e., $[t \mapsto \texttt{Data}]$. However, if there is a previous constraint on t, this constraint is kept; i.e, if $\Delta(t) \neq \perp$, then $\Delta(t) \cap \texttt{Data} = \Delta(t)$. For the analysis of a statement `await` g in (AWAIT), the type system analyzes the guard g, deriving an effect Δ'. There are two possibilities, depending on whether new constraints have been introduced in the analysis of g. This corresponds to deciding $live(t, \bar{s})$, where $\bar{s}$ is the statement list with effect Δ and $t \in \text{dom}(\Delta')$. Thus, if $t \in \text{dom}(\Delta')$ and $\neg live(t, \bar{s})$ then t will not be read in $\bar{s}$ and can be deallocated after executing `await` g. In the type system, the new constraints are collected in a set t. If $L = \emptyset$, the instruction `free`$(\emptyset)$ is inserted, which is considered the same as `skip`. On the other hand if $L \neq \emptyset$ then there are new constraints on

$$(\textsc{Seq})\quad \frac{\Gamma,\Delta \vdash \overline{s}_2 \rhd \overline{s}'_2,\Delta_2 \quad \Gamma,\Delta\cdot\Delta_2 \vdash \overline{s}_1 \rhd \overline{s}'_1,\Delta_1}{\Gamma,\Delta\vdash \overline{s}_1;\ \overline{s}_2 \rhd \overline{s}'_1;\ \overline{s}'_2,\Delta_2\cdot\Delta_1} \qquad (\textsc{Skip})\quad \frac{}{\Gamma,\Delta\vdash \mathtt{skip}\rhd}$$

$$(\textsc{Assign})\quad \frac{\Gamma\vdash e:T' \quad \Gamma(v)\neq \mathsf{Label} \quad T'\preceq\Gamma(v)}{\Gamma,\Delta\vdash v:=e\rhd} \qquad (\textsc{New})\quad \frac{C\preceq\Gamma(v)}{\Gamma,\Delta\vdash v:=\mathtt{new}\,C()\rhd}$$

$$(\wedge\textsc{-guards})\quad \frac{\Gamma,\Delta\vdash g_1\rhd g_1,\Delta_1 \qquad \Gamma,\Delta\vdash g_2\rhd g_2,\Delta_2}{\Gamma,\Delta\vdash g_1\wedge g_2\rhd\ \Delta_1\cdot\Delta_2} \qquad (\textsc{Release})\quad \frac{}{\Gamma,\Delta\vdash \mathtt{release}\rhd}$$

$$(t\textsc{-guards})\quad \frac{\Gamma(t)=\mathtt{Label}}{\Gamma,\Delta\vdash t?\rhd\ [t\mapsto\Delta(t)\cap\mathtt{Data}]} \qquad (\textsc{b-guards})\quad \frac{\Gamma\vdash b:\mathtt{Bool}}{\Gamma,\Delta\vdash b\rhd}$$

$$(\textsc{Await})\quad \frac{\Gamma,\Delta\vdash g\rhd g,\Delta' \quad L=\{t\mid\Delta'(t)=\mathtt{Data}\wedge\Delta(t)=\bot\}}{\Gamma,\Delta\vdash \mathtt{await}\ g\rhd\mathtt{await}\ g;\ \mathtt{free}(L),\Delta'}$$

$$(\textsc{Reply})\quad \frac{\Gamma(t)=\mathtt{Label} \qquad \Delta(t)=\bot \qquad \Gamma(v)=T}{\Gamma,\Delta\vdash t?(v)\rhd[t\mapsto T]}$$

$$(\textsc{Call1})\quad \frac{\Gamma(t)=\mathtt{Label} \quad \Gamma\vdash\overline{e}:T \quad \Gamma\vdash e:C \quad \Delta(t)\neq\bot \quad \mathtt{lookup}(C,m,T\to\Delta(t))}{\Gamma,\Delta\vdash t!e.m(\overline{e})\rhd t!e.m(T\to\Delta(t),\overline{e}),[t\mapsto\bot]}$$

$$(\textsc{Call2})\quad \frac{\Gamma(t)=\mathtt{Label} \quad \Gamma\vdash\overline{e}:T \quad \Gamma\vdash e:C \quad \Delta(t)=\bot \quad \mathtt{lookup}(C,m,T\to\mathtt{Data})}{\Gamma,\Delta\vdash t!e.m(\overline{e})\rhd t!e.m(T\to\mathtt{Data},\overline{e});\ \mathtt{free}(t),[t\mapsto\bot]}$$

$$(\textsc{Choice})\quad \frac{\begin{array}{c}\Gamma,\Delta\vdash\overline{s}_1\rhd\overline{s}'_1,\Delta_1 \qquad \Gamma,\Delta\vdash\overline{s}_2\rhd\overline{s}'_2,\Delta_2 \qquad \mathit{well\text{-}defined}((\Delta\cdot\Delta_1)\cup(\Delta\cdot\Delta_2))\\ L_1=\{t\mid\Delta_1(t)\preceq\mathtt{Data}\wedge\Delta(t)=\bot\} \qquad L_2=\{t\mid\Delta_2(t)\preceq\mathtt{Data}\wedge\Delta(t)=\bot\}\end{array}}{\Gamma,\Delta\vdash\overline{s}_1\Box\overline{s}_2\rhd\mathtt{free}(L_2\setminus L_1);\ \overline{s}'_1\Box\mathtt{free}(L_1\setminus L_2);\ \overline{s}'_2,(\Delta\cdot\Delta_1)\cup(\Delta\cdot\Delta_2)}$$

$$(\textsc{Method})\quad \frac{\begin{array}{c}\Gamma'=\Gamma[\overline{x\mapsto T_0}][\overline{x'\mapsto T'}] \qquad T_0\neq\mathsf{Label}\\ \Gamma'\vdash e:T \qquad \Gamma',\varepsilon\vdash\overline{s}\rhd\overline{s}',\Delta \qquad \mathrm{ran}(\Delta)=\{\bot\}\end{array}}{\begin{array}{l}\Gamma,\varepsilon\vdash T\ m\ (\overline{T_0\,x})\{\overline{T'\,x'};\ \overline{s};\ \mathtt{return}\ e\}\\ \qquad\rhd T\ m\ (\overline{T_0\,x})\langle\overline{s}';\ \mathtt{return}\ e,\overline{x'\mapsto\mathtt{default}(T')}\rangle\end{array}}$$

$$(\textsc{Class})\quad \frac{\mathtt{for\ all}\ M\in\overline{M}\ \cdot\ \Gamma[\mathtt{fields}(C)],\varepsilon\vdash M\rhd M'}{\begin{array}{l}\Gamma,\varepsilon\vdash\mathtt{class}\ C\ \mathtt{extends}\ D\ \{\overline{T\ f};\ \overline{M}\}\\ \qquad\rhd\mathtt{class}\ C\ \mathtt{extends}\ D\ \{\overline{f\mapsto\mathtt{default}(T)},\overline{M}'\}\end{array}}$$

$$(\textsc{Program})\quad \frac{\mathtt{for\ all}\ L\in\overline{L}\ \cdot\ \varepsilon,\varepsilon\vdash L\rhd L' \qquad [\overline{x\mapsto T}],\varepsilon\vdash\overline{s}\rhd\overline{s}'}{\Gamma_0,\varepsilon\vdash\overline{L}\ \{\overline{T\ x};\ \overline{s}\}\rhd\overline{L}'\ \langle\overline{s}',\overline{x\mapsto\mathtt{default}(T)}\rangle}$$

Figure 8.4: The type and effect system, where static type information is captured by an initial mapping $\Gamma_0=[\mathtt{true}\mapsto\mathtt{Bool},\mathtt{false}\mapsto\mathtt{Bool},\mathtt{default}(T)\mapsto T]$ for all T.

labels in t. In this case, these constraints are propagated as the effect and since we know that $\neg\mathit{live}(t,\overline{s})$ for all $t\in L$, these labels are never needed later, they are deallocated in the runtime code, i.e., by the statement free(L).

When arriving at a reply assignment $t?(v)$, it is required that a corresponding asynchronous invocation with label t is encountered later during type checking. Furthermore, the type of v is used to infer a correct output type for the method call on t. For this purpose, (Reply) imposes a constraint $[t \mapsto \Gamma(v)]$ as an effect of the type analysis. The reply assignment is destructive (see rule R11 in the operational semantics of Section 8.4). Therefore a reply assignment is only allowed when $\neg live(t, \overline{s})$ where $\overline{s}$ is the statement list that has already been type checked. Technically, we here require $\Delta(t) = \perp$ in the backwards analysis.

For (Call1) and (Call2) the actual signature of an asynchronous invocation $t!e.m(\overline{e})$ can now be directly derived, as $\Delta(t)$ provides the constraint associated with the out-parameter type. If $\Delta(t) = \perp$, there is no reply assignment or guard corresponding to the invocation, so $\neg live(t, \overline{s})$ where $\overline{s}$ gives effect Δ, and (Call2) applies. In this case any out-parameter type is acceptable, which is checked by $\texttt{lookup}(C, m, T \rightarrow \texttt{Data})$, and the invocation in the runtime code gets this signature. Furthermore, since there is no reply assignment or guard for this invocation, deallocation is immediately introduced after the call. In contrast if $\Delta(t) \neq \perp$, there are reply assignments or guards corresponding to the invocation, and (Call1) applies. In this case, the out-parameter type is given by the constraint $\Delta(t)$ and checked by $\texttt{lookup}(C, m, T \rightarrow \Delta(t))$. Finally, the runtime invocation gets this derived signature and the label t is reset by updating Δ with the effect $[t \mapsto \perp]$. In this case, deallocation is not needed.

In a nondeterministic choice $\overline{s}_1 \Box \overline{s}_2$, only one branch is evaluated at runtime. Hence, in (Choice) each branch is type checked in the same context. If there is a reply assignment with label t in each of the branches that corresponds to the same invocation, which is an invocation textually occurring to the left of the choice statement, then Δ is updated by mapping t to a largest common subtype of the two return types, i.e., with $[t \mapsto (\Delta_1(t) \cap \Delta_2(t))]$. Note that this type intersection may give $[t \mapsto \texttt{Error}]$, in which case the mapping $(\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2)$ is not well-defined and the rule fails. If the operation succeeds, represented by the predicate *well-defined*, the invocation has a return type that is well-typed for both executions by subtyping. Moreover, the type system ensures that a reply assignment with label t can occur after a nondeterministic choice only if a call with label t is pending after the evaluation of either branch. Deallocation applies in one branch if there are reply assignments or guards only in the other branch. There may be many such labels in each branch, identified by the sets L_1 and L_2. Thus $L_1 \setminus L_2$ and $L_2 \setminus L_1$ contain exactly the labels which should be deallocated in branches $\overline{s}_2$ and $\overline{s}_1$, respectively.

Note that if a call is pending on a label t before a choice statement (say $\Delta(t) = T$) and this call is overwritten in one branch (e.g., $\overline{s}_1$ is $t?(\overline{v})$; $t!o.m(\overline{e})$ so that $\Delta_1(t) = \Gamma(\overline{v})$) but the label is untouched in the other branch (i.e., $t \notin \text{dom}(\Delta_2)$), then updating Δ to $\Delta_1 \cup \Delta_2$ would result in $(\Delta \cdot (\Delta_1 \cup \Delta_2))(t) = \Delta_1 \cup \Delta_2(t) = \Gamma(\overline{v})$. This would be incorrect; if the branch $\overline{s}_2$ were chosen, the reply should still be assigned to $\Delta(t) = T$. In order to avoid losing this information, (Choice) instead updates Δ to $(\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2)$. This has the advantage that if $t \notin \text{dom}(\Delta_2)$ then $\Delta \cdot \Delta_2(t) = \Delta(t)$, which solves the problem of one branch not touching the label. Remark that it may be the case that $t \in \text{dom}(\Delta_2)$ such that $\Delta_2(t) = \perp$, for example if $\overline{s}_2$ is $t!o.m(\overline{e})$. In this case the branch has assigned $\perp$ to t by (Call1) and $\Delta \cdot \Delta_2(t) = \Delta[t \mapsto \perp](t) = \perp$. Thus if $t \in \text{dom}(\Delta_1) \cup \text{dom}(\Delta_2)$, then $((\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2))(t) = (\Delta \cdot (\Delta_1 \cup \Delta_2))(t)$.

In (Class), the function $\texttt{fields}(C)$ provides the typing context for the fields of a class C and its superclass, including $\texttt{this}$ of type C. Similarly in (Method) the type information for input variables and local variables extend the typing context. Note that the analysis in (Method), (Class), and (Program) starts with empty label mappings and the update after analysis in (Method) only contains $\perp$ constraints. Furthermore, labels may not be passed as method parameters. These requirements enforce the encapsulation of method replies within the scope of a method body; a reply assignment or reply guard must be preceded by a method invocation on the same label within the method body. Otherwise a deadlock would be possible when executing the method body, as the processor would block permanently waiting for a reply to a non-existing or previously consumed method invocation. Method calls in the runtime code include signatures derived by the type analysis, which will be used in the runtime method lookup.

Example 3 Now reconsider the wide-area network services of Example 2, which provide news at request. We derive the runtime code generated by the type analysis for *newsRelay*, *newsRelay2*, and *newsRelay3*, assuming that variables *CNN* and *BBC* have type `News` in the class where the methods are declared. Hence, (Class) builds a typing context $\Gamma = [\mathit{CNN} \mapsto \texttt{News}, \mathit{BBC} \mapsto \texttt{News}, \mathit{email} \mapsto \texttt{Email}]$ in which the methods are type checked. For the method

$$\begin{array}{l}\texttt{Bool}\ \mathit{newsRelay}(\texttt{Date}\ d, \texttt{Client}\ a)\{\\ \quad \texttt{XML}\ v, \textsf{Label}\ l;\ \mathit{CNN} := \texttt{new News}();\ t!\mathit{CNN.news}(d);\ \texttt{await}\ t?;\ t?(v);\\ \quad \mathit{email} := \texttt{new Email}();\ t!\mathit{email.send}(v,a);\ \texttt{return true}\}\end{array}$$

we get the typing environment $\Gamma' = \Gamma \cdot [d \mapsto \texttt{Date}, a \mapsto \texttt{Client}, v \mapsto \texttt{XML}, t \mapsto \textsf{Label}]$ and the judgment

$$\begin{array}{c}\Gamma', \varepsilon \vdash \mathit{CNN} := \texttt{new News}();\ t!\mathit{CNN.news}(d);\ \texttt{await}\ t?;\ t?(v);\\ \mathit{email} := \texttt{new Email}();\ t!\mathit{email.send}(v,a) \rhd \overline{s}_r\end{array}$$

where $\overline{s}_r$ is the derived runtime code obtained by the analysis of the method body. We start with the analysis of $t!\mathit{email.send}(v,a)$, which is type checked by (Call2) since $\varepsilon(t) = \perp$. Since $\neg \mathit{live}(t,\varepsilon)$, t will not be read by any reply or guard statements later and the insertion of $\texttt{free}(t)$ is safe. Assuming that *lookup* succeeds for the given types of t, v, and a, we get the following judgment:

$$\Gamma', \varepsilon \vdash t!\mathit{email.send}(v,a) \rhd t!\mathit{email.send}(\texttt{XML} \times \texttt{Client} \rightarrow \texttt{Data}, v, a);\ \texttt{free}(t)$$

We proceed with the statement $\mathit{email} := \texttt{new Email}()$ and get the judgment

$$\Gamma', \varepsilon \vdash \mathit{email} := \texttt{new Email}() \rhd \mathit{email} := \texttt{new Email}()$$

Continue with the statement $t?(v)$, for which we get the judgment

$$\Gamma', \varepsilon \vdash t?(v) \rhd t?(v), [t \mapsto \texttt{XML}]$$

by applying (Reply). For $t?(v)$, we have $\Gamma', [t \mapsto \texttt{XML}] \vdash t? \rhd t?, [t \mapsto \texttt{XML}]$ by (t-Guards), since $\texttt{XML} \cap \texttt{Data} = \texttt{XML}$ and no new constraints on label t have been introduced. Consequently, we can apply (Await) and obtain the judgment

$$\Gamma', [t \mapsto \texttt{XML}] \vdash \texttt{await}\ t? \rhd \texttt{await}\ t?$$

Applying (Call1) to $t!\mathit{CNN.news}(d)$ since $[t \mapsto \texttt{XML}](t) \neq \perp$, we get the judgment

$$\Gamma', [t \mapsto \texttt{XML}] \vdash t!\mathit{CNN.news}(d) \rhd t!\mathit{CNN.news}(\texttt{Date} \rightarrow \texttt{XML}, d), [t = \perp]$$

Finally, we apply rule (New) to $CNN := \texttt{new News()}$ and obtain the judgment

$$\Gamma', \varepsilon \vdash CNN := \texttt{new News()} \rhd CNN := \texttt{new News()}$$

Since $\mathrm{ran}([t = \perp]) = \perp$ the type analysis is correct by rule (Method) for the reassembled runtime code $\bar{s}_r$, which becomes

$$\begin{aligned}&CNN := \texttt{new News()};\ t!CNN.news(\texttt{Date} \to \texttt{XML}, d);\ \texttt{await}\ t?;\ t?(v);\\ &\quad email := \texttt{new Email()};\ t!email.send(\texttt{XML} \times \texttt{Client} \to \texttt{Data}, v, a));\ \texttt{free}(t)\end{aligned}$$

If we let σ denote the state $v \mapsto \texttt{default(XML)}, t \mapsto \texttt{default}(\mathsf{Label}), t' \mapsto \texttt{default}(\mathsf{Label})$, we similarly obtain for *newsRelay2* and *newsRelay3* the following runtime method representations:

```
Bool $newsRelay2(\texttt{Date}\ d, \texttt{Client}\ a)\langle$
  $t!CNN.news(\texttt{Date} \to \texttt{XML}, d);\ t'!BBC.news(\texttt{Date} \to \texttt{XML}, d);$
  $(\texttt{await}\ t?;\ t?(v);\ t!email.send(\texttt{XML} \times \texttt{Client} \to \texttt{Data}, v, a);\ \texttt{free}(t);$
  $\texttt{await}\ t'?;\ t'?(v);\ t'!email.send(\texttt{XML} \times \texttt{Client} \to \texttt{Data}, v, a);\ \texttt{free}(t'))$
  $\Box(\texttt{await}\ t'?;\ t'?(v);\ t'!email.send(\texttt{XML} \times \texttt{Client} \to \texttt{Data}, v, a);\ \texttt{free}(t');$
  $\texttt{await}\ t?;\ t?(v);\ t!email.send(\texttt{XML} \times \texttt{Client} \to \texttt{Data}, v, a);\ \texttt{free}(t));$
  $\texttt{return true}, \sigma\rangle$
Bool $newsRelay3(\texttt{Date}\ d, \texttt{Client}\ a)\langle$
  $t!CNN.news(\texttt{Date} \to \texttt{XML}, d);\ t'!BBC.news(\texttt{Date} \to \texttt{XML}, d);$
    $((\texttt{free}(t');\ \texttt{await}\ t?;\ t?(v))\Box\ (\texttt{free}(t);\ \texttt{await}\ t'?;\ t'?(v)));$
      $t!email.send(\texttt{XML} \times \texttt{Client} \to \texttt{Data}, v, a);\ \texttt{free}(t));\ \texttt{return true}, \sigma\rangle$
```

The type system in Figure 8.4 is basically syntax-directed; i.e., only one rule applies to any syntactic construct. Consequently the implementation of the type system is straightforward: each rule corresponds to one case in the algorithm. There is one exception: the rules (Call1) and (Call2) for method invocation. Here, rule selection is determined by the additional condition $(\Gamma(t) = \perp)$. The exception is due to the insertion of instructions for message deallocation and reflects the low cost of inserting these instructions during the type analysis.

8.3.1 Properties of the Type Based Translation

We consider some properties of the type based translation (for proofs, see Appendix 8.A). Theorem 1 relates live labels to the type system's effect mapping.

Theorem 1 *If* $\Gamma, \varepsilon \vdash \bar{s} \rhd \bar{s}', \Delta'$ *then* $\forall t : \mathsf{Label} \cdot \neg live(t, \bar{s}) \iff \Delta'(t) = \perp$.

The following corollaries follow from Theorem 1, since $\Delta(t) = \perp$ is a premise of the rules (Reply), (Await), and (Call2).

Corollary 1 *If* $\Gamma, \varepsilon \vdash t?(v);\ \bar{s}_0 \rhd t?(v);\ \bar{s}'_0, \Delta'$ then $\neg live(t, \bar{s}_0)$.

Corollary 2 *If* $\Gamma, \varepsilon \vdash s;\ \bar{s}_0 \rhd s';\ \texttt{free}(t);\ \bar{s}'_0, \Delta'$ *then* $\neg live(t, \bar{s}_0)$.

It follows that the type based translation does not introduce too many deallocation statements. Note that only reply assignments and reply guards on a label t make $\Delta(t)$ different from $\bot$, that only (CALL1) actually resets $\Delta(t)$ to $\bot$, and that (METHOD) requires that $\mathrm{ran}(\Delta) = \bot$. Therefore, it follows from Theorem 1 that there are enough method invocations in well-typed programs. Deallocation operations are only inserted by rules (AWAIT) and (CALL2), corresponding to the cases without a (destructive) reply assignment on a label t before a reply guard or invocation on t; i.e., the rules capture the cases when there is another operation on t than a destructive read and t is not live. Consequently, by Theorem 1, there are enough deallocation operations in the runtime code. Thus, the type based translation has the following properties:

1. A deallocation statement for a label t is not introduced when t is live.
2. Every reply assignment and reply guard on a label t in a well-typed program is textually preceded by a corresponding invocation on t.
3. Method returns are deallocated in all program branches where they are not destructively read.

Furthermore, the signatures which expand the method invocations in the runtime code ensure that the assignment of reply values is well-typed in every execution path. This ensures that late binding preserves the well-typedness of the local object state of the caller:

Theorem 2 (Well-typedness of return value) *Given $\Gamma, \varepsilon \vdash \bar{s} \rhd \bar{s}', \Delta$. If $t!o.m(\bar{e})$ is an invocation in $\bar{s}$ which gets translated into $t!o.m(T \to T', \bar{e})$ and $t?(v)$ is reply assignment corresponding to that invocation, then $T' \preceq \Gamma(v)$.*

The requirements to the type system which were stated at the beginning of Section 8.3 are now reconsidered. The type safety of reply guards is given by Property 2 above. The type safety of reply assignments is given by Theorem 2 and Property 2. Property 1 ensures that the deallocation of a label t only occurs when t is not live and Property 3 that deallocation statements are inserted for all invocations that are not destructively read.

8.4 Operational Semantics

The operational semantics of the language is given in rewriting logic [24] and is executable on the Maude machine [7]. A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where the signature Σ defines the function symbols of the language, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. Rewrite rules apply to terms of given types. Types are specified in (membership) equational logic (Σ, E). Different system components will be modeled by terms of the different types defined in the equational logic. The global state configuration is defined as a multiset of these terms.

Rewriting logic extends algebraic specification techniques with transition rules: The dynamic behavior of a system is captured by rewrite rules supplementing the equations which define the term language. Let $\to$ denote the reflexive and transitive closure of the rewrite relation. A rewrite rule $p \to p'$ may be interpreted as a *local transition rule* allowing an instance

$$
\begin{array}{rcl}
\mathit{config} & ::= & \varepsilon \mid \mathit{object} \mid \mathit{msg}\ \mathbf{to}\ o \mid \mathit{config}\ \mathit{config} \\
\mathit{object} & ::= & \langle \mathit{oid} : C \mid \mathit{Fld} : \mathit{sub}, \mathit{Pr} \colon \mathit{active}, \mathit{PrQ} \colon \mathit{processQ}, \mathit{EvQ}_{(\overline{\mathit{mid}}, \overline{\mathit{mid}})} : \mathit{eventQ} \rangle \\
\mathit{msg} & ::= & \mathit{invoc}(m, T \to T, \overline{e}, \langle o, \mathit{mid} \rangle) \mid \mathit{comp}(\mathit{mid}, T, e) \\
\mathit{active} & ::= & \mathit{process} \mid \texttt{idle} \\
\mathit{process} & ::= & \langle s_r, \mathit{sub} \rangle \mid \texttt{lookup-error} \mid \texttt{deallocation-error} \\
\mathit{processQ} & ::= & \varepsilon \mid \mathit{process} \mid \mathit{processQ}\ \mathit{processQ} \\
\mathit{eventQ} & ::= & \varepsilon \mid \mathit{msg} \mid \mathit{eventQ}\ \mathit{eventQ}
\end{array}
$$

Figure 8.5: Runtime configurations.

of the pattern p to evolve into the corresponding instance of the pattern p'. Conditional rewrite rules $p \to p'$ **if** *cond* are allowed, where the condition *cond* can be formulated as a conjunction of rewrites and equations which must hold for the main rule to apply. When auxiliary functions are needed in the semantics, these are defined in equational logic, and are evaluated in between the rewrite transitions. Rewrite rules apply to local fragments of a state configuration. If rewrite rules may be applied to nonoverlapping subconfigurations, the transitions may be performed in parallel. Consequently, concurrency is implicit in rewriting logic.

The runtime configurations of Creol's operational semantics, given in Figure 8.5, are multisets combining objects and messages. An object identifier consists of a value *oid* and a class name. When the class of an object is of no significance, the object identifier is denoted o. With a slight abuse of notation we denote by $\overline{c}$ in the presentation of the operational semantics either a list, set, or multiset of a construct c, depending on the context. An asynchronous method call in the language is reflected by a pair of messages. An invocation message $\mathit{invoc}(m, T \to T', \overline{e}, \langle o, \mathit{mid} \rangle)$ addressed to a callee o' represents a call to method m of object o' with actual signature $T \to T'$, the actual in-parameters $\overline{e}$, and the value *mid* (of type `Label`) is a locally unique label value identifying the call for the caller o. In a corresponding completion message $\mathit{comp}(\mathit{mid}, T, e)$ addressed to a caller o, the label value *mid* identifies the call, and e is reduced to the return value of type T. Thus, the label value associated with a method call identifies the invocation and completion messages for that call.

In order to improve readability in the presentation of the rules the different elements of runtime objects are tagged and elements that are irrelevant for a particular rule are omitted, as usual in rewriting logic. Thus in an object

$$\langle \mathit{oid} : C \mid \mathit{Fld} : \mathit{sub}, \mathit{Pr} \colon \mathit{active}, \mathit{PrQ} \colon \mathit{processQ}, \mathit{EvQ}_{(\overline{l}, \overline{g})} : \mathit{eventQ} \rangle,$$

oid is an object identifier tagged by a class name C, *sub* are the fields, *active* is the active process, *processQ* is the process queue, and *eventQ* is the event queue. (When the class of an object is of no significance, the object identifier $\mathit{oid} : C$ is simply denoted o.) A process consists of a state for local variables and a list of program statements. (Any process derived from a well-typed method definition has `return` e as its final statement.) Note that each field and local process variable is initialized with a type-correct default value. The active process may be `idle`. A special process `lookup-error` is introduced in the program queue to represent failed method lookup. The event queue *eventQ* consists of incoming unprocessed messages to the object and has associated sets $\overline{\mathit{mid}}$ representing completion messages to be deallocated and

$$\begin{array}{ll}
 & \langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle v := e;\ \overline{s}, \overline{l} \rangle\ \rangle \\
\text{(R1)} & \longrightarrow\ \textbf{if}\ v\ \mathit{in}\ \overline{l}\ \textbf{then}\ \langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \overline{s}, (\overline{l}[v \mapsto [\![e]\!]_{(\overline{f}\cdot\overline{l})}]) \rangle\ \rangle \\
 & \qquad\qquad\qquad\quad \textbf{else}\ \ \langle o \,|\, \mathit{Fld} : (\overline{f}[v \mapsto [\![e]\!]_{(\overline{f}\cdot\overline{l})}]), \mathit{Pr} : \langle \overline{s}, \overline{l} \rangle\ \rangle\ \textbf{fi} \\[1ex]
 & \langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle (v := \mathtt{new}\ C();\ \overline{s}), \overline{l} \rangle\ \rangle \\
\text{(R2)} & \longrightarrow\ \langle o \,|\, \mathit{Fld}\!: \overline{f}, \mathit{Pr}\!: \langle (v := (n : C);\ \overline{s}), \overline{l} \rangle\ \rangle \\
 & \langle (n : C) \,|\, \mathit{Fld} : \mathit{fields}(C), \mathit{Pr}\!: \langle \mathtt{this} := (n : C), \varepsilon \rangle, \mathit{PrQ}\!: \varepsilon, \mathit{EvQ}_{(\emptyset,\emptyset)} : \varepsilon \rangle\ \textbf{if}\ \mathit{fresh}(n) \\[1ex]
\text{(R3)} & \langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \mathtt{await}\ g;\ \overline{s}, \overline{l} \rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle \\
 & \longrightarrow\ \langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \overline{s}, \overline{l} \rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle\ \textbf{if}\ \mathit{enabled}(g, (\overline{f}\cdot\overline{l}), \overline{q}) \\[1ex]
\text{(R4)} & \langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \mathtt{await}\ g;\ \overline{s}, \overline{l} \rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle \\
 & \longrightarrow\ \langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \mathtt{deallocation\text{-}error}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle\ \textbf{if}\ \mathit{dealloc}(g, (\overline{f}\cdot\overline{l}), \mathcal{G}) \\[1ex]
 & \langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \overline{s}, \overline{l} \rangle, \mathit{PrQ} : \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle \\
\text{(R5)} & \longrightarrow\ \langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \mathtt{release};\ \overline{s}, \overline{l} \rangle, \mathit{PrQ} : \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle \\
 & \ \textbf{if}\ \neg\mathit{enabled}(\overline{s}, (\overline{f}\cdot\overline{l}), \overline{q})\ \wedge\ \neg\mathit{dealloc}(\overline{s}, (\overline{f}\cdot\overline{l}), \mathcal{G}) \\[1ex]
\text{(R6)} & \langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \mathtt{release};\ \overline{s}, \overline{l} \rangle, \mathit{PrQ} : \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle \\
 & \longrightarrow\ \langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \mathtt{idle}, \mathit{PrQ} : (\overline{w}\ \langle \overline{s}, \overline{l} \rangle), \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle \\[1ex]
\text{(R7)} & \langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \mathtt{idle}, \mathit{PrQ} : \langle \overline{s}, \overline{l} \rangle\ \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle \\
 & \longrightarrow\ \langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \overline{s}, \overline{l} \rangle, \mathit{PrQ} : \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle\ \textbf{if}\ \mathit{ready}(\overline{s}, (\overline{f}\cdot\overline{l}), \overline{q}) \\[1ex]
 & \langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle (t!x.m(\mathit{Sig}, \overline{e});\ \overline{s}), \overline{l} \rangle \rangle \\
\text{(R8)} & \longrightarrow\ \langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle (t := n;\ \overline{s}), \overline{l} \rangle \rangle\ \mathit{invoc}(m, \mathit{Sig}, ([\![\overline{e}]\!]_{(\overline{f}\cdot\overline{l})}), \langle o, n \rangle)\ \textbf{to}\ [\![x]\!]_{(\overline{f}\cdot\overline{l})} \\
 & \ \textbf{if}\ \mathit{fresh}(n)
\end{array}$$

Figure 8.6: The operational semantics (1). Any process $\langle \varepsilon, \overline{l} \rangle$ is reduced to `idle`.

completion messages that have been deallocated. A special process `deallocation-error` is introduced to represent the dereferencing of a deallocated completion message. For process and event queues, we let $\emptyset$ be the empty queue and whitespace the associative and commutative constructor (following rewriting logic conventions [7]).

For simplicity, the operational semantics given in Figures 8.6 and 8.7 abstracts from the representation of classes. Consequently the (local) uniqueness of a name n is represented by a predicate $\mathit{fresh}(n)$, which provides (global) uniqueness in combination with either a class or object identifier. We assume given two auxiliary functions which depend on the representation of classes. The function $\mathit{fields}(C)$ returns an initial object state in which `this` and the declared fields of class C and its superclasses are bound to default values. The function *lookup* returns a process, given a class, a method name, the signature of the actual in- and out-parameters to the call, and actual in-parameters. This process has a state in which the local variables have been given default values. The reserved local variables γ, α, and β are instantiated and store values for the method activation's return label, caller, and return type, respectively. Note that we assume that `this` as well as γ, α, and β are read-only variables. Consequently, these local variables ensure that each activation's return value may be correctly returned to its caller when process execution is interleaved. If method binding fails, the `lookup-error` process is returned.

If σ is a state and e an expression, we denote by $[\![e]\!]_\sigma$ the result of reducing e in σ. We assume that the functional language of (side effect free) expressions is type sound, so well-typed expressions remain well-typed during evaluation, and that the reduction terminates. To simplify the presentation we omit the details of this standard reduction [39]. The enabledness of a guard in a given state σ with a given event queue $\overline{q}$ may be defined by induction over the

(R9) $(\mathit{msg}\ \textbf{to}\ o)\ \langle o \,|\, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle \ \longrightarrow\ \langle o \,|\, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\ \mathit{msg}\rangle$

(R10) $\langle \mathit{oid} : C \,|\, \mathit{PrQ} : \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\ \mathit{invoc}(m, \mathit{Sig}, \overline{e}, \langle o, \mathit{mid}\rangle)\rangle$
$\longrightarrow\ \langle \mathit{oid} : C \,|\, \mathit{PrQ} : (\overline{w}\ \mathit{lookup}(C, m, \mathit{Sig}, (\overline{e}, o, \mathit{mid}))), \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle$

(R11) $\langle o \,|\, \mathit{Pr} : \langle (t?(v);\ \overline{s}), \overline{l}\rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\ \mathit{comp}(\mathit{mid}, T, e)\rangle$
$\longrightarrow\ \langle o \,|\, \mathit{Pr} : \langle (v := e;\ \overline{s}), \overline{l}\rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle\ \textbf{if}\ \mathit{mid} = \llbracket t \rrbracket_{\overline{l}}$

(R12) $\langle o \,|\, \mathit{Pr} : \langle (t?(v);\ \overline{s}), \overline{l}\rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle$
$\longrightarrow\ \langle o \,|\, \mathit{Pr} : \texttt{deallocation-error}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle\ \textbf{if}\ \llbracket t \rrbracket_{\overline{l}} \in \mathcal{G}$

(R13) $\langle o | \mathit{Pr} : \langle \texttt{return}\ e;\ \overline{s}, \overline{l}\rangle, \mathit{Fld} : \overline{f}\rangle$
$\longrightarrow\ \langle o \,|\, \mathit{Pr} : \langle \overline{s}, \overline{l}\rangle, \mathit{Fld} : \overline{f}\rangle\ \mathit{comp}(\llbracket \gamma \rrbracket_{\overline{l}}, \llbracket \beta \rrbracket_{\overline{l}}, \llbracket e \rrbracket_{\overline{f}\cdot\overline{l}})\ \textbf{to}\ \llbracket \alpha \rrbracket_{\overline{l}}$

(R14) $\langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle (\overline{s}_1 \Box \overline{s}_2);\ \overline{s}_3, \overline{l}\rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle$
$\longrightarrow \langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \overline{s}_1;\ \overline{s}_3, \overline{l}\rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle\ \textbf{if}\ \mathit{ready}(\overline{s}_1, (\overline{f}\cdot\overline{l}), \overline{q})$

(R15) $\langle o \,|\, \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle (\texttt{free}(\overline{t});\ \overline{s}), \overline{l}\rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle$
$\longrightarrow \langle o \,|\, \mathit{Pr} : \langle \overline{s}, \overline{l}\rangle, \mathit{EvQ}_{(\{\llbracket \overline{t} \rrbracket_{\overline{f}\cdot\overline{l}}\} \cup \mathcal{L}, \mathcal{G})} : \overline{q}\rangle$

(R16) $\langle o \,|\, \mathit{EvQ}_{(\{\mathit{mid}\} \cup \mathcal{L}, \mathcal{G})} : \mathit{comp}(\mathit{mid}, T, e)\ \overline{q}\rangle \ \longrightarrow \langle o \,|\, \mathit{EvQ}_{(\mathcal{L}, \mathcal{G} \cup \{\mathit{mid}\})} : \overline{q}\rangle$

Figure 8.7: The operational semantics (2).

construction of guards, as follows:

$$\begin{aligned}
\mathit{enabled}(t?, \sigma, \overline{q}) &= \llbracket t \rrbracket_\sigma\ \mathit{in}\ \overline{q} \\
\mathit{enabled}(b, \sigma, \overline{q}) &= \llbracket b \rrbracket_\sigma \\
\mathit{enabled}(g \wedge g', \sigma, \overline{q}) &= \mathit{enabled}(g, \sigma, \overline{q}) \wedge \mathit{enabled}(g', \sigma, \overline{q})
\end{aligned}$$

Here, *in* checks whether a completion message corresponding to the given label value $\llbracket t \rrbracket_\sigma$ is in a message queue $\overline{q}$. The predicate *dealloc* determines if a guard dereferences a deallocated message in the set of label values S:

$$\begin{aligned}
\mathit{dealloc}(t?, \sigma, S) &= \llbracket t \rrbracket_\sigma \in S \\
\mathit{dealloc}(b, \sigma, S) &= \texttt{false} \\
\mathit{dealloc}(g \wedge g', \sigma, S) &= \mathit{dealloc}(g, \sigma, S) \vee \mathit{dealloc}(g', \sigma, S)
\end{aligned}$$

Enabledness is lifted to statement lists to express whether a process may be suspended or not. All atomic statements except `await` statements are enabled. However, in order to select a branch of a process, we may want to only choose a branch which is ready to execute without immediately blocking. This is expressed by the *ready* predicate. The two predicates are defined by induction over the construction of statement lists:

$$
\begin{aligned}
&\mathit{enabled}(s;\ \bar{s},\sigma,\overline{q}) = \mathit{enabled}(s,\sigma,\overline{q})\\
&\mathit{enabled}(\mathrm{await}\ g,\sigma,\overline{q}) = \mathit{enabled}(g,\sigma,\overline{q})\\
&\mathit{enabled}(\bar{s}\Box\bar{s}',\sigma,\overline{q}) = \mathit{enabled}(\bar{s},\sigma,\overline{q})\vee\mathit{enabled}(\bar{s}',\sigma,\overline{q})\\
&\mathit{enabled}(s,\sigma,\overline{q}) = \texttt{true}\quad \mathit{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ready}(\texttt{free}(\bar{l});\ \bar{s},\sigma,\overline{q}) = \mathit{ready}(\bar{s},\sigma,\overline{q})\\
&\mathit{ready}(s;\ \bar{s},\sigma,\overline{q}) = \mathit{ready}(s,\sigma,\overline{q})\\
&\mathit{ready}(t?(v),\sigma,\overline{q}) = \mathit{enabled}(\mathrm{await}\ t?,\sigma,\overline{q})\\
&\mathit{ready}(\bar{s}\Box\bar{s}',\sigma,\overline{q}) = \mathit{ready}(\bar{s},\sigma,\overline{q})\vee\mathit{ready}(\bar{s}',\sigma,\overline{q})\\
&\mathit{ready}(s,\sigma,\overline{q}) = \mathit{enabled}(s,\sigma,\overline{q})\quad \mathit{otherwise}
\end{aligned}
$$

Object activity is organized around a message queue *eventQ* containing a multiset of unprocessed incoming messages and a process queue *processQ* for suspended processes; i.e., remaining parts of method activations. There is exactly one active process. In the assignment rule R1, an expression is evaluated and bound to a program variable. In the object creation rule R2, an object with an unique identifier is constructed. The first execution step of the new object is to reassign to `this` its own identifier. Rules R3 and R5 for guards use the auxiliary *enabledness* function. If a guard evaluates to `true`, the execution of the process may continue in rule R3, otherwise a `release` is introduced in rule R5, given that the guard does not contain any read operation on a deallocated label. In rule R6 the active process is suspended on the process queue. If a guard contains read operations on a deallocated label value, a `deallocation-error` process is returned in rule R4. If the active process is `idle`, any suspended process may be activated in rule R7 if it is ready.

A message is emitted into the configuration by a method call in rule R8 where the in-parameter expressions are reduced to values in the local state of the sender. The label is translated into a label assignment where the value uniquely identifies the call. The message is delivered to the callee in rule R9 where *msg* ranges over invocations and completions. Message overtaking is captured by the non-determinism inherent in rewriting logic: messages sent by an object in one order may arrive in any order. In rule R10, the method invocation is bound to a process by the *lookup* function. The reply assignment in rule R11 fetches the return value from the completion message in the object's queue. As the message queue is a multiset, the rule will match any occurrence of $\mathit{comp}(\mathit{mid},T,e)$. If the completion message has already been deallocated, the `deallocation-error` process is returned in rule R12.

The return of a method call is captured by rule R13 where a uniquely labeled completion message is returned to the caller and the return expression is reduced to a return value in the local state of the sender. This rule makes use of the special local variables γ, α, and β which store information from the associated invocation message. Rule R14 deals with internal branching. The choice operator is associative and commutative, so R14 covers the selection of any branch in a compound nondeterministic choice statement. Here, the *ready* predicate provides a lazy branch selection. Rules R15 and R16 deal with the deallocation of messages from the message queue. The statement $\texttt{free}(\bar{t})$ expresses that messages with labels in $\bar{t}$ may be deallocated. In R15, these are added to the event queue's label list. In R16, a message with a label value in that list is deallocated. The deallocated label values are stored in the set $\mathcal{G}$.

Example 4 We consider an execution sequence of the wide-area network services of Example 2. In Example 3 we showed the runtime code generated by the type analysis for *newsRelay*. Let an object o create an instance of class `NewsService` and make an invocation to the new object $(1:\texttt{NewsService})$. Figure 8.8 shows the execution sequence of the method body *newsRelay*.

8.5 Typing of Runtime Configurations

In this section, we assume that runtime code is derived by type analysis from well-typed source programs. Consequently, a much simpler type system can be used for type analysis of runtime configurations than for the static analysis. In particular, we can rely on Theorems 1 and 2 for the correctness of the return types and of the placement of deallocation operations.

The initial runtime typing context Γ provides static information; i.e., types for the Booleans, `null`, and the fields and local variables of every class. Without loss of generality, we may assume that all classes, fields, and local variables have unique names. The context is gradually extended with types for identifiers of dynamically created objects and futures. The type system for runtime configurations is given in Figure 8.9. Runtime type judgments are on the form $\Gamma, \overline{q} \vdash C \ \texttt{ok}$ and express that a construct C is well-typed in Γ with a message queue $\overline{q}$. The message queue is used to type check reply assignments (in (REPLY1) and (REPLY2)). To simplify the presentation, $\overline{q}$ is omitted when not needed. The rules (MSG1) and (MSG2) are used to type check both messages in transit between objects and messages in the event queues. Type checking the process queue is similar to type checking the active process. Note that `lookup-error` and `deallocation-error` processes are not well-typed.

Definition 2 A runtime configuration *config* is well-typed in an environment Γ if $\Gamma \vdash \textit{config}\ \texttt{ok}$.

For a program $P = \overline{L}\,\{\overline{T\,x};\,\overline{s}\}$, if $\Gamma_0, \varepsilon \vdash P \triangleright \overline{L}'\ \{\overline{s}', \overline{x \mapsto \texttt{default}(T)}\}$, the *initial configuration* $\langle o \,|\, \mathit{Fld}: \varepsilon, \mathit{Pr}\colon \langle \overline{s}', \overline{x \mapsto \texttt{default}(T)}\rangle, \mathit{PrQ}\colon \emptyset, \mathit{EvQ}_{(\emptyset,\emptyset)}\colon \emptyset\rangle$ is obviously well-typed in Γ_0 extended with types for the program's variables. A *method activation*, with correctly typed actual parameters, of any method in a well-typed program results in a well-typed runtime process. It is assumed that side effect free functional expressions are *type sound* in well-typed configurations [39]; if an expression e is evaluated in an object state σ in a well-typed configuration with typing environment Γ and $\Gamma \vdash e : T$, then $\Gamma([\![e]\!]_\sigma) \preceq T$.

Example 5 To illustrate the typing of runtime configurations, consider the configuration derived from the initial configuration in Figure 8.8 by applying R2 and R1. By (PROCESS), processes in $(1:\texttt{NewsService})$ and $(1:\texttt{News})$ are well-typed. An empty *eventQ* is well-typed by (STATE). For fields, after evaluating the statement $\mathit{CNN} := \texttt{new News()}$ of the initial configuration, the substitution $\mathit{CNN} \mapsto (1:\texttt{News})$ is well-typed in the extended typing extended environment $\Gamma' = \Gamma + [(1:\texttt{News}) \mapsto \texttt{News}]$ by (STATE). Thus, the configuration is well-typed.

8.5.1 Subject Reduction

We establish a subject reduction property in the style of Wright and Felleisen [39]. This property ensures that runtime configurations remain well-typed and that the deallocation operations

$\langle(1:\texttt{NewsService})\,|\,\mathit{Fld}:(\mathit{CNN}\mapsto \mathit{null},\mathit{email}\mapsto \mathit{null}),\mathit{Pr}:\langle(\mathit{CNN}:=\texttt{new News}();$
$t!\mathit{CNN.news}(\texttt{Date}\to\texttt{XML},d);\texttt{await}\ t?;t?(v);\mathit{email}:=\texttt{new Email}();t!\mathit{email.send}(\texttt{XML}\times\texttt{Client}\to\texttt{Data},v,a);$
$\texttt{free}(t);\texttt{return true}),(d\mapsto(2007-12-31),a\mapsto \mathit{john.doe@email.com},v\mapsto\texttt{default(XML)},$
$t\mapsto\texttt{default(Label)},\gamma\mapsto 2,\alpha\mapsto(1:\texttt{o}),\beta\mapsto\texttt{Bool})\rangle,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$
$\longrightarrow$ R2, $\longrightarrow$ R1
$\langle(1:\texttt{NewsService})\,|\,\mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}),\mathit{email}\mapsto \mathit{null}),\mathit{Pr}:\langle(t!\mathit{CNN.news}(\texttt{Date}\to\texttt{XML},d);$
$\texttt{await}\ t?;t?(v);\mathit{email}:=\texttt{new Email}();t!\mathit{email.send}(\texttt{XML}\times\texttt{Client}\to\texttt{Data},v,a);\texttt{free}(t);\texttt{return true}),$
$(d\mapsto(2007-12-31),a\mapsto \mathit{john.doe@email.com},v\mapsto\texttt{default(XML)},t\mapsto\texttt{default(Label)},$
$\gamma\mapsto 2,\alpha\mapsto(1:\texttt{o}),\beta\mapsto\texttt{Bool})\rangle,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$
$\langle(1:\texttt{News})\,|\,\mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})],\mathit{Pr}:\varepsilon,\mathit{PrQ}:\varepsilon,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$
$\longrightarrow$ R8
$\langle(1:\texttt{NewsService})\,|\,\mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}),\mathit{email}\mapsto \mathit{null}),\mathit{Pr}:\langle(t:=2;\texttt{await}\ t?;t?(v);\mathit{email}:=\texttt{new Email}();$
$t!\mathit{email.send}(\texttt{XML}\times\texttt{Client}\to\texttt{Data},v,a);\texttt{free}(t);\texttt{return true}),(d\mapsto(2007-12-31),$
$a\mapsto \mathit{john.doe@email.com},v\mapsto\texttt{default(XML)},t\mapsto\texttt{default(Label)},\gamma\mapsto 2,\alpha\mapsto(1:\texttt{o}),\beta\mapsto\texttt{Bool})\rangle,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$
$\langle(1:\texttt{News})\,|\,\mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})],\mathit{Pr}:\varepsilon,\mathit{PrQ}:\varepsilon,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$
$\mathit{invoc}(\mathit{news},\texttt{Date}\to\texttt{XML},(2007-12-31),\langle(1:\texttt{NewsService}),2\rangle)$ **to** $(1:\texttt{News})$
$\longrightarrow$ R1, $\longrightarrow$ R9, $\longrightarrow$ R10, $\longrightarrow$ R7
$\langle(1:\texttt{NewsService})\,|\,\mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}),\mathit{email}\mapsto \mathit{null}),\mathit{Pr}:\langle(\texttt{await}\ t?;t?(v);\mathit{email}:=\texttt{new Email}();$
$t!\mathit{email.send}(\texttt{XML}\times\texttt{Client}\to\texttt{Data},v,a);\texttt{free}(t);\texttt{return true}),(d\mapsto(2007-12-31),$
$a\mapsto \mathit{john.doe@email.com},v\mapsto\texttt{default(XML)},t\mapsto 2,\gamma\mapsto 2,\alpha\mapsto(1:\texttt{o}),\beta\mapsto\texttt{Bool})\rangle,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$
$\langle(1:\texttt{News})\,|\,\mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})],$
$\mathit{Pr}:\langle\bar{s};\texttt{return}(\mathit{xml}),(\bar{l},\gamma\mapsto 2,\alpha\mapsto(1:\texttt{NewsService}),\beta\mapsto\texttt{XML})\rangle,\mathit{PrQ}:\varepsilon,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$
$\longrightarrow$ we omit the execution of $\bar{s}$, R13
$\langle(1:\texttt{NewsService})\,|\,\mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}),\mathit{email}\mapsto \mathit{null}),\mathit{Pr}:\langle(\texttt{await}\ t?;t?(v);\mathit{email}:=\texttt{new Email}();$
$t!\mathit{email.send}(\texttt{XML}\times\texttt{Client}\to\texttt{Data},v,a);\texttt{free}(t);\texttt{return true}),(d\mapsto(2007-12-31),$
$a\mapsto \mathit{john.doe@email.com},v\mapsto\texttt{default(XML)},t\mapsto 2,\gamma\mapsto 2,\alpha\mapsto(1:\texttt{o}),\beta\mapsto\texttt{Bool})\rangle,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$
$\langle(1:\texttt{News})\,|\,\mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})],\mathit{Pr}:\langle\varepsilon,(\bar{l},\gamma\mapsto 2,\alpha\mapsto(1:\texttt{NewsService}),\beta\mapsto\texttt{XML})\rangle,\mathit{PrQ}:\varepsilon,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$
$\mathit{comp}(2,\texttt{XML},[\![\mathit{xml}]\!]_{\bar{l}})$ **to** $(1:\texttt{NewsService})$
$\longrightarrow$ R9, $\longrightarrow$ R3 and let (...) be the value of $[\![\mathit{xml}]\!]_{\bar{l}}$
$\langle(1:\texttt{NewsService})\,|\,\mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}),\mathit{email}\mapsto \mathit{null}),\mathit{Pr}:\langle(t?(v);\mathit{email}:=\texttt{new Email}();$
$t!\mathit{email.send}(\texttt{XML}\times\texttt{Client}\to\texttt{Data},v,a);\texttt{free}(t);\texttt{return true}),(d\mapsto(2007-12-31),a\mapsto \mathit{john.doe@email.com},$
$v\mapsto\texttt{default(XML)},t\mapsto 2,\gamma\mapsto 2,\alpha\mapsto(1:\texttt{o}),\beta\mapsto\texttt{Bool})\rangle,\mathit{EvQ}_{\emptyset,\emptyset}:\mathit{comp}(2,\texttt{XML},(...))\rangle$
$\langle(1:\texttt{News})\,|\,\mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})],\mathit{Pr}:\langle\varepsilon,(\bar{l},\gamma\mapsto 2,\alpha\mapsto(1:\texttt{NewsService}),\beta\mapsto\texttt{XML})\rangle,\mathit{PrQ}:\varepsilon,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$
$\longrightarrow$ R11
$\langle(1:\texttt{NewsService})\,|\,\mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}),\mathit{email}\mapsto \mathit{null}),\mathit{Pr}:\langle(v:=(...);\mathit{email}:=\texttt{new Email}();$
$t!\mathit{email.send}(\texttt{XML}\times\texttt{Client}\to\texttt{Data},v,a);\texttt{free}(t);\texttt{return true}),(d\mapsto(2007-12-31),$
$a\mapsto \mathit{john.doe@email.com},v\mapsto\texttt{default(XML)},t\mapsto 2,\gamma\mapsto 2,\alpha\mapsto(1:\texttt{o}),\beta\mapsto\texttt{Bool})\rangle,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon)\rangle$
$\langle(1:\texttt{News})\,|\,\mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})],\mathit{Pr}:\langle\varepsilon,(\bar{l},\gamma\mapsto 2,\alpha\mapsto(1:\texttt{NewsService}),\beta\mapsto\texttt{XML})\rangle,\mathit{PrQ}:\varepsilon,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$
$\longrightarrow$ R1, $\longrightarrow$ R2, $\longrightarrow$ R1
$\langle(1:\texttt{NewsService})\,|\,\mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}),\mathit{email}\mapsto(1:\texttt{Email})),$
$\mathit{Pr}:\langle(t!\mathit{email.send}(\texttt{XML}\times\texttt{Client}\to\texttt{Data},v,a);\texttt{free}(t);\texttt{return true}),$
$(d\mapsto(2007-12-31),a\mapsto \mathit{john.doe@email.com},v\mapsto(...),t\mapsto 2,\gamma\mapsto 2,\alpha\mapsto(1:\texttt{o}),\beta\mapsto\texttt{Bool})\rangle,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon)\rangle$
$\langle(1:\texttt{News})\,|\,\mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})],\mathit{Pr}:\langle\varepsilon,(\bar{l},\gamma\mapsto 2,\alpha\mapsto(1:\texttt{NewsService}),\beta\mapsto\texttt{XML})\rangle,\mathit{PrQ}:\varepsilon,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$
$\langle(1:\texttt{Email})\,|\,\mathit{Fld}:\mathit{fields}(\texttt{Email})[\mathit{this}\mapsto(1:\texttt{Email})],\mathit{Pr}:\varepsilon,\mathit{PrQ}:\varepsilon,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$
$\longrightarrow$ R8, $\longrightarrow$ R1, $\longrightarrow$ R15, $\longrightarrow$ R9, $\longrightarrow$ R10, $\longrightarrow$ R7, $\longrightarrow$ R13, $\longrightarrow$ R9,
$\langle(1:\texttt{NewsService})\,|\,\mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}),\mathit{email}\mapsto(1:\texttt{Email})),\mathit{Pr}:\langle(\texttt{return true}),(d\mapsto(2007-12-31),$
$a\mapsto \mathit{john.doe@email.com},v\mapsto(...),t\mapsto 3,\gamma\mapsto 2,\alpha\mapsto(1:\texttt{o}),\beta\mapsto\texttt{Bool})\rangle,\mathit{EvQ}_{3,\emptyset}:\mathit{comp}(3,\texttt{Bool},\mathit{true})\rangle$
$\langle(1:\texttt{News})\,|\,\mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})],\mathit{Pr}:\langle\varepsilon,(\bar{l},\gamma\mapsto 2,\alpha\mapsto(1:\texttt{NewsService}),\beta\mapsto\texttt{XML})\rangle,\mathit{PrQ}:\varepsilon,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$
$\langle(1:\texttt{Email})\,|\,\mathit{Fld}:\mathit{fields}(\texttt{Email})[\mathit{this}\mapsto(1:\texttt{Email})],\mathit{Pr}:\langle\varepsilon,\bar{l}\rangle,\mathit{PrQ}:\varepsilon,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$
$\longrightarrow$ R16, $\longrightarrow$ R13
$\langle(1:\texttt{NewsService})\,|\,\mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}),\mathit{email}\mapsto(1:\texttt{Email})),\mathit{Pr}:\langle\varepsilon,(d\mapsto(2007-12-31),$
$a\mapsto \mathit{john.doe@email.com},v\mapsto(...),t\mapsto 3,\gamma\mapsto 2,\alpha\mapsto(1:\texttt{o}),\beta\mapsto\texttt{Bool})\rangle,\mathit{EvQ}_{\emptyset,3}:\varepsilon\rangle$
$\langle(1:\texttt{News})\,|\,\mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})],\mathit{Pr}:\langle\varepsilon,(\bar{l},\gamma\mapsto 2,\alpha\mapsto(1:\texttt{NewsService}),\beta\mapsto\texttt{XML})\rangle,\mathit{PrQ}:\varepsilon,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$
$\langle(1:\texttt{Email})\,|\,\mathit{Fld}:\mathit{fields}(\texttt{Email})[\mathit{this}\mapsto(1:\texttt{Email})],\mathit{Pr}:\langle\varepsilon,\bar{l}\rangle,\mathit{PrQ}:\varepsilon,\mathit{EvQ}_{\emptyset,\emptyset}:\varepsilon\rangle$ $\mathit{comp}(2,\texttt{Bool},\mathit{true})$ **to** $(1:\texttt{o})$

Figure 8.8: The execution sequence of method *newsRelay*

$$\text{(Assign)}\ \frac{\Gamma \vdash e : T \quad T \preceq \Gamma(v)}{\Gamma \vdash v := e\ \texttt{ok}} \qquad \text{(New)}\ \frac{C \preceq \Gamma(v)}{\Gamma \vdash v := \texttt{new}\ C()\ \texttt{ok}} \qquad \text{(Skip)}\ \frac{}{\Gamma \vdash \texttt{skip}\ \texttt{ok}}$$

$$\text{(Await)}\ \frac{\Gamma \vdash g\ \texttt{ok}}{\Gamma \vdash \texttt{await}\ g\ \texttt{ok}} \qquad \text{(Release)}\ \frac{}{\Gamma \vdash \texttt{release}\ \texttt{ok}} \qquad \text{(free)}\ \frac{\forall\, t \in \bar{t} \cdot \Gamma(t) = \mathsf{Label}}{\Gamma \vdash \texttt{free}(\bar{t})\ \texttt{ok}}$$

$$\text{($\wedge$-Guards)}\ \frac{\Gamma \vdash \bar{s}_1\ \texttt{ok} \quad \Gamma \vdash \bar{s}_1\ \texttt{ok}}{\Gamma \vdash \bar{s}_1 \wedge \bar{s}_2\ \texttt{ok}} \qquad \text{(B-Guards)}\ \frac{\Gamma \vdash b : \texttt{Bool}}{\Gamma \vdash b\ \texttt{ok}} \qquad \text{(t-Guards)}\ \frac{\Gamma(t) = \mathsf{Label}}{\Gamma \vdash t?\ \texttt{ok}}$$

$$\text{(Calls)}\ \frac{\Gamma \vdash e : C \quad \Gamma \vdash \bar{e} : T_1 \quad T_1 \preceq T \quad \mathit{lookup}(C, m, T \rightarrow T')}{\Gamma \vdash\ t!e.m(T \rightarrow T', \bar{e})\ \texttt{ok}}$$

$$\text{(Reply1)}\ \frac{\mathit{comp}(t, T, e) \in \bar{q} \quad T \preceq \Gamma(v)}{\Gamma, \bar{q}\ \vdash t?(v)\ \texttt{ok}} \qquad \text{(Choice)}\ \frac{\Gamma, \bar{q} \vdash \bar{s}_1\ \texttt{ok} \quad \Gamma, \bar{q} \vdash \bar{s}_2\ \texttt{ok}}{\Gamma, \bar{q} \vdash \bar{s}_1 \Box \bar{s}_2\ \texttt{ok}}$$

$$\text{(Reply2)}\ \frac{\neg \exists\, e, T \cdot \mathit{comp}(t, T, e) \in \bar{q} \quad T \preceq \Gamma(v)}{\Gamma, \bar{q}\ \vdash t?(v)\ \texttt{ok}} \qquad \text{(Seq)}\ \frac{\Gamma, \bar{q} \vdash \bar{s}_1\ \texttt{ok} \quad \Gamma, \bar{q} \vdash \bar{s}_2\ \texttt{ok}}{\Gamma, \bar{q} \vdash \bar{s}_1;\ \bar{s}_2\ \texttt{ok}}$$

$$\text{(Msg1)}\ \frac{\Gamma \vdash \bar{e} : T \quad T \preceq T_1}{\Gamma \vdash \mathit{invoc}(m, T_1 \rightarrow\ T_2, \bar{e}, \langle o, n \rangle)\ \texttt{ok}} \qquad \text{(Msg2)}\ \frac{\Gamma \vdash e : T \quad T \preceq\ T'}{\Gamma \vdash \mathit{comp}(n, T', e)\ \texttt{ok}}$$

$$\text{(Msg3)}\ \frac{\Gamma \vdash \mathit{msg}\ \texttt{ok}}{\Gamma \vdash \mathit{msg}\ \mathbf{to}\ o\ \texttt{ok}} \qquad \text{(EventQ)}\ \frac{\Gamma \vdash \mathit{eventQ}_1\ \texttt{ok} \quad \Gamma \vdash \mathit{eventQ}_2\ \texttt{ok}}{\Gamma \vdash \mathit{eventQ}_1\ \mathit{eventQ}_2\ \texttt{ok}}$$

$$\text{(Process)}\ \frac{\begin{array}{c}\Gamma, \bar{q} \vdash \bar{s}\ \texttt{ok} \quad \Gamma \vdash \mathit{sub}\ \texttt{ok} \\ \Gamma \vdash \bar{e} : T \quad T \preceq \mathit{sub}(\beta)\end{array}}{\Gamma, \bar{q} \vdash\ \langle \bar{s};\ \texttt{return}\ \bar{e}, \mathit{sub} \rangle\ \texttt{ok}} \qquad \text{(State)}\ \frac{\texttt{for all}\ (v \mapsto \mathit{val}) \cdot \Gamma(\mathit{val}) \preceq \Gamma(v)}{\Gamma \vdash\ \overline{v \mapsto \mathit{val}}\ \texttt{ok}}$$

$$\text{(ProcessQ)}\ \frac{\Gamma, \bar{q} \vdash \mathit{processQ}_1\ \texttt{ok} \quad \Gamma, \bar{q} \vdash \mathit{processQ}_2\ \texttt{ok}}{\Gamma, \bar{q} \vdash \mathit{processQ}_1\ \mathit{processQ}_2\ \texttt{ok}} \qquad \text{(IdleProcess)}\ \frac{}{\Gamma \vdash \texttt{idle}\ \texttt{ok}}$$

$$\text{(Object)}\ \frac{\begin{array}{c}\Gamma \vdash \mathit{sub}\ \texttt{ok} \qquad \Gamma, \mathit{eventQ} \vdash \mathit{active}\ \texttt{ok} \qquad \Gamma, \mathit{eventQ} \vdash \mathit{processQ}\ \texttt{ok} \\ \texttt{for all}\ t \cdot \Gamma(t) = \texttt{Label} \qquad \Gamma \vdash \mathit{eventQ}\ \texttt{ok}\end{array}}{\Gamma \vdash \langle o \mid \mathit{Fld} : \mathit{sub}, \mathit{Pr} \colon \mathit{active}, \mathit{PrQ} \colon \mathit{processQ}, \mathit{EvQ}_{\bar{t}} \colon \mathit{eventQ} \rangle\ \texttt{ok}}$$

$$\text{(Empty)}\ \frac{}{\Gamma \vdash \varepsilon\ \texttt{ok}} \qquad \text{(Config)}\ \frac{\Gamma \vdash \mathit{config}_1\ \texttt{ok} \qquad \Gamma \vdash \mathit{config}_2\ \texttt{ok}}{\Gamma \vdash \mathit{config}_1\ \mathit{config}_2\ \texttt{ok}}$$

Figure 8.9: The type system for runtime configurations.

inserted in the code are safe. The dynamic aspect of well-typed runtime configurations is due to the late binding of asynchronous method calls. Technically, this is represented as the absence of the `lookup-error` process in the runtime configurations. Deallocation operations are unsafe if they occur too early; i.e., if method returns are deallocated at a program point p then there should not be any read operations on those returns in the execution paths after p. Similar to method calls, safe deallocation is represented as the absence of the `deallocation-error` process.

Theorem 3 (Subject reduction) *Let config be an initial runtime configuration of a well-typed program such that* $\Gamma \vdash \mathit{config}\ \texttt{ok}$*. If* $\mathit{config} \rightarrow \mathit{config}'$*, then there exists a* $\Gamma' \supseteq \Gamma$ *such that* $\Gamma' \vdash \mathit{config}'\ \texttt{ok}$*.*

For well-typed programs, subject reduction leads to the following properties:

Corollary 3 *Let config be the initial configuration of a well-typed program. If config $\rightarrow$ config$'$, then config$'$ does not contain* `lookup-error`.

Corollary 4 *Let config be the initial configuration of a well-typed program. If config $\rightarrow$ config$'$, then config$'$ does not contain* `deallocation-error`.

8.6 Related work

For a detailed comparison of Creol with other distributed object languages here, see [19, 20]. *Asynchronous method calls* as discussed in this paper may be compared to message exchange in languages based on the Actor model [1]. Actor languages are conceptually attractive for distributed programming because they base communication on asynchronous messages, focusing on loosely coupled processes. An actor encapsulates its fields, procedures that manipulate the state, and a single thread of control. Creol objects are similar to actors, except that our methods return values managed by futures, and control can be released at specific points during a method execution. Messages to actors return no result and run to completion before another message is handled. The lack of return makes programming directly with actors cumbersome [1].

Futures express concurrency in a simple manner, reducing the latency dependency by enabling synchronization by necessity. Futures were discovered by Baker and Hewitt [3], and used by Halstead in MultiLisp [13] and by Liskov and Shrira as Promises [23]. Futures now appear in languages like Alice [30], Oz-Mozart [33], Concurrent ML [29], C++ [22], Java [38, 18], and Polyphonic C$^\sharp$ [4], often as libraries. Futures in these languages resemble labels in Creol. Implementations associate a future with the asynchronous execution of an expression in a new thread. The future is a placeholder which is immediately returned to the caller. From the perspective of the caller, this placeholder is a read-only structure [26]. However, the placeholder can also be explicitly manipulated to write data. In some approaches, the placeholder can be accessed in both modes (e.g., CML, Alice, Java, C++), though typically the caller and callee interfaces are distinct, as formalized by the calculus $\lambda(\mathsf{fut})$ [26]. Programming explicitly with promises is quite low-level, so Creol identifies the write operation on the future with method call return. This way, programming with future variables corresponds to programming with Creol's asynchronous method calls without processor release points; in fact, Creol's processor release points extend the computation flexibility originally motivating futures.

In this paper, we follow the approach of Promises [23] and use futures only inside the scope of a method body. This restriction is enforced by our type system (e.g., $\mathrm{ran}(\Delta) = \perp$ in (METHOD)) and leads to a strict state encapsulation in objects. However futures can also be shared. In this case the futures become global variables, which breaks with object encapsulation but facilitates the delegation of method returns to other objects. Futures can be transparent or non-transparent. Transparent futures cannot be explicitly manipulated, the type of the future is the same as the expected result, and futures are accessed as ordinary program variables, possibly after waiting (e.g., in Multilisp). Flanagan and Felleisen present semantic models of futures in terms of an abstract machine [11]. In contrast to Creol, their language is purely functional. Caromel, Henrio, and Serpett present an imperative, asynchronous object calculus with transparent futures [6]. Their active objects may have internal passive objects which can be passed between the active objects by so-called "sheep" copying. This feature is orthogonal to the issues discussed in this paper.

Non-transparent futures have a separate type to denote the future, and future objects can be explicitly manipulated (e.g., in CML, Alice, Java, C++, and Creol). If the future is shared, its type includes the type of its value (e.g., `future T` is a future of type `T`). However, this limits the *reusability* of the future variable. A formalization and proof system for Creol's concurrency model combined with shared non-transparent futures is presented in [8]. The present paper investigates a different approach, where the futures are encapsulated within the calling method but future variables can be shared between futures of different types, and shows how type analysis can be used to infer the underlying types such that method calls are bound correctly.

Type and effect systems add context information to type analyses [31, 2, 14] and have been used to, for example, ensure that guards controlling method availability do not have side effects [9], estimate the effects of an object reclassification [10], and ensure security properties. Gordon and Jeffrey use effects to track correspondences for authenticity properties of cryptographic protocols in the spi-calculus, by tracking matching begin- and end-assertions [12]. Broberg and Sands use effects to track different flow locks for dynamic flow policies by open and close operations for the different locks [5].

The decoupling of Creol's invocation and reply assignments results in a scoped use of labels. Part of the complexity of our type system stems from the lack of explicit scoping. Scoping is implicitly given by the operations on a label, which may cause memory leakage. For a given label, an invocation need not be succeeded by a reply although a reply must be preceded by an invocation. Thus, the analysis formalized in our type system corresponds to a combination of several static analysis techniques: scope detection, type inference for labels, live variable analysis [27], and linearity analysis [35]. Linear type systems guarantee that certain values are used exactly once at runtime, and have been used for resource analysis [35, 16], including file handling and memory management. Type systems have also been used to impose linearity conditions and I/O usage for channels in the π-calculus [15, 28]. In particular, Igarashi and Kobayashi develop a type system and inference algorithm to identify linear use of channels inside explicitly given scopes [15]. They exploit linearity information in compile-time optimizations to reduce the number of dynamically created processes, and to refine the runtime representation of linear channels and operations on these. Whereas labels in Creol always have linearity restrictions, channels may but need not be linear, which adds complexity. The authors identify reclamation of memory as an important application of linearity-based optimization, but leave specific solutions for future work.

The effect system used to infer return types for method calls contains sufficient information to automatically insert `free` operations which enable the deallocation of method replies. This usage of type and effect systems is related to the region-based approach to memory management proposed by Tofte and Talpin [32]. In their work, programs written in a kernel language extracted from SML are transformed by means of region inference analysis into a target program with annotated allocation and deallocation operations. The inference rules use an effect system to capture operations on regions. However the language does not consider branching structures. In Creol deallocation operations are inserted locally for each execution branch. This allows memory leaks to be restricted when the reply to method calls is only needed in certain execution branches. Explicit deallocation is attractive for memory management because it can be implemented using simple constant-time routines [36]. The automatic insertion of deallocation statements, also found in [32, 36], lets the programmer ignore low-level memory management issues, for which it is easy to make mistakes leading to memory leaks. Thus the local deallocation operations automatically inserted for method replies are much simpler than the tracing garbage collector proposed by Baker and Hewitt [3] in order to avoid memory leakage due to redundant futures.

Backwards analysis facilitates an *eager deallocation strategy* which allows the deallocation of method returns in an execution dependent manner even if the return is still accessible from memory. This seems desirable in the case of, e.g., tail-recursive calls. Linear types have been proposed as a basis for eager deallocation [16], as linear values can be garbage collected immediately after being used even if they are still referenced, improving on a tracing garbage collector. This approach is incorporated directly in our operational semantics, because rule R9 consumes the method return (in contrast to reply guards in rule R3). With shared futures, this could no longer be done directly. To the best of the authors' knowledge, an explicit deallocation strategy for shared futures is an open issue. In future work, we plan to extend the approach of this paper to Creol with shared futures [8]. The deallocation of shared method replies seems to require more overhead in the operational semantics, as the deallocation depends on the execution in all objects which might access the future. In future work, we further plan to investigate how far type-based insertion of deallocation operations extends to deallocate the active objects themselves, thus avoiding distributed garbage collectors for the active objects [37, 34].

8.7 Conclusion

This paper presents a kernel language for distributed concurrent objects communicating by asynchronous method calls. The approach emphasizes flexibility with respect to the possible delays and instabilities of distributed computing, and allows active and reactive behavior to be combined in an object. Asynchronous method calls and process release points add flexibility to execution in the distributed setting because waiting activities may yield processor control to suspended and enabled processes. A type and effect system for backwards analysis of programs is introduced, using effects to track information about method calls. A particular effect of our system is the translation of source programs into runtime code. The effects of the type analysis are explicitly used to expand each asynchronous call statement with a type-correct signature, ensuring a type-correct runtime method lookup. With the proposed type and effect system, this can be done directly in the analysis of the method invocation. Thus, the type analysis and translation to runtime code is done in one pass.

The analysis is exploited to avoiding memory leakage for passively stored method replies at runtime. The lifetime of label values is statically determined and deallocation operations are inserted by the type system, so no runtime traversal of labels is needed to locate redundant method replies. By exploiting effects in a backwards manner, we derive a fine-grained deallocation strategy. Deallocation operations are inserted in the specific execution branches where the method reply is redundant, even if the reply is reachable from the object state. The backwards analysis enables an eager strategy in the sense that deallocation instructions are inserted as early as possible in each branch after the method invocation. An advantage of explicit deallocation is that deallocation can be handled by a simple routine. Another advantage of the approach of this paper is that the insertion of these operations is incorporated in the type analysis. In fact, the correctness of the deallocation operations is asserted from the type system; i.e., a deallocation is safe if the method reply can no longer be accessed from the code. A subject reduction property is established for the language: method binding is guaranteed to succeed at runtime, deallocation operations are safe in all execution paths, and the redundant method replies of the different execution branches are deallocated.

References

[1] G. A. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In E. Najm and J.-B. Stefani, editors, *Proc. 1st IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'96)*, pages 135–153, Paris, 1996. Chapman & Hall.

[2] T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.

[3] H. G. Baker and C. E. Hewitt. The incremental garbage collection of processes. *ACM SIGPLAN Notices*, 12(8):55–59, 1977.

[4] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C$^\sharp$. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, Sept. 2004.

[5] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Proc. 15th European Symposium on Programming (ESOP'06)*, volume 3924 of *Lecture Notes in Computer Science*, pages 180–196. Springer-Verlag, 2006.

[6] D. Caromel, L. Henrio, and B. Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL'04)*, pages 123–134. ACM Press, 2004.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.

[8] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, Mar. 2007.

[9] P. Di Blasio and K. Fisher. A calculus for concurrent objects. In U. Montanari and V. Sassone, editors, *7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 655–670. Springer-Verlag, Aug. 1996.

[10] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: Fickle$_{\text{II}}$. *ACM Transactions on Programming Languages and Systems*, 24(2):153–191, 2002.

[11] C. Flanagan and M. Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999.

[12] A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations (CSFW'01)*, pages 145–159. IEEE Computer Society Press, 2001.

[13] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[14] F. Henglein, H. Makholm, and H. Niss. Effect types and region-based memory management. chapter 3, pages 87–135.

[15] A. Igarashi and N. Kobayashi. Type reconstruction for linear π-calculus with I/O subtyping. *Information and Computation*, 161(1):1–44, 2000.

[16] A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2):264–313, Mar. 2005.

[17] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[18] G. S. Itzstein and M. Jasiunas. On implementing high level concurrency in Java. In A. Omondi and S. Sedukhin, editors, *Proc. 8th Asia-Pacific Computer Systems Architecture Conference (ACSAC 2003)*, volume 2823 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, 2003.

[19] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[20] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.

[21] R. E. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley, 1996.

[22] R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. In J. O. Coplien, J. Vlissides, and N. Kerth, editors, *Proc. Pattern Languages of Program Design*, pages 483–499. Addison-Wesley, 1996.

[23] B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In D. S. Wise, editor, *Proceedings of the SIGPLAN Conference on Programming Lanugage Design and Implementation (PLDI'88)*, pages 260–267, Atlanta, GE, USA, June 1988. ACM Press.

[24] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[25] J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Software and Systems Modeling*, 6(1):83–110, Mar. 2007.

[26] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364:338–356, 2006.

[27] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, Germany, 2 edition, 2005.

[28] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.

[29] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[30] A. Rossberg, D. L. Botlan, G. Tack, T. Brunklaus, and G. Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming*, pages 79–96. Intellect Books, Bristol, UK, Feb. 2006.

[31] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[32] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[33] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Mar. 2004.

[34] N. Venkatasubramanian, G. Agha, and C. L. Talcott. Scalable distributed garbage collection for systems of active objects. In Y. Bekkers and J. Cohen, editors, *International Workshop on Memory Management (IWMM'92)*, volume 637 of *Lecture Notes in Computer Science*, pages 134–147. Springer-Verlag, 1992.

[35] D. Walker. Substructural type systems. chapter 1, pages 1–43.

[36] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.

[37] W.-J. Wang and C. A. Varela. Distributed garbage collection for mobile actor systems: The pseudo root approach. In Y.-C. Chung and J. E. Moreira, editors, *Advances in Grid and Pervasive Computing (GPC'06)*, volume 3947 of *Lecture Notes in Computer Science*, pages 360–372. Springer-Verlag, 2006.

[38] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. Object oriented programming, systems, languages, and applications (OOPSLA'05)*, pages 439–453, New York, NY, USA, 2005. ACM Press.

[39] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[40] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System*. Series in Computer Systems. The MIT Press, 1990.

8.A Proofs

8.A.1 Proof of Theorem 1

The proof is by induction on the derivation of $\Gamma, \varepsilon \vdash \overline{s} \rhd \overline{s}', \Delta'$. For the base case, $\Gamma, \varepsilon \vdash \texttt{skip} \rhd \texttt{skip}$. By definition, $\mathit{live}(t, \varepsilon) = \texttt{false}$ and $\varepsilon(t) = \perp$. For the induction step, the induction hypothesis (IH) is that $\neg\mathit{live}(t, \overline{s}_0) \iff (\Delta)(t) = \perp$ holds for $\Gamma, \varepsilon \vdash \overline{s}_0 \rhd \overline{s}_0', \Delta$ and consider $\Gamma, \Delta \vdash s \rhd \overline{s}', \Delta'$ by case analysis on s.

Cases (Assign), (Skip), *and* (New) follow directly from IH.

Case (Await). We have $s = \texttt{await}\ g$. There are two subcases for every label t, depending on whether t is contained in g:

1. *Guard g contains t?.* From Definition 1 we have $\mathit{live}(t, s; \overline{s}_0)$. Rule ($t$-Guards) results in the update $[t \mapsto \Delta(t) \cap \texttt{Data}]$ which propagates to the update Δ' for the `await` statement in (Await). Consequently, $(\Delta \cdot \Delta')(t) \neq \perp$.

2. *Guard g does not contain a reply guard on t.* From Definition 1 we get $\mathit{live}(t, s; \overline{s}_0) = \mathit{live}(t, \overline{s}_0)$. From the type analysis of g we get $t \notin \mathrm{dom}(\Delta')$, and it follows by (Await) that $(\Delta \cdot \Delta')(t) = \Delta(t)$. Consequently, $\neg\mathit{live}(t, s; \overline{s}_0) \iff (\Delta \cdot \Delta')(t) = \perp$ follows from IH.

It follows that $\forall l \colon \mathsf{Label} \cdot \neg\mathit{live}(t, \texttt{await}\ g; \overline{s}_0) \iff (\Delta \cdot \Delta')(t) = \perp$.

Case (Reply). From Definition 1 we have $\mathit{live}(t, t?(v); \overline{s}_0)$. The effect of (Reply) is $[t \mapsto T]$ for some type T, so $(\Delta \cdot \Delta')(t) \neq \perp$. For other labels t', $\neg\mathit{live}(t', s; \overline{s}_0) \iff (\Delta \cdot \Delta')(t') = \perp$ follows directly from IH.

Case (Call1). From Definition 1, $\neg\mathit{live}(t, t!o.m(\overline{e}); \overline{s}_0)$. From (Call1), $\Delta'(t) = \perp$. For other labels t', $\neg\mathit{live}(t', t!o.m(\overline{e}); \overline{s}_0) \iff (\Delta \cdot \Delta')(t') = \perp$ follows from IH.

Case (Call2). Similar to (Call1).

Case (Choice). We assume $\mathit{well\text{-}defined}((\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2))$ and that IH holds for the branches $\overline{s}_1$ (IH1) and $\overline{s}_2$ (IH2). From Definition 1 we have $\mathit{live}(t, \overline{s}_1 \Box \overline{s}_2; \overline{s}_0) = \mathit{live}(t, \overline{s}_1; \overline{s}_0) \vee \mathit{live}(t, \overline{s}_2; \overline{s}_0)$. Now assume that $\mathit{live}(t, \overline{s}_1 \Box \overline{s}_2; \overline{s}_0)$. There are three subcases for every label t:

1. $\mathit{live}(t, \overline{s}_1; \overline{s}_0)$ and $\neg\mathit{live}(t, \overline{s}_2; \overline{s}_0)$. In this subcase, we get $(\Delta \cdot \Delta_1)(t) \neq \perp$ by IH1 and $(\Delta \cdot \Delta_2)(t) = \perp$ by IH2. It follows that $((\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2))(t) \neq \perp$.

2. $\neg\mathit{live}(t, \overline{s}_1; \overline{s}_0)$ and $\mathit{live}(t, \overline{s}_2; \overline{s}_0)$. Similar to Subcase 1.

3. $\mathit{live}(t, \overline{s}_1; \overline{s}_0)$ and $\mathit{live}(t, \overline{s}_2; \overline{s}_0)$. In this subcase, we get $(\Delta \cdot \Delta_1)(t) \neq \perp$ by IH1 and $(\Delta \cdot \Delta_2)(t) \neq \perp$ by IH2. Since $\mathit{well\text{-}defined}((\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2))$, there must be some type T where $T \neq \perp$ and $T \neq \texttt{Error}$ such that $((\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2))(t) = T$.

It follows that $\forall l \colon \mathsf{Label} \cdot \neg\mathit{live}(t, \overline{s}_1 \Box \overline{s}_2; \overline{s}_0) \iff (\Delta \cdot ((\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2)))(t) = \perp$.

Case (Seq) follows by induction over the length of s. ■

8.A.2 Proof of Theorem 2

To make the link between method invocations and possible matching reply assignments explicit we introduce a notion of *reachable* replies, inspired by static analysis techniques [27]. Intuitively, a reply assignment $t?(v)$ is reachable in a statement list $\overline{s}$ if there exists an execution path through $\overline{s}$ in which $t?(v)$ is encountered before a method invocation with label t.

Definition 3 The set $\mathcal{R}(\overline{s})$ of reachable reply assignments in $\overline{s}$ is defined as

$$\begin{array}{l}\mathcal{R}(\varepsilon) = \emptyset \\ \mathcal{R}(s;\overline{s}) = \mathcal{R}(\overline{s}) \quad \text{if } s \in \{v := e, v := \texttt{new}\, C, \texttt{await}\ g, \texttt{skip}\} \\ \mathcal{R}(t?(v);\overline{s}) = \{t?(v)\} \cup \mathcal{R}(\overline{s}) \\ \mathcal{R}(t!o.m(\overline{e});\overline{s}) = \{t'?(v) \in \mathcal{R}(\overline{s}) \mid t \neq t'\} \\ \mathcal{R}(\overline{s}_1 \Box \overline{s}_2;\overline{s}) = \mathcal{R}(\overline{s}_1;\overline{s}) \cup \mathcal{R}(\overline{s}_2;\overline{s})\end{array}$$

A reply statement $t?(v)$ *corresponds* to an invocation $t!o.m(\overline{e})$ if $t?(v)$ is reachable at the program point after $t!o.m(\overline{e})$. The definition of reachable reply assignments resembles the definition of live variables; in fact, if $\Gamma, \varepsilon \vdash \overline{s} \rhd \overline{s}', \Delta'$ then $t?(v) \in \mathcal{R}(\overline{s}) \Rightarrow \mathit{live}(t, \overline{s})$. The implication only goes one way because reply guards also affect whether a label is live. We now establish a lemma which is used in the proof of Theorem 2.

Lemma 4 *Let $\Gamma, \varepsilon \vdash \overline{s} \rhd \overline{s}', \Delta$ be well-typed. Then $\forall t?(v) \in \mathcal{R}(\overline{s}) \cdot \Delta(t) \preceq \Gamma(v)$.*

Proof of Lemma 4. By induction on the derivation of $\Gamma, \varepsilon \vdash \overline{s} \rhd \overline{s}', \Delta$. For the base case, $\mathcal{R}(\varepsilon) = \emptyset$. For the induction step, assume as induction hypothesis (IH) that $\forall t?(v) \in \mathcal{R}(\overline{s}_0) \cdot \Delta'(t) \preceq \Gamma(v)$ for $\Gamma, \varepsilon \vdash \overline{s}_0 \rhd \overline{s}'_0, \Delta'$. Consider $\forall t?(v) \in \mathcal{R}(s;\overline{s}_0) \cdot \Delta(t) \preceq \Gamma(v)$ for $\Gamma, \varepsilon \vdash s;\overline{s}_0 \rhd s';\overline{s}'_0, \Delta$ by case analysis over s.

Cases (Assign), (Skip), (New), *and* (Await) follow directly from IH.

Case (Reply). From Definition 3, $\mathcal{R}(t?(v);\overline{s}_0) = \{t?(v)\} \cup \mathcal{R}(\overline{s}_0)$. From (Reply), $\Delta(t) = \Gamma(v)$ and by IH we get $\forall t'?(v') \in \mathcal{R}(t'(v);\overline{s}_0) \cdot \Delta(t') \preceq \Gamma(v')$.

Case (Call1). From Definition 3, $\mathcal{R}(t!o.m(\overline{e});\overline{s}) = \{t'?(v) \in \mathcal{R}(\overline{s}) \mid t \neq t'\}$. Hence, the case follows directly from IH.

Case (Call2). Similar to (Call1).

Case (Choice). We assume that *well-defined*$((\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2))$ and that IH holds for the branches $\overline{s}_1$ (IH1) and $\overline{s}_2$ (IH2). From Definition 3, we have $\mathcal{R}(\overline{s}_1 \Box \overline{s}_2;\overline{s}_0) = \mathcal{R}(\overline{s}_1;\overline{s}_0) \cup \mathcal{R}(\overline{s}_2;\overline{s}_0)$. There are three cases for a label t:

1. $t?(v) \in \mathcal{R}(\overline{s}_1;\overline{s}_0)$ and $t?(v') \notin \mathcal{R}(\overline{s}_2;\overline{s}_0)$. We have $((\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2))(t) = (\Delta_0 \cdot \Delta_1)(t)$. By IH1, $(\Delta_0 \cdot \Delta_1)(t) \preceq \Gamma(v)$.

2. $t?(v) \notin \mathcal{R}(\overline{s}_1;\overline{s}_0)$ and $t?(v') \in \mathcal{R}(\overline{s}_2;\overline{s}_0)$. We have $((\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2))(t) = (\Delta_0 \cdot \Delta_2)(t)$. By IH2, $(\Delta_0 \cdot \Delta_2)(t) \preceq \Gamma(v)$.

3. $t?(v) \in \mathcal{R}(\overline{s}_1;\overline{s}_0)$ and $t?(v') \in \mathcal{R}(\overline{s}_2;\overline{s}_0)$. We have $((\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2))(t) = (\Delta_0 \cdot \Delta_1)(t) \cap (\Delta_0 \cdot \Delta_2)(t)$. Moreover we have that $(\Delta_0 \cdot \Delta_1)(t) \neq \bot$, $(\Delta_0 \cdot \Delta_2)(t) \neq \bot$, $(\Delta_0 \cdot \Delta_1)(t) \neq \texttt{Error}$, and $(\Delta_0 \cdot \Delta_2)(t) \neq \texttt{Error}$. Since *well-defined*$((\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2))$, we know that $((\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2))(t) \preceq (\Delta_0 \cdot \Delta_1)(t)$ and $((\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2))(t) \preceq (\Delta_0 \cdot \Delta_2)(t)$. By IH1 and IH2, we then obtain $((\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2))(t) \preceq \Gamma(v)$ and $((\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2))(t) \preceq \Gamma(v')$

It follows that $\forall t?(v) \in \mathcal{R}(\overline{s}_1 \Box \overline{s}_2;\overline{s}) \cdot \Delta(t) \preceq \Gamma(v)$.

Case (Seq) follows by induction over the length of s. ■

Proof of Theorem 2. There are two cases, depending on the rule used to derive a signature for a method invocation. For (CALL1), $\Delta(t) \neq \bot$. It follows from Lemma 4 that any reachable reply assignment $t?(v)$ is such that $\Delta(t) \preceq \Gamma(v)$. For (CALL2), $\Delta(t) = \bot$. By Theorem 1, t is not live after the method invocation. Hence, there are no reachable reply assignments. ■

8.A.3 Proof of Theorem 3

Consider a well-typed program $P = \overline{L}\ \{\overline{T\ x}, \overline{s}\}$ and assume that the judgment $\Gamma_0, \varepsilon \vdash P \rhd \overline{L'}\ \{\overline{s}', \overline{x \mapsto \texttt{default}(T)}\}$ holds. The proof is by induction over the length of an execution $\mathit{config}_0, \mathit{config}_1, \ldots$. Let Γ_1 extend Γ_0 with types for variables declared in P (for simplicity assuming uniqueness of names). For the *base case*, the initial runtime configuration config_0 is well-typed:

$$\Gamma_1 \vdash \langle o \,|\, \mathit{Fld}: \varepsilon, \mathit{Pr}\colon \langle \overline{s}', \overline{x \mapsto \texttt{default}(T)} \rangle, \mathit{PrQ}\colon \emptyset, \mathit{EvQ}_{(\emptyset,\emptyset)}\colon \emptyset \rangle.$$

For the *induction step*, assume that $\Gamma \vdash \langle o \,|\, \mathit{Fld}: \overline{f}, \mathit{Pr}: \langle s; \overline{s}, \overline{l} \rangle, \mathit{PrQ}: \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})}: \overline{q} \rangle\ \mathit{config}\ \texttt{ok}$ and consider the reduction

$$\begin{aligned} &\langle o \,|\, \mathit{Fld}: \overline{f}, \mathit{Pr}: \langle s; \overline{s}, \overline{l} \rangle, \mathit{PrQ}: \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})}: \overline{q} \rangle\ \mathit{config} \\ &\quad \to \langle o \,|\, \mathit{Fld}: \overline{f}', \mathit{Pr}: \langle s'; \overline{s}, \overline{l}' \rangle, \mathit{PrQ}: \overline{w}', \mathit{EvQ}_{(\mathcal{L}',\mathcal{G}')}: \overline{q}' \rangle\ \mathit{config}. \end{aligned}$$

The proof proceeds by case analysis over the reduction rules.

Case R1. The process $\langle o \,|\, \mathit{Fld}: \overline{f}, \mathit{Pr}: \langle v := e; \overline{s}, \overline{l} \rangle, \mathit{PrQ}: \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})}: \overline{q} \rangle$ reduces to $\langle o \,|\, \mathit{Fld}: \overline{f}, \mathit{Pr}: \overline{s}, (\overline{l}[v \mapsto [\![e]\!]_{(\overline{f}\cdot\overline{l})}]) \rangle, \mathit{PrQ}: \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})}: \overline{q} \rangle$ or to $\langle o \,|\, \mathit{Fld}: \overline{f}[v \mapsto [\![e]\!]_{(\overline{f}\cdot\overline{l})} \mathit{Pr}: \langle \overline{s}, \overline{l} \rangle, \mathit{PrQ}: \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})}: \overline{q} \rangle$. Since the object is well-typed, we may assume from (ASSIGN) that $\Gamma(v) = T$, $\Gamma \vdash e : T'$, $T' \preceq T$, and $\Gamma([\![e]\!]_{(\overline{f}\cdot\overline{l})}) = T''$ such that $T'' \preceq T'$. By transitivity $T'' \preceq T$ and by either (PROCESS) or (STATE) the object is well-typed after the assignment.

Case R2. For $\mathit{Pr} = \langle v{:=}\texttt{new}\, C(); \overline{s}, \overline{l} \rangle$, it follows from (NEW) that $C \preceq \Gamma(v)$. Consequently, the reduced process $\langle v := (n : C); \overline{s}, \overline{l} \rangle$ is well-typed in the *extended typing context* $\Gamma' = \Gamma[(n : C) \mapsto C]$. The predicate *fresh*$(n)$ guarantees that $(n : C)$ is a unique object identifier, so we have $\Gamma \subseteq \Gamma'$. For the new object $(n : C)$, the state provided by *fields*(C) is well-typed by assumption (binding declared variables to default values of their respective types) and the assignment $\texttt{this} := (n : C)$ is well-typed in Γ'. Finally the queues are empty and, by (OBJECT), the new object is well-typed in Γ'.

Case R3. The process $\mathit{Pr} = \langle \texttt{await}\ g; \overline{s}, \overline{l} \rangle$ reduces to $\langle \overline{s}, \overline{l} \rangle$. By IH, $\Gamma \vdash \langle \overline{s}, \overline{l} \rangle\ \texttt{ok}$.

Case R4. By assumption, the source program is well-typed, so all method bodies in the source program are also well-typed. The considered process is a reduction of the translated runtime code of a well-typed method body. Hence, there must be a static judgment $\Gamma, \varepsilon \vdash \texttt{await}\ g \rhd \texttt{await}\ g, \Delta$, where Γ is the static typing context for the method body. Note that the effect of the (AWAIT) rule ensures that $\Delta(t) \neq \bot$. Furthermore two label variables in an object cannot have the same value, due to the freshness condition in rule R7 and the exclusion of explicit assignment to label variables in typing rule (ASSIGN).

Now assume that R4 were applicable at runtime. Then *dealloc*$(g, \overline{f} \cdot \overline{l}, G)$ must hold; i.e., for some guard $t? \in g$, the completion message associated with t has already been deallocated before the execution arrives at $\texttt{await}\ g$. Consequently, a statement $\texttt{free}(t)$ must have been inserted into the runtime code before `await` g by the type-based translation from the source code of the method body, such that the value bound to t is unchanged by the execution between `free`(t) and `await` g.

Let Δ_1 be the effect after the analysis of $\overline{s1}$ in the context Γ, Δ_0. In order for the type-based translation to insert the statement `free`(t) in the runtime code, three subcases correspond to the different possible judgments:

1. $\Gamma, \Delta_0 \vdash \texttt{await } g'; \overline{s1} \rhd \texttt{await } g'; \texttt{free}(t); \overline{s1}', \Delta_0'$
2. $\Gamma, \Delta_0 \vdash t!e.m(\overline{e}); \overline{s1} \rhd t!e.m(T \to \texttt{Data}, \overline{e}); \texttt{free}(t); \overline{s1}', \Delta_0'$
3. $\Gamma, \Delta_0 \vdash \overline{s1} \Box \overline{s2} \rhd (\texttt{free}(t); \overline{s1}) \Box \overline{s2}, (\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2)$

In subcases 1 and 2, $\Delta_0(t) = \bot$ by rule (Await) or (Call2) in the static type system. In subcase 3, $\Delta_0 = \bot$ and $\Delta_1 = \bot$ by rule (Choice). (The case where `free`(t) occurs in the right hand branch is similar. The cases where both `await` g and `free`(t) occurs in one of the branches are covered previously.) Since the value bound to t at runtime does not change between `free`(t) and `await` g, it follows by induction over the program statements preceding `await` g, that $\Delta_0(t) = \Delta(t) \neq \bot$, and we get a contradiction. Thus, rule R4 is not applicable in the execution from a well-typed source program.

Case R5. We have the process $Pr = \langle \texttt{await } g; \overline{s}, \overline{l} \rangle$ and the object reduces to $\langle o \,|\, Fld : \overline{f}, Pr : \texttt{release}; \texttt{await } g; \overline{s}, PrQ : \overline{w}, EvQ_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle$. As Pr is well-typed, the resulting configuration is well-typed.

Case R6. We have the process $Pr = \langle \texttt{release}; \overline{s}, \overline{l} \rangle$ and the object reduces to $\langle o \,|\, Fld : \overline{f}, Pr : \texttt{idle}, PrQ : (\overline{w}\ \langle \overline{s}, \overline{l} \rangle), EvQ_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle$. As Pr and PrQ are well-typed, the resulting configuration is well-typed.

Case R7. By IH PrQ is well-typed, so $\Gamma \vdash \langle \overline{s}, \overline{l} \rangle\ \overline{w}$ `ok`. For PrQ, $\Gamma \vdash \overline{w}$ `ok` and for Pr, $\Gamma \vdash \langle \overline{s}, \overline{l} \rangle$ `ok`. Consequently, by (Object), the object is well-typed.

Case R8. The process $Pr = \langle t!x.m(Sig, \overline{e}); \overline{s}, \overline{l} \rangle$ reduces to $\langle t := mid; \overline{s}, \overline{l} \rangle$. By IH, $\Gamma \vdash \langle t!x.m(Sig, \overline{e}); \overline{s}, \overline{l} \rangle$ `ok`, and we may assume $\Gamma(t) = \texttt{Label}$. Since mid has type `Label` the assignment is well-typed, and $\Gamma \vdash \langle t := mid; \overline{s}, \overline{l} \rangle$ `ok`. A message $invoc(m, T \to T', (\llbracket \overline{e} \rrbracket_{(\overline{f} \cdot \overline{l})}), \langle o, mid \rangle)$ **to** $\llbracket x \rrbracket_{(\overline{f} \cdot \overline{l})}$ is added to the configuration. (An invocation message from o is uniquely identified by mid) Since $\Gamma \vdash t!x.m(T \to T', \overline{e})$ `ok`, we have $\Gamma \vdash \overline{e} : T$, $\Gamma \vdash \llbracket \overline{e} \rrbracket_{(\overline{f} \cdot \overline{l})} : T'$, and $T' \preceq T$. Consequently, by (Msg3) we get $\Gamma \vdash invoc(m, T \to T', (\llbracket \overline{e} \rrbracket_{(\overline{f} \cdot \overline{l})}), \langle o, mid \rangle)$ **to** $\llbracket x \rrbracket_{(\overline{f} \cdot \overline{l})}$ `ok`.

Case R9. We have a well-typed message $\Gamma \vdash msg$ **to** o ok and a well-typed object o with event queue $EvQ_{(\mathcal{L},\mathcal{G})} = \overline{q}$. Since $\Gamma \vdash msg$ `ok` and $\Gamma \vdash \overline{q}$ `ok`, by (EventQ) we get $\Gamma \vdash \overline{q}\ msg$ `ok`.

Case R10. We have $EvQ_{(\mathcal{L},\mathcal{G})} = \overline{q}\ invoc(m, Sig, \overline{e}, \langle o, mid \rangle)$ and $PrQ = \overline{w}$. The reduction results in $PrQ = \overline{w}\ lookup(C, m, Sig, (\overline{e}, o, mid))$ and $EvQ_{(\mathcal{L},\mathcal{G})} = \overline{q}$. Since $\Gamma \vdash \overline{q}\ invoc(m, Sig, \overline{e}, \langle o, mid \rangle)$ `ok`, $\Gamma \vdash \overline{q}$ `ok`. By assumption every object identifier is unique, so the class of o has among its superclasses the statically assumed class for which $lookup(\Gamma(o), m, Sig)$ succeeds. Consequently, the runtime $lookup(\Gamma(o), m, Sig, (\overline{e}, o, mid))$ returns a process which correctly matches the call and which is well-typed by (Process).

Case R11. We have $Pr = \langle t?(v); \overline{s}, \overline{l} \rangle$ and $EvQ_{(\mathcal{L},\mathcal{G})} = \overline{q}\ comp(mid, T, e)$ such that t is bound to mid in object o. The process reduces to $\langle v := e; \overline{s}, \overline{l} \rangle$ and the event queue to $\overline{q}$. As completion messages result from method invocations only and mid is fresh by R8, there must be a corresponding well-typed invocation with label t, say $t!x.m(T' \to T'', \overline{e})$, such that $T'' \preceq \Gamma(v)$,

$\Gamma(x) = C$, and $lookup(C,m,T' \to T'')$. Consequently $T \preceq T''$, and by Theorem 2 and transitivity, $T \preceq \Gamma(v)$, so (REPLY1) holds and $\Gamma \vdash \langle v := e;\bar{s},\bar{l}\rangle$ `ok` and $\Gamma \vdash \bar{q}$ `ok`.

Case R12. Similar to *Case R4*.

Case R13. The completion message $comp([\![\gamma]\!]_{\bar{l}}, [\![\beta]\!]_{\bar{l}}, [\![e]\!]_{\bar{f}\cdot\bar{l}})$ **to** $[\![\alpha]\!]_{\bar{l}}$ is introduced into the configuration. The process reduces to $\langle \bar{s},\bar{l}\rangle$, which is well-typed. Since the program is statically well-typed, we may assume $\Gamma \vdash e : T$ and $\Gamma \vdash [\![e]\!]_{\bar{l}} : T'$ such that $T' \preceq T$. The typing rule (METHOD) asserts $T \preceq [\![\beta]\!]_{\bar{l}}$. Consequently, $T' \preceq [\![\beta]\!]_{\bar{l}}$ and by (MSG3) we get $\Gamma \vdash comp([\![\gamma]\!]_{\bar{l}}, [\![\beta]\!]_{\bar{l}}, [\![e]\!]_{\bar{l}})$ **to** $[\![\alpha]\!]_{\bar{l}}$ `ok`.

Case R14. We have $Pr = \langle \bar{s}_1 \Box \bar{s}_2\rangle$, which reduces $\bar{s}_1 \Box \bar{s}_2$ to either $\bar{s}_1$ or $\bar{s}_2$. The IH gives us directly that the object is well-typed.

Case R15. We have $Pr = \langle (\texttt{free}(\bar{t});\bar{s}),\bar{l}\rangle$ and $EvQ_{(\mathcal{L},\mathcal{G})} = \bar{q}$. Since the program is well-typed, $\Gamma(t) = \mathsf{Label}$. It follows from IH that the object is well-typed.

Case R16. We have the event queue $EvQ_{(\{mid\}\cup\mathcal{L},\mathcal{G})} = comp(mid,T,e)\ \bar{q}$. By IH, $\Gamma \vdash comp(mid,T,e)\ \bar{q}$ `ok`. Consequently by (EVENTQ), $\Gamma \vdash \bar{q}$ `ok`. ■

Chapter 9

Paper 3: Type-Safe Runtime Class Upgrades in Creol

Ingrid Chieh Yu, Einar Broch Johnsen, and Olaf Owe

Publication.
Published in Proceedings of the 8th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006).

Abstract.
Modern applications distributed across networks such as the Internet may need to evolve without compromising application availability. Object systems are well suited for runtime update, as encapsulation clearly separates internal structure and external services. This paper considers a type-safe asynchronous mechanism for dynamic class upgrade, allowing class hierarchies to be updated in such a way that the existing objects of the upgraded class and of its subclasses gradually evolve at runtime. New external services may be introduced in classes and old services may be reprogrammed while static type checking ensures that asynchronous class updates maintain type safety. A formalization is shown in the Creol language which, addressing distributed and object-oriented systems, provides a natural framework for dynamic upgrades.

9.1 Introduction

Long-lived distributed applications with high availability requirements need the ability to adapt to new requirements that arise over time without compromising application availability. These requirements include bugfixes but also new or improved features. Examples of such applications are found in financial transaction processes, aeronautics and space missions, and mobile and Internet applications. In these examples, updates must be applied at runtime. Early approaches to software updates [5, 13, 17] do not address the issue of continuous availability, but runtime reconfiguration and upgrade have recently attracted attention [4,3,10,11,12,20,18,2,6,22]. In large distributed systems runtime updates need to be applied in an asynchronous and modular way, and propagate gradually through the distributed system. An appropriate update system should [2, 22]: propagate updates automatically, provide a means to control *when* components may be upgraded, and ensure the availability of system services during the upgrade process.

This paper considers a type-safe mechanism for distributed runtime updates in Creol [14], a formally defined object-oriented language which specifically targets open distributed systems. We consider updates in the form of runtime upgrades of existing classes combined with runtime additions of new interfaces and new classes. Upgrading a class affects all future and existing object instances of the class and its subclasses. As runtime upgrades are handled by asynchronous messages, allowing message overtaking, dependencies between different upgrades could violate type safety. Extending previous work [15], this paper introduces a type system for class upgrades which derives the upgrade dependencies of each upgrade. These dependencies enforce an ordering of the upgrades in the runtime system, formalized in rewriting logic [19], which ensures that the application of the distributed upgrades is type-safe. Consequently, runtime class upgrades will not introduce type errors. The upgrade mechanism proposed in this paper allows new interfaces to be added to classes at runtime. This way upgraded classes may provide new external services. The following simple example illustrates dependencies between several updates.

Motivating example. We adopt a separation of concerns between external service specifications, given as interfaces, and implementation code, organized in classes. Object pointers are typed by interfaces while objects are instances of classes. A type system is used to ensure that methods invoked on object pointers are supported by the objects. Consider a simple scenario with three classes C_1, C_2, and C_3, where C_3 inherits C_2 (the comment *V:1* means version 1 of a class):

```
class C1 --- V:1, U:0        class C2 --- V:1, U:0        class C3 --- V:1, U:0
begin                        begin   end                  inherits C2
op run() == n(); run()                                    begin   end
op n() == skip
end
```

The example sketch is given in Creol, *U:0* comments that a class has not (yet) been upgraded. Here, C_1 objects are active as the *run* method is activated at object creation, with a nonterminating behavior consisting of repeated local calls to a method *n*. The external functionality of each class is given by its interfaces. None are given here, so in this example only internal calls are possible in C_1.

By *dynamically upgrading* the class C_2 with a new method m, this method will become available via objects of classes C_2 and its subclass C_3. However, after the update the new method is only known internally in these classes. In order to *export* the new functionality, we dynamically add a new interface I providing a method m with an appropriate signature, after which m may be invoked on pointers typed by I. If we can type check that C_3 implements I, it is type-safe to bind a pointer typed by I to an instance of C_3 and invoke the new method m on this object. This may be achieved by dynamically redefining method n in class C_1 to create an appropriately typed instance of C_3 and invoke m on this instance, for instance by the code **var** $x : I; x :=$ **new** $C_3(); x.m()$. These dynamic updates may be realized by four update messages added to the running system: introducing I, upgrading C_1 by the redefinition of n, C_2 by a new method m, and C_3 by the new interface I. After successful upgrades (*U:1*), the following classes replace the previous runtime class definitions:

```
class C1 --- V:2, U:1              class C2 --- V:2, U:1        class C3 --- V:3, U:1
begin                              begin                        implements I
op run() == n(); run()             op m() == Body               inherits C2
op n() == var x : I;               end                          begin    end
    x := new C3(); x.m()
end
```

Furthermore, the active behavior of existing instances of C_1 now create instances of C_3 on which the new method m is invoked.

A type-safe introduction of these upgrades in a distributed system requires a combination of type checking and careful timing at runtime. In particular, the redefinition of method n has an immediate effect on any instance of C_1. In order to avoid errors, this upgrade cannot be applied *before* C_3 implements the new interface I. However, the addition of the new interface requires the presence of method m, which in turn requires that the application of the upgrade of C_2 has *already* occurred. In fact, C_3 has been upgraded twice, once directly and once indirectly through the upgrade of C_2. This paper formalizes an asynchronous update mechanism which handles these dependencies, maintaining runtime type safety throughout the upgrade process.

Paper overview. Sect. 9.2 introduces behavioral interfaces, Sect. 9.3 summarizes Creol, Sect. 9.4 presents Creol's type system, and Sect. 9.5 presents the dynamic class construct. Sect. 9.6 discusses related work and Sect. 9.7 concludes the paper.

9.2 Behavioral Interfaces

An object may assume different roles, depending on the context of interaction, which are captured by specifications of aspects of its externally observable behavior. A *behavioral interface* consists of a set of method names with signatures and semantic constraints on the use of these methods. In this paper we restrict semantic constraints to cointerface requirements, explained as follows: For active objects it may be desirable to restrict access to the methods in an interface to calling objects of a particular *cointerface*. This way the called object may invoke methods of the caller and not only passively complete invocations of its own methods, thus providing support for callback. *Mutual dependency* is specified if two interfaces have each other as cointerface. Let *Any* be the superinterface of all interfaces; *Any* is used as cointerface if no callback knowledge is required.

Object references (pointers) are typed by behavioral interfaces. References typed by different interfaces may refer to the same object identifier. A class *implements* an interface if its object instances provide the behavior described by the interface. A class may implement several interfaces and different classes may implement the same interface. Reasoning control is ensured by interface-level substitutability: *a reference typed by an interface I may be replaced by another reference typed by I or by a subinterface of I.* This substitutability is reflected in the executable language by the fact that late binding applies to all external method calls, as the runtime class of the object need not be statically known.

Let τ_B be a set of basic data type names and τ_I a set of interface names, such that $\tau_B \cap \tau_I = \emptyset$. Let τ denote the set of all types; $\tau_B \subseteq \tau$ and $\tau_I \subseteq \tau$. Let I and J be typical elements of τ_I, and T of τ. We assume that τ_B includes standard types such as Booleans and natural numbers. Type schemes such as parametrized data types may be applied to types in τ to form new types in τ, $\mathsf{Set}[T]$ and $\mathsf{List}[T]$ are included among the type schemes. To conveniently organize object viewpoints, interfaces may be structured in an inheritance hierarchy.

Definition 1 An *interface* is denoted by a term $\mathit{int}(\mathit{Inh}, \mathit{Mtd})$ of type I, where Inh is a list of (inherited) interfaces and Mtd is a set of method declarations $\mathit{mdecl}(\mathit{Nm}, \mathit{Co}, \mathit{In}, \mathit{Out})$, where Nm is a method name, Co is a cointerface, and In and Out are lists of parameter types.

Dot notation is used to access the elements of tuples such as methods and interfaces; e.g., $\mathit{int}(\mathit{Is}, M).\mathit{Mtd} = M$. The empty list is denoted ε. The name $\mathit{Any} \in \tau_I$ is reserved for $\mathit{int}(\varepsilon, \emptyset)$, and the name $\mathit{Internal} \in \tau_I$ is reserved for type checking purposes (see Sect. 9.3). If I inherits J, the methods of both I and J must be available in any class that implements I. We consider a nominal subtype relation [21] for interfaces. Two interfaces with the same set of methods may be part of different subtype relationships.

9.3 Creol: A Language for Distributed Concurrent Objects

Creol is a high-level object-oriented language targeting open distributed systems by combining interface types and concurrent objects with asynchronous method calls, and by combining active and reactive object behavior [16, 14]. In this paper blocking and nonblocking (suspending) method calls are considered, although the results of the paper apply to the full language. An object has its own processor which evaluates local processes. Processes result from method activations. Active behavior is initiated by the special *run* method, activated at object creation, and interleaved with reactive behavior by means of suspension. Due to suspension, the values of object variables may depend on the nondeterministic interleaving of processes, so local process variables supplement the object variables and include the formal parameters. An object may contain several (pending) activations of a method, possibly with different values for local variables.

Objects only interact through asynchronous method calls. Calls can always be emitted, as a receiving object cannot block communication. Method overtaking is allowed: if methods offered by an object are invoked in one order, the object may start execution of the method

$$
\begin{array}{lll}
CL & ::= & [\textbf{class}\ C\ [(\mathit{Vdecl})]^?\ [\textbf{implements}\ [I]^+_{;}]^?\ [\textbf{inherits}\ [C[(\mathrm{E})]^?]^+_{;}]^? \\
 & & \textbf{begin}\ [\textbf{var}\ \mathit{Vdecl}]^?\ [[\textbf{with}\ I]^?\ \mathit{Methods}]^*\ \textbf{end}]^* \\
\mathit{Methods} & ::= & [\textbf{op}\ m\ ([\textbf{in}\ \mathit{Vdecl}]^?\ [\textbf{out}\ \mathit{Vdecl}]^?) == [\textbf{var}\ \mathit{Vdecl};]^?\ \mathrm{S}]^+ \\
\mathit{Vdecl} & ::= & [v : T]^+_{;}
\end{array}
$$

Figure 9.1: An outline of the language syntax for classes, excluding expressions e, expression lists E, and statement lists S. The meta notation $[\dots]^?$ denotes optional parts, $[\dots]^*$ repetition zero or more times, and $[\dots]^+_d$ non-empty repetition with d as delimiter.

activations in another order. A *blocking* call $x.m(\mathrm{E};\mathrm{V})$ immediately blocks the processor while waiting for a reply. A *nonblocking* call **await** $x.m(\mathrm{E};\mathrm{V})$ releases the processor while waiting for a reply, allowing other processes to execute. When the reply arrives, the suspended process becomes enabled and evaluation may resume. This approach provides flexibility in the distributed setting: suspended processes or new method activations may be evaluated while waiting. If the called object never replies, deadlock is avoided as other activity in the object is possible. However, when the reply arrives, the *continuation* of the process must compete with other pending and enabled processes. After processor release, any enabled pending process may be selected for evaluation. When x evaluates to *self*, the call is said to be local. *Internal* calls are not prefixed by an object identifier and are identified syntactically, otherwise the call is external. All internal calls are here late bound.

The language distinguishes data, typed by data types, and objects, typed by interfaces. We assume given a *strongly typed functional language* of well-typed expressions $e \in \mathsf{Expr}$ without side effects, including two subtypes $\mathsf{ObjExpr}$ and $\mathsf{BoolExpr}$ whose expressions reduce to object references (typed by interface) and Booleans, respectively. There are no constructors or field access functions for terms in $\mathsf{ObjExpr}$, but variables bound to object references may be compared by an equality function. Let Γ_F be a typing environment which includes all relevant type information for the constants and functions of the functional language, and let Γ extend Γ_F with variable declarations. Then $\Gamma \vdash_{\mathrm{F}} e : T$ denotes that e has type T in Γ. It is assumed that expressions are *type-sound*: well-typed expressions remain well-typed during evaluation. If $\Gamma \vdash_{\mathrm{F}} e : T$ and e reduces to e', then $\Gamma \vdash_{\mathrm{F}} e' : T'$ such that $T' \preceq T$.

Object-oriented features extend the functional language. Class definitions include declarations of persistent state variables and method definitions.

Definition 2 A *class* is denoted by a term $\mathit{class}(\mathit{Par}, \mathit{Upg}, \mathit{Imp}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd})$, where *Par* is a list of typed program variables, *Upg* the current upgrade number, *Imp* a list of interface names, *Inh* a list of class names, defining class inheritance, *Var* a list of typed program variables (possibly with initial expressions), and *Mtd* a set of methods $\mathit{mtd}(\mathit{Nm}, \mathit{Co}, \mathit{In}, \mathit{Out}, \mathit{Body})$ where *Nm* is a method name, *Co* an interface, *In* and *Out* lists of variable declarations, and *Body* a pair of variable declarations *Vdecl* and statements S.

The *Upg* attribute is not a part of the Creol syntax and cannot be altered by programmers. For internal methods, the cointerface field is *Internal*. The field *Imp* represents interfaces supported by this class. The typing of remote method calls in a class C relies on the fact that the calling object supports the interfaces of C, and these are used to check any cointerface requirements of the calls.

Syntactic categories.						*Definitions.*
s	in	Stm	v	in	Var	$p ::= m \mid x.m$
m	in	Mtd	p	in	MtdCall	$\mathrm{S} ::= s \mid s;\mathrm{S}$
e	in	Expr	x	in	ObjExpr	$s ::= \mathbf{skip} \mid \mathrm{V} := \mathrm{E} \mid v := \mathbf{new}\ C(\mathrm{E}) \mid p(\mathrm{E};\mathrm{V}) \mid \mathbf{await}\ p(\mathrm{E};\mathrm{V})$

Figure 9.2: Program statements in method definitions, with typical terms for each category. Capitalized terms such as E denote lists of the given syntactic categories.

Let τ_C denote the set of class names, with typical element C, and $\mathcal{C}$ the set of class terms. An abstract representation of a class may be given following the BNF syntax of Figure 9.1. Method declarations in classes consist of local variable declarations and a list of program statements (see Figure 9.2). Assignment to local and object variables is expressed as $\mathrm{V} := \mathrm{E}$ for a disjoint list of program variables V and an expression list E, of matching types. In-parameters as well as the pseudo-variables `this`, for self reference, and *caller* are read-only variables.

Due to the interface typing of object variables, the actual class of the receiver of an external call is not statically known. Consequently, external calls are *late bound.* Let the nominal subtype relation $\preceq$ be a reflexive partial ordering on types, including interfaces. The nominal subtype relation restricts a structural subtype relation which ensures substitutability; If $T \preceq T'$ then any value of T may masquerade as a value of T' [21]. For product types R and R', $R \preceq R'$ is the point-wise extension of the subtype relation. To explain the typing and binding of methods, $\preceq$ is extended to function spaces $A \rightarrow B$, where A and B are (possibly empty) product types: $A \rightarrow B \preceq A' \rightarrow B' \Leftrightarrow A' \preceq A \wedge B \preceq B'$. The static analysis of an internal call $m(\mathrm{E};\mathrm{V})$ or **await** $m(\mathrm{E};\mathrm{V})$ will assign unique types to the in- and out-parameters depending on the textual context, say $\mathrm{E} : T_{\mathrm{E}}$ and $\mathrm{V} : T_{\mathrm{V}}$. The call is *type-correct* if there is a method declaration $m : T_1 \rightarrow T_2$ in the class C such that $T_1 \rightarrow T_2 \preceq T_{\mathrm{E}} \rightarrow T_{\mathrm{V}}$. An external call $o.m(\mathrm{E};\mathrm{V})$ or **await** $o.m(\mathrm{E};\mathrm{V})$ to an object o of interface I is type-correct if it can be bound to a method declaration in I in a similar way. The static analysis of a class will verify that it implements its declared interfaces. Assuming that any object variable typed by I is an instance of a class implementing I, method binding at runtime will succeed regardless of the dynamically identified class of the object.

9.4 Typing

The typing environment Γ in Creol's nominal type system is a *mapping family*: Γ_I maps interface names to interfaces, Γ_C class names to classes, and Γ_{V} program variable names to types. Without class upgrades, Γ_I and Γ_C correspond to static tables. Declarations may only update Γ_{V}, and program statements may not update Γ_{V}. For the purposes of dynamic updates, a *dependency mapping* Γ_d captures the dependencies that a class has to different classes in the program.

Definition 3 The *dependency mapping* $\Gamma_d : \tau_C \times \mathsf{Nat} \rightarrow \mathsf{Set}[\tau_C \times \mathsf{Nat}]$ maps pairs of class names and upgrade numbers to sets of such pairs.

Each upgrade of a class C is uniquely identified by a pair $\langle C, u \rangle$. Thus, elements in $\Gamma_d(\langle C, u \rangle)$ represent classes on which upgrade u of class C depends, and structural requirements to these classes. At runtime Γ_d helps to monitor whether these structural requirements are fulfilled, and to enforce an ordering of local updates obeying the dependency requirements.

The type analysis of a syntactic construct D is formalized by a deductive system for judgments $\Gamma \vdash D \langle\Delta\rangle$, where Γ is the typing environment and Δ the *update* of the typing environment. After analysis of D, the typing environment becomes Γ *overridden by* Δ, denoted $\Gamma+\Delta$. Sequential composition has the rule

$$\text{(SEQ)} \quad \frac{\Gamma \vdash D \langle\Delta\rangle \qquad \Gamma+\Delta \vdash D' \langle\Delta'\rangle}{\Gamma \vdash D;D' \langle\Delta+\Delta'\rangle}$$

where $+$ is an associative operator on mappings with the identity element $\emptyset$. We abbreviate $\Gamma \vdash D \langle\emptyset\rangle$ to $\Gamma \vdash D$. Mapping families are now formally defined.

Definition 4 Let n be a name, d a declaration, $i \in I$ a mapping index, and $[n \mapsto_i d]$ the binding of n to d indexed by i. A *mapping family* Γ is built from the empty mapping family $\emptyset$ and indexed bindings by the constructor $+$. The *extraction* of an indexed mapping Γ_i from Γ and *application* for the indexed mapping Γ_i, are defined as follows

$$\begin{array}{lcl} \emptyset_i & = & \varepsilon \\ (\Gamma+[n \mapsto_{i'} d])_i & = & \textbf{if } i=i' \textbf{ then } \Gamma_i+[n \mapsto_i d] \textbf{ else } \Gamma_i \\ \\ \varepsilon(n) & = & \perp \\ (\Gamma_i+[n \mapsto_i d])(n') & = & \textbf{if } n=n' \textbf{ then } d \textbf{ else } \Gamma_i(n'). \end{array}$$

A class or interface declaration binds a name to a class or interface term, respectively. Class and interface names need not be distinct. A program consists of a list of interface and class declarations, represented by the mappings Γ_I and Γ_C. For type checking a program, each interface and class term is type checked based on these mappings (binding `this` to the class name in the second case). The type rules are given in Figure 9.3 (omitting the rule for interfaces). To simplify the exposition, some auxiliary functions are used to retrieve information from the typing environment. The predicate *matchpar* verifies that the formal and actual parameters of (inherited) classes match, given a list of classes and a typing environment. The predicate *matchext* checks that an external invocation may be bound through the interface of the callee, based on the types of actual parameter values and the possible cointerfaces of the caller. The function *matchint* returns a list of classes in which an internal invocation may be bound given a method, a list of classes, and a typing environment. This function is used to check that a class provides method bodies for the method declarations of its interfaces, and for type checking internal calls. The function *InhAttr* returns a list of typed variables when given a list of classes and a typing environment, and is used to extend the typing environment with inherited attributes.

The main type rules are now briefly explained. Programs are type checked in the context of Γ_F. Variable declarations extend the context used to type check methods in rule (CLASS). Local variable declarations extend the typing environment used to type check the program statements of a method in rule (METHOD). For object creation, (NEW) ensures that the class must implement an interface which is a subtype of the declared interface of the object pointer. For external calls $x.m$, (EXT) checks that the interface of x offers a method m with a cointerface implemented by the class of the caller. Consequently, *remote calls* to `this` are allowed when the class implements an interface used as the cointerface of the method in the current class. For internal calls

$$\text{(PROG)}\quad \frac{\forall I \in \tau_I \cdot \Gamma_I \vdash \Gamma_I(I) \qquad \forall C \in \tau_C \cdot \Gamma_F + \Gamma_I + \Gamma_C + [\mathtt{this} \mapsto_{\mathrm{v}} C] \vdash \Gamma_C(C)}{\Gamma_F \vdash \Gamma_I, \Gamma_C}$$

$$\text{(CLASS)}\quad \frac{\begin{array}{c} \Gamma \vdash \mathit{Par}\,\langle\Delta\rangle \qquad \Gamma + \Delta \vdash \mathit{InhAttr}(\mathit{Inh}, \Gamma_C), \mathit{Var}\,\langle\Delta'\rangle \\ \mathit{matchpar}(\Gamma + \Delta, \mathit{Inh}) \qquad \forall m \in \mathit{Mtd} \cdot \Gamma + \Delta + \Delta' \vdash m\,\langle\Delta^m\rangle \\ \forall I \in \mathit{Imp} \cdot \forall m \in \Gamma_I(I).\mathit{Mtd} \cdot \mathit{matchint}(m, \Gamma_{\mathrm{V}}(\mathtt{this}), \Gamma) \neq \varepsilon \end{array}}{\Gamma \vdash \mathit{class}\,(\mathit{Par}, \mathit{Upg}, \mathit{Imp}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd})\,\langle\Delta + \Delta' + \bigcup_{m \in \mathit{Mtd}} \Delta^m\rangle}$$

$$\text{(METHOD)}\quad \frac{\Gamma \vdash (\mathit{caller} : \mathit{Co}); \mathit{In}; \mathit{Out}; \mathit{Body}\,\langle\Delta\rangle}{\Gamma \vdash \mathit{mtd}\,(\mathit{Nm}, \mathit{Co}, \mathit{In}, \mathit{Out}, \mathit{Body})\,\langle\Delta_d\rangle}$$

$$\text{(SKIP)}\quad \Gamma \vdash \mathbf{skip} \qquad \text{(ASSIGN)}\quad \frac{\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T' \qquad T' \preceq \Gamma_{\mathrm{V}}(\mathrm{V})}{\Gamma \vdash \mathrm{V} := \mathrm{E}\,\langle[\bullet \mapsto_d \Gamma_d(\bullet) \cup [\![\mathrm{V};\mathrm{E}]\!]]\rangle}$$

$$\text{(VAR)}\quad \frac{v \notin \Gamma_{\mathrm{V}} \quad T \preceq \mathsf{Data}}{\Gamma \vdash v : T\,\langle[v \mapsto_{\mathrm{V}} T]\rangle} \qquad \text{(NON-BL)}\quad \frac{\Gamma \vdash p(\mathrm{E};\mathrm{V})\,\langle\Delta\rangle}{\Gamma \vdash \mathbf{await}\ p(\mathrm{E};\mathrm{V})\,\langle\Delta\rangle}$$

$$\text{(NEW)}\quad \frac{\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T \quad T \preceq \mathit{type}(\Gamma_C(C).\mathit{Par}) \qquad \exists I \in \Gamma_C(C).\mathit{Imp} \cdot I \preceq \Gamma_{\mathrm{V}}(v)}{\Gamma \vdash v := \mathbf{new}\,C(\mathrm{E})\,\langle[\bullet \mapsto_d \Gamma_d(\bullet) \cup [\![v;\mathrm{E}]\!] \cup \{\langle C, \Gamma_C(C).\mathit{Upg}\rangle\}]\rangle}$$

$$\text{(EXT)}\quad \frac{\Gamma \vdash_{\mathrm{F}} e : I \quad \Gamma \vdash_{\mathrm{F}} \mathrm{E} : T \quad \mathrm{matchext}(m, T, \mathrm{V}, I, \Gamma_{\mathrm{V}}(\mathtt{this}), \Gamma)}{\Gamma \vdash e.m(\mathrm{E};\mathrm{V})\,\langle[\bullet \mapsto_d \Gamma_d(\bullet) \cup [\![\mathrm{E};\mathrm{V}]\!]]\rangle}$$

$$\text{(INT)}\quad \frac{\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T \qquad C' \in \mathrm{matchint}(\mathit{mtd}(m, \mathit{Internal}, T, \Gamma_{\mathrm{V}}(\mathrm{V}), \varepsilon), \Gamma_{\mathrm{V}}(\mathtt{this}), \Gamma)}{\Gamma \vdash m(\mathrm{E};\mathrm{V})\,\langle[\bullet \mapsto_d \Gamma_d(\bullet) \cup [\![\mathrm{E};\mathrm{V}]\!] \cup \{\langle C', \Gamma_C(C').\mathit{Upg}\rangle\}]\rangle}$$

Figure 9.3: The type system, where $\bullet$ acts as a placeholder for values of type $\langle\tau_C \times \mathsf{Nat}\rangle$, $[\![\mathrm{E}]\!]$ returns a set of class names and upgrade numbers for the classes in which the attributes in an expression list E are declared (relative to `this` in Γ), and *type* extracts the types of a declaration list.

m, (INT) checks that the method has cointerface *Internal*. For a variable occurring in a method body, the pair consisting of the name of the class in which the variable is declared and the upgrade number of this class, are added to the dependency mapping for the method. Similarly, matching classes for internal calls and object creations also extend the mapping. This way, the type system constructs a dependency mapping which captures the dependencies a method has to different classes in the program. This dependency mapping is exploited for system upgrades.

9.5 Dynamic Class Upgrades

New interfaces, new classes, and class upgrades may update the running system. New interfaces and classes extend the system while class upgrades allow method redefinition as well as extending the class with new attributes, methods, interfaces, and superclasses. Modifications should not compromise the type safety of the running program; e.g., a method redefinition must preserve the signature so the class consistently supports its interfaces. In an open distributed setting, upgrades of classes and objects are not sequentialized; rather, upgrades propagate *asynchronously* through the network causing objects of different versions to coexist. Consequently, the order in which upgrades happen at runtime may differ from the order in which they were type checked. For upgrades with no syntactic dependencies, this overtaking does not affect run-

$$\text{(NEW-INTERFACE)}\quad \frac{N \notin \Gamma_I \qquad \Gamma + [N \mapsto_I R] \vdash R}{\Gamma \vdash \textit{new-interface}(N,R)\,\langle N \mapsto_I R\rangle}$$

$$\text{(NEW-CLASS)}\quad \frac{N \notin \Gamma_C \qquad \Gamma + [\texttt{this} \mapsto_{\mathrm{v}} N] + [N \mapsto_C R]\ \vdash R\,\langle\Delta\rangle}{\Gamma \vdash \textit{new-class}(N,R)\,\langle [N \mapsto_C R] + [\langle N,1\rangle \mapsto_d (\Delta_d(\bullet) \setminus \{\langle N,0\rangle\})]\rangle}$$

$$\text{(UP)}\quad \frac{\begin{array}{ll} \Gamma \vdash \texttt{this} : N;\Gamma_C(N)\,\langle\Gamma'\rangle & \forall I \in \mathit{Imp} \cdot I \in \Gamma_I \\ \Gamma + \Gamma' \vdash \mathit{InhAttr}(\mathit{Inh},\Gamma_C);\mathit{Var}\ \langle\Delta\rangle & \mathit{matchpar}(\Gamma+\Gamma',\mathit{Inh}) \\ \multicolumn{2}{l}{\forall m \in \mathit{Mtd} \cdot\ \textbf{if}\ m.\mathit{Nm} \in \Gamma_C(N).\mathit{Mtd}} \\ \multicolumn{2}{l}{\quad \textbf{then}\ \Gamma+\Gamma'+[N \mapsto_C \mathit{upg}(\Gamma_C(N),0,\varepsilon,\mathit{Inh},\varepsilon,\mathit{Mtd}\setminus m)] + \Delta \vdash_r m\,\langle\Delta^m\rangle} \\ \multicolumn{2}{l}{\quad \textbf{else}\ \ \Gamma+\Gamma'+[N \mapsto_C \mathit{upg}(\Gamma_C(N),0,\varepsilon,\mathit{Inh},\varepsilon,\mathit{Mtd})] + \Delta \vdash m\,\langle\Delta^m\rangle\ \textbf{fi}} \\ \multicolumn{2}{l}{\forall I \in \mathit{Imp} \cdot \forall m' \in \Gamma_I(I).\mathit{Mtd}\ \cdot (\mathit{matchint}(m',(N;\mathit{Inh}),\Gamma) \neq \varepsilon} \\ \multicolumn{2}{l}{\qquad\qquad \vee (\exists m \in \mathit{Mtd}(m'.\mathit{Nm}) \cdot \mathit{Sig}(m) \preceq \mathit{Sig}(m')))} \end{array}}{\begin{array}{l} \Gamma \vdash \mathit{upd}(N,\mathit{Imp},\mathit{Inh},\mathit{Var},\mathit{Mtd})\ \langle\,[N \mapsto_C \mathit{upg}(\Gamma_C(N),1,\mathit{Imp},\mathit{Inh},\mathit{Var},\mathit{Mtd})] \\ \qquad + [\langle N,\Gamma_C(N).\mathit{Upg}+1\rangle \mapsto_d \bigcup_{m\in \mathit{Mtd}} \Delta^m_d(\bullet) \cup \{\langle N,\Gamma_C(N).\mathit{Upg}\rangle\}]\rangle \end{array}}$$

$$\text{(MTD-RDEF)}\quad \frac{\begin{array}{c} \mathit{Sig}(\mathit{mdef}) \preceq \mathit{Sig}(\Gamma_C(\Gamma_{\mathrm{v}}(\texttt{this})).\mathit{Mtd}(\mathit{mdef}.\mathit{Nm})) \\ \Gamma + [\Gamma_C(\Gamma_{\mathrm{v}}(\texttt{this}) \mapsto_C \mathit{upg}(\Gamma_C(\Gamma_{\mathrm{v}}(\texttt{this}),0,\varepsilon,\varepsilon,\varepsilon,\mathit{mdef})] \vdash \mathit{mdef}\,\langle\Delta\rangle \end{array}}{\Gamma \vdash_r \mathit{mdef}\,\langle\Delta_d(\bullet)\rangle}$$

Figure 9.4: The type system for class upgrades. Here, $\vdash_r$ is used for type checking of redefined methods, and $\mathit{Mtd}(N)$ denotes the subset of methods in Mtd with name N.

time type safety. If there are syntactic dependencies between upgrades, the order of upgrades must respect these dependencies. The following kinds of system updates are considered:

Definition 5 Systems are updated through the following operations:

- An *interface addition* is represented by a term $\textit{new-interface}(N,R)$, where N is an interface name and R is an interface term.
- A *class addition* is represented by a term $\textit{new-class}(N,R)$, where N is a class name and R is a class term.
- A *class upgrade* is represented by a term $\mathit{upd}(N,\mathit{Imp},\mathit{Inh},\mathit{Var},\mathit{Mtd})$, where N is the name of the class to be upgraded, Imp a list of interfaces, Inh a list of classes, defining additional superclasses to be inherited, Var a list of typed program variables, and Mtd a set of methods.

Type checking class upgrades results in dependency conditions which ensure that system modifications do not violate the type safety of the running system. Given an upgrade of a class C in a well-typed program P, an upgrade is type checked based on the current typing environment Γ of P: the mappings in Γ are modified by upgrades. Thus, the upgraded versions of classes as accumulated in the environment resulting from a (successful) type checking, serve as the starting point of future updates.

9.5.1 Type Checking System Updates

The rules to type check new interfaces and classes, class upgrades, and method redefinitions are given in Figure 9.4. After type checking new interfaces and classes, the typing environment is extended. Let Γ be the typing environment after type checking a well-typed program P. An upgrade of a class $C \in P$ is then type checked in Γ; i.e., $\Gamma \vdash \mathit{upd}(C, \mathit{Imp}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd})\ \langle \Gamma'_d + {\Gamma_C}' \rangle$, where ${\Gamma_C}'$ is updates of the class representation in Γ_C, computed by the auxiliary function *upg*, and Γ'_d is dependency requirements to classes in P for the upgrade of C accumulated while type checking. The next update is type checked in $\Gamma + \Gamma'_d + {\Gamma_C}'$.

Definition 6 Let n be a natural number, I a list of interfaces, I' a list of classes, V a list of variables, and M a set of methods. The upgraded version of a class resulting from a class update is defined by the *upg* function:

$$\begin{aligned}&\mathit{upg}(\mathit{class}(\mathit{Par}, \mathit{Upg}, \mathit{Imp}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd}), n, \mathrm{I}, \mathrm{I'}, \mathrm{V}, \mathrm{M})\\ &\quad = \mathit{class}(\mathit{Par}, \mathit{Upg} + n, \mathit{Imp}; \mathrm{I}, \mathit{Inh}; \mathrm{I'}, \mathit{Var}; \mathrm{V}, \mathit{Mtd} \oplus \mathrm{M})\end{aligned}$$

For class upgrades, the typing environment is reloaded for the upgrading class before type checking the upgrade elements with the rule (UP). By adding new interfaces, the class may provide new external services. For each new interface, the type system requires that the class provides, either by inheritance, by local declarations, or by the current upgrade, at least one type-correct method body for each method in the interface. The function *Sig* takes a method as argument and returns its signature, including the cointerface as an explicit in-parameter. If new superclasses are added, the inheritance list in Γ_C must be extended accordingly before type checking method bodies, as there might be internal calls to methods in the new superclasses. This also applies to methods, due to calls to methods introduced in the same upgrade. The function *matchpar* verifies that the formal and actual parameters of new inherited classes match, and that these classes are contained in the class mapping Γ_C. Inherited attributes, as well as new object variables, will further extend the typing environment. For each method, the effect system of rule (METHOD) computes the dependencies associated with the method body. Finally, after the type analysis of the upgrade term of a class C, the Γ_C mapping is upgraded and the dependency mapping for the $(\Gamma_C(C).\mathit{Upg} + 1)$'th upgrade of class C is constructed, which is a mapping from $\langle C, \Gamma_C(C).\mathit{Upg} + 1 \rangle$ to the dependencies identified by the type analysis of the upgrade term. For method redefinition, the rule (MTD-RDEF) ensures that the redefined method still satisfies the interface requirements implemented by the class. For purely internal methods, the new cointerface must be *Internal*.

At runtime, upgrades are asynchronous and may bypass each other. Hence, well-typed upgrades may give runtime errors if not applied in a type-correct order. We show that Γ_d, provided by the type system, helps to ensure that each upgrade is applied at an appropriate time: If both a class C' and a superclass C are updated, then upgrades will be applied at runtime in the order decided by the static type system, e.g., C is upgraded first if the upgrade of C' depends on the upgrade of C. However, upgrades that do not depend on each other may be applied in parallel. It is therefore necessary that $\Gamma_d(\langle C, u \rangle)$ is included as an argument to the runtime class upgrade $\langle C, u \rangle$. This is achieved by translating the update term $\mathit{upd}(C, \mathit{Imp}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd})$ into the runtime message $\mathit{upgrade}(C, \mathit{Inh}, \mathit{Var}, \mathit{Mtd}, \Gamma_d(\langle C, \Gamma_C(C).\mathit{Upg} \rangle))$ where Γ is the environment obtained from type checking the update term. Note that the implements-clause is not needed after type checking.

9.5.2 Operational Semantics

The operational semantics of Creol is defined in rewriting logic (RL) [19] and is executable on the RL system Maude [7]. A rewrite theory is a 4-tuple (Σ, E, L, R) where the signature Σ defines the function symbols, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. Rewrite rules apply to terms of given sorts. Sorts are specified in (membership) equational logic (Σ, E). When modeling computational systems, different system components are typically modeled by terms of the different sorts defined in the equational logic. The global state configuration is defined as a multiset of these terms. RL extends algebraic specification techniques with transition rules: The dynamic behavior of a system is captured by rewrite rules supplementing the equations which define the term language. From a computational viewpoint, a rewrite rule $t \longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t'. When auxiliary functions are needed in the semantics, these are defined in equational logic, and are evaluated in between the state transitions [19]. If rewrite rules apply to non-overlapping sub-configurations, the transitions may be performed in parallel. Consequently, concurrency is implicit in RL. Conditional rewrite rules $t \longrightarrow t'$ **if** *cond* are allowed, where the condition *cond* can be formulated as a conjunction of rewrites and equations that must hold for the main rule to apply.

A *system configuration* is a multiset combining Creol classes, objects, and messages. A Creol method call is reflected by a pair of messages, and object activity is organized around a *message queue* which contains incoming messages and a *process queue* which contains pending processes, i.e., remaining parts of method activations. The associative list constructor is written as ';', and the associative and commutative constructors for multisets and sets by whitespace. Representing argument positions by "_", terms $\langle \_ : Ob \,|\, Cl : \_, Pr : \_, PrQ : \_, Lvar : \_, Att : \_, Qu : \_ \rangle$ denote Creol objects, where *Ob* is the object identifier, *Cl* the *class identifier* which consists of a class name and *version number*, *Pr* the active process code, *PrQ* and *Qu* are multisets of pending processes and incoming messages with unspecified queue orderings, respectively, and *Lvar* and *Att* the local and object state, respectively. Terms $\langle \_ : Cl \,|\, Upd : \_, Inh : \_, Att : \_, Mtds : \_ \rangle$ represent Creol classes, where *Cl* is the class identifier, *Upd* the upgrade number, *Inh* a list of class identifiers, *Att* a list of attributes, and *Mtds* a set of methods. The class identifier for version n of class C is denoted $C\#n$. The rules for the static language constructs may be found in [14]. Focus here is on method binding and dynamic class constructs, given in Figure 9.5.

An *implicit inheritance graph* is used to facilitate dynamic reconfiguration mechanisms. The binding mechanism dynamically inspects the class hierarchy in the system configuration. When an invocation message $invoc(m, Sig, In)$ representing a call to a method m is found in the message queue of an object o of class $C\#n$, where *Sig* is the method signature as provided by the caller and *In* is the list of actual in-parameters, a message $bind(o, m, Sig, In)$ **to** $C\#n$ is generated. If m is defined locally in $C\#n$ with a matching signature, a process with the declared method code and local state is returned in a *bound* message. Otherwise, the *bind* message is retransmitted to the superclasses of C. Thus the *bind* message is sent from a class

$$
\begin{array}{l}
\langle o\!:\!Ob \,|\, Cl : C\#n\rangle \; \langle o\!:\!Qu \,|\, Ev : \textsc{q}\; invoc(m,Sig,In)\rangle \\
\quad \longrightarrow \langle o\!:\!Ob \,|\, Cl : C\#n\rangle \; \langle o\!:\!Qu \,|\, Ev : \textsc{q}\rangle \; (bind(o,m,Sig,In) \;\mathbf{to}\; C\#n)
\end{array}
$$

$$
\begin{array}{l}
bind(o,m,Sig,In) \;\mathbf{to}\; \varepsilon \longrightarrow bound(none) \;\mathbf{to}\; o \\
bind(o,m,Sig,In) \;\mathbf{to}\; (C\#n);\textsc{i}' \; \langle C\#n'\!:\!Cl \,|\, Inh : \textsc{i}, Mtds : \textsc{m}\rangle \\
\quad \longrightarrow \mathbf{if}\; match(m,Sig,\textsc{m}) \;\mathbf{then}\; (bound(get(m,\textsc{m},In)) \;\mathbf{to}\; o) \\
\qquad\qquad\qquad\qquad\qquad \mathbf{else}\; (bind(o,m,Sig,In) \;\mathbf{to}\; \textsc{i};\textsc{i}') \;\mathbf{fi} \\
\qquad \langle C\#n\!:\!Cl \,|\, Inh : \textsc{i}, Mtds : \textsc{m}\rangle
\end{array}
$$

$$
(bound(w) \;\mathbf{to}\; o) \; \langle o\!:\!Ob \,|\, PrQ : \textsc{w}\rangle \longrightarrow \langle o\!:\!Ob \,|\, PrQ : w\;\textsc{w}\rangle
$$

$$
\begin{array}{l}
new\text{-}class(C,\textsc{i},\textsc{a},\textsc{m},((C'\#n)\;\textsc{r})) \; \langle C'\#n'\!:\!Cl \,|\, Upd\!:\!u\rangle \\
\quad \longrightarrow new\text{-}class(C,\textsc{i},\textsc{a},\textsc{m},\textsc{r}) \; \langle C'\#n'\!:\!Cl \,|\, Upd\!:\!u\rangle \;\mathbf{if}\; u \geq n
\end{array}
$$

$$
new\text{-}class(C,\textsc{i},\textsc{a},\textsc{m},\varepsilon) \longrightarrow \langle C\#1\!:\!Cl \,|\, Upd\!:\!1, Inh : \textsc{i}, Att : \textsc{a}, Mtds : \textsc{m}, Tok : 1\rangle
$$

$$
\begin{array}{l}
upgrade(C,\textsc{i},\textsc{a},\textsc{m},((C'\#n)\;\textsc{r})) \; \langle C'\#n'\!:\!Cl \,|\, Upd\!:\!u\rangle \\
\quad \longrightarrow upgrade(C,\textsc{i},\textsc{a},\textsc{m},\textsc{r}) \; \langle C'\#n'\!:\!Cl \,|\, Upd\!:\!u\rangle \;\mathbf{if}\; u \geq n
\end{array}
$$

$$
\begin{array}{l}
upgrade(C,\textsc{i}',\textsc{a}',\textsc{m}',\emptyset) \; \langle C\#n\!:\!Cl \,|\, Upd\!:\!u, Inh : \textsc{i}, Att : \textsc{a}, Mtds : \textsc{m}, Tok : T\rangle \\
\quad \longrightarrow \langle C\#(n+1)\!:\!Cl \,|\, Upd\!:\!u+1, Inh : \textsc{i};\textsc{i}', Att : \textsc{a};\textsc{a}', Mtds : \textsc{m} \oplus \textsc{m}', Tok : T\rangle
\end{array}
$$

$$
\begin{array}{l}
\langle C\#n\!:\!Cl \,|\, Inh : \textsc{i};(C'\#n');\textsc{i}'\rangle \; \langle C'\#n''\!:\!Cl \,|\, \rangle \\
\quad = \langle C\#(n+1)\!:\!Cl \,|\, Inh : \textsc{i};(C'\#n'');\textsc{i}'\rangle \; \langle C'\#n''\!:\!Cl \,|\, \rangle \;\mathbf{if}\; n'' > n'
\end{array}
$$

$$
\begin{array}{l}
\langle o\!:\!Ob \,|\, Cl : C\#n, Pr : \varepsilon\rangle \langle C\#n'\!:\!Cl \,|\, Att : \textsc{a}\rangle \\
\quad = \langle o\!:\!Ob \,|\, Cl : C\#n', Pr : \varepsilon\rangle \; \langle C\#n'\!:\!Cl \,|\, Att : \textsc{a}\rangle \; (getAttr(o,\textsc{a}) \;\mathbf{to}\; C) \;\mathbf{if}\; n' > n
\end{array}
$$

$$
(gotAttr(\textsc{a}') \;\mathbf{to}\; o) \; \langle o\!:\!Ob \,|\, Att : \textsc{a}\rangle = \langle o\!:\!Ob \,|\, Att : \textsc{a}'\rangle
$$

Figure 9.5: A RL specification of method binding and dynamic class upgrades.

to its superclasses, dynamically unfolding the inheritance graph as far as needed and resulting in a *bound* message returned to the object which generated the *bind* message. The auxiliary predicate $match(m, Sig, \textsc{m})$ is true if m is declared in $\textsc{m}$ with a signature Sig' such that $Sig' \preceq Sig$, and the function *get* fetches method m in the method set $\textsc{m}$ of the class and returns a process, resulting from the method activation. Values of the actual in-parameters *In* are stored in the local process state. The process is loaded into the internal process queue of the callee.

Class upgrades may be direct, or indirect through the upgrade of one of the superclasses. In order to control the upgrade propagation, class representations include an *upgrade number* and a *version number*; i.e., counters which record the number of times a class has been directly upgraded and (directly or indirectly) modified, respectively. When a class is upgraded, both its upgrade and version numbers are incremented. When a super-class of a class C is modified, the version number of C is incremented but the upgrade number of C does not change.

A *direct class upgrade* of a class C is realized through the insertion of a message $upgrade(C,\textsc{i},\textsc{a},\textsc{m},\Gamma_d(\langle C,\Gamma_C(C).Upg\rangle))$ in the system configuration at runtime, where $\textsc{i}$ is an inheritance list, $\textsc{a}$ a state, $\textsc{m}$ a set of method definitions, and $\Gamma_d(\langle C,\Gamma_C(C).Upg\rangle)$ the set of upgrade requirements to classes in the runtime system directly derived from Γ, found by type checking. The upgrade of a class may not be applied unless these requirements are fulfilled. As upgrade is asynchronous, several upgrades may be pending in the runtime system, and the current upgrade may need to wait. A message $upgrade(C,\textsc{i}',\textsc{a}',\textsc{m}',\varepsilon)$, with an empty requirement set, does not have unverified dependencies, and the upgrade may be applied to C. The rule for *direct class upgrade* uses an operator $\oplus$ to overwrite the method set $\textsc{m}$ with the new or redefined methods in $\textsc{m}'$. During the upgrade, the upgrade and version numbers of the class are also incremented.

Indirect class upgrade propagates upgrade information to subclasses by means of an equation, so instances of the subclasses will acquire new state attributes. Note that by using an equation the indirect class upgrade happens in zero rewrite steps, which corresponds to temporarily locking the upgraded class.

The *upgrade of object instances* must ensure that new attributes are acquired before new code which may rely on new class attributes is evaluated. New object instances automatically get the new class attributes. However, the upgrade of existing object instances of the class must be closely controlled. Each time an object needs to evaluate a method, it requests the code associated with this method name. Problems may arise when executing new or redefined methods which rely on new attributes that are not presently available in the object. With recursive or nonterminating processes, objects cannot generally be expected to reach a state without pending processes, even if the loading of processes corresponding to new method calls from the environment is postponed as in [8,2]. Consequently, it is too restrictive to wait for the completion of all pending methods before applying an upgrade. However, objects may reach *quiescent* states when the processor has been released and before any pending process has been activated. Any object which does not deadlock will eventually reach a quiescent state. In particular nonterminating activity is implemented by means of recursion, which ensures at least one quiescent state in each cycle. In the case of process termination or an inner suspension point, *Pr* is empty. The rule for *object upgrade* applies to quiescent states. Exploiting the implicit inheritance graph, attribute upgrade is handled by a message *getAttr*, similar to *bind*, which recursively extends the object state $\textsc{a}$ and results in a message $gotAttr(\textsc{a}')$. The new object state $\textsc{a}'$ finally replaces $\textsc{a}$. The use of equations corresponds to locking the object.

The described runtime mechanism allows the upgrade of active objects. Attributes are collected at upgrade time while code is loaded “on demand”. A class may be upgraded several times before the object reaches a quiescent state, so the object may have missed some upgrades. However a single state upgrade suffices to ensure that the object, once upgraded, is a complete instance of the present version of its class. The upgrade mechanism ensures that an object upgrade has occurred before new code is evaluated.

9.5.3 Type-Safe Execution with Dynamic Class Upgrades

The problem of type-safe execution of programs is now addressed. We prove that errors such as method-not-understood do not occur at runtime, even with the proposed dynamic class construct. A type soundness theorem for Creol without dynamic classes was shown in [16]: *Runtime type errors do not occur for well-typed programs.* The theorem implies that runtime assignments to program variables, object creation, and method invocations are type-correct. The proof is by induction over the length of the execution sequence as given by the operational semantics. However, dynamic upgrades as considered in this paper introduce runtime changes as the state adapts to the upgrades. By reasoning about the type system and operational semantics, the following properties are proved for the class upgrade mechanism of this paper:

Lemma 1 *A well-typed class upgrade does not affect the execution of code of existing processes in an object.*

Lemma 2 *The execution of a method activation from a new version of an object's class will not begin before the object's state is updated.*

Lemma 3 *Let Γ be the typing environment for a well-typed program after a series of upgrades, including the upgrade $\langle C, u \rangle$. The upgrade $\langle C, u \rangle$ is applicable iff the runtime structure satisfies $\Gamma_d(\langle C, u \rangle)$.*

Lemma 4 *The execution of processes introduced in a well-typed upgrade will not cause runtime type errors.*

Lemma 4 follows from Lemmas 2 and 3. Lemmas 1 and 4 show that variable assignments, object creation, and method invocations are type-correct when classes are upgraded, for old and new processes, respectively. A type soundness property for Creol with class upgrades can now be proved by induction over the length of execution sequences, extending the proof for the language without dynamic classes. Lemmas 1 and 4 are used for the application of class upgrades:

Theorem 5 (Type soundness) *Well-typed upgrades do not introduce runtime type errors in well-typed programs.*

9.6 Related Work

Availability during reconfiguration is an essential feature of many modern distributed applications. Dynamic or online system upgrade considers how running systems may evolve. Recently, several authors have investigated type-safe mechanisms for runtime upgrade of imperative [23], functional [4], and object-oriented [9] languages. These approaches consider the upgrade of single type declarations, procedures, objects, or components in the sequential setting. Reclassification in Fickle [9] is based on a type system which guarantees type safety when an object changes its class. Fickle has been extended to multithreading [8], but restrictions to runtime reclassification are needed; e.g., an object with a nonterminating (recursive) method will not be reclassified.

Version control systems aim at a more modular upgrade support. Some approaches allow multiple module versions to coexist after an upgrade [4, 3, 10, 11, 12], while others only keep the last version by doing a global update or "hot-swapping" [20, 18, 2, 6]. The approaches also differ in their treatment of active behavior, which may be disallowed [20, 18, 11, 6], delayed [8, 2], or supported [23, 12]. Approaches based on global update mostly disallow upgrades of active modules. An upgrade system for type declarations and procedures in active code is proposed in [23] for (sequential) C. Type-safe updates occur at annotated program points found by the type system. However, the approach is synchronous as upgrades which cannot be applied immediately will fail.

Dynamic class constructs support modular upgrades. The approach of Hjálmtýsson and Gray [12] for C++, based on proxy classes which link to the actual classes (reference indirection), supports multiple versions of each class. Existing instances are not upgraded, so the activity in existing objects is uninterrupted. Existing approaches for Java, either using proxies [20] or modifying the Java virtual machine [18], are based on global upgrade and are not applicable to active objects. In [20], each class version supports the same interfaces. New interfaces can only be introduced by adding new classes. In [6] the ordering of upgrades is serialized and in [18] invalid upgrades are handled by exceptions.

Automatic upgrade based on lazy global update is addressed in [2] for distributed objects and in [6] for persistent object stores. Here the object instances of upgraded classes are upgraded, but inheritance and (nonterminating) active code are not addressed, which limits the effect of class upgrade. Our approach supports multiple inheritance, but restricts upgrades to addition and redefinition and may therefore avoid these limitations. Only one version of an upgraded class is kept in the system but active objects may still be upgraded. Upgrade is asynchronous and distributed, and may therefore be temporarily delayed. Moreover, the type system handles upgrade dependencies among distributed objects.

9.7 Conclusion

In this paper a construct for dynamic class upgrades in Creol is presented, including its type system and operational semantics, which allows method redefinition as well as extending classes with new attributes, methods, superclasses, and interfaces, in the running system. By adding new interfaces, classes may provide new external services, while the redefinition of methods may improve existing ones. Our approach exempts programmers from handling the different version numbers of classes when writing upgrade codes.

To address open distributed systems with concurrent objects, we consider an asynchronous update mechanism where upgrade overtaking is possible in the runtime system, and allow objects of different versions to coexist. A successful introduction of upgrades in this setting requires both type checking and careful timing of when the upgrades are applied. Runtime errors would occur if upgrades are applied at a bad time. The type system captures upgrade dependencies and enforces an ordering of upgrades. If the type checking of an upgrade succeeds, an *effect* system provides a list of dependencies for the upgrade. This list of dependencies is used by the runtime system to ensure that dependent upgrades are applied in an order which preserves type correctness, while independent upgrades may be performed simultaneously. Furthermore, it is shown that well-typed runtime upgrades do not introduce type errors. In future work we plan to extend the dynamic construct proposed in this paper with type-safe mechanisms for removing attributes and method definitions, using similar techniques.

References

[1] *Proceedings of the General Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*. USENIX, 2003.

[2] S. Ajmani, B. Liskov, and L. Shrira. Scheduling and simulation: How to upgrade distributed systems. In *9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, pages 43–48, Lihue, Hawaii, May 2003.

[3] J. L. Armstrong and S. R. Virding. Erlang - an experimental telephony programming language. In *XIII International Switching Symposium*, June 1990.

[4] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating. In *Proc. 2nd Intl. Workshop on Unanticipated Software Evolution (USE)*, April 2003.

[5] T. Bloom. *Dynamic Module Replacement in a Distributed Programming System.* PhD thesis, MIT, 1983. Also available as MIT LCS Tech. Report 303.

[6] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In R. Crocker and G. L. S. Jr., editors, *Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, pages 403–417. ACM Press, 2003.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.

[8] F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Re-classification and multithreading: Fickle$_{\mathrm{MT}}$. In *Proc. Symposium on Applied Computing (SAC'04)*, volume 2, pages 1297–1304. ACM Press, 2004.

[9] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: Fickle$_{\mathrm{II}}$. *ACM Transactions on Programming Languages and Systems*, 24(2):153–191, 2002.

[10] D. Duggan. Type-Based hot swapping of running modules. In C. Norris and J. J. B. Fenwick, editors, *Proc. 6th Intl. Conf. on Functional Programming (ICFP'01)*, volume 36, 10 of *ACM SIGPLAN notices*, pages 62–73, New York, Sept. 3–5 2001. ACM Press.

[11] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, 1996.

[12] G. Hjálmtýsson and R. S. Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proc. USENIX Tech. Conf. (USENIX '98)*, May 1998.

[13] C. R. Hofmeister and J. M. Purtilo. A framework for dynamic reconfiguration of distributed programs. Technical Report CS-TR-3119, University of Maryland, College Park, 1993.

[14] E. B. Johnsen and O. Owe. A dynamic binding strategy for multiple inheritance and asynchronously communicating objects. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. 3rd International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *Lecture Notes in Computer Science*, pages 274–295. Springer-Verlag, 2005.

[15] E. B. Johnsen, O. Owe, and I. Simplot-Ryl. A dynamic class construct for asynchronous concurrent objects. In M. Steffen and G. Zavattaro, editors, *Proc. 7th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, volume 3535 of *Lecture Notes in Computer Science*, pages 15–30. Springer-Verlag, June 2005.

[16] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. Research Report 327, Department of Informatics, University of Oslo, June 2005. Revised May 2006. To appear in *Theoretical Computer Science*.

[17] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, Nov. 1990.

[18] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In E. Bertino, editor, *Proc. 14th European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, pages 337–361. Springer-Verlag, June 2000.

[19] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[20] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. In *Proc. International Conference on Software Maintenance (ICSM'02)*, pages 649–658. IEEE Computer Society Press, Oct. 2002.

[21] B. C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Mass., 2002.

[22] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. D. Silva, G. R. Ganger, O. Krieger, M. Stumm, M. A. Auslander, M. Ostrowski, B. S. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *USENIX Annual Technical Conference, General Track* [1], pages 141–154.

[23] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, pages 183–194, Jan. 2005.

Chapter 10

Paper 4: Dynamic Classes: Modular Asynchronous Evolution of Distributed Concurrent Objects

Einar Broch Johnsen, Marcel Kyas, and Ingrid Chieh Yu

Publication.
Published as Research Report 383, Department of Informatics, University of Oslo, May 2009. A shorter version of this report is published in the Proceedings of the 16th International Symposium on Formal Methods (FM 2009).

Abstract.
Many long-lived and distributed systems must remain available yet evolve over time, due to, e.g., bugfixes, feature extensions, or changing user requirements. To facilitate such changes, formal methods can help in modeling and analyzing runtime software evolution. This paper presents an executable object-oriented modeling language which supports runtime software evolution. The language, based on Creol, targets distributed systems by active objects, asynchronous method calls, and futures. A dynamic class construct is proposed in this setting, providing an asynchronous and modular upgrade mechanism. At runtime, class redefinitions gradually upgrade existing instances of a class and of its subclasses. An upgrade may depend on previous upgrades of other classes. For asynchronous runtime upgrades, the static picture may differ from the actual runtime system. An operational semantics and a type and effect system are given for the language. The type analysis of an upgrade infers and collects dependencies on previous upgrades. These dependencies are exploited as runtime constraints to ensure type safety.

10.1 Introduction

Many long-lived distributed systems require continuous system availability, but still need to change their code due to bugfixes as well as new, improved, or redundant functionality. Examples of such systems are found in, e.g., financial transactions, aeronautics and space missions, biomedical sensors, and telephony and Internet services. For these systems, code changes must happen at runtime. In large distributed systems, runtime updates need to be applied in an asynchronous and modular manner, and propagate gradually through the distributed system. A challenge for software upgrade systems is to balance flexibility, robustness, and user-friendliness. An appropriate upgrade system should propagate upgrades automatically, provide means to control *when* components are upgraded, and ensure the availability of system services during the upgrade [2, 23]. In order to ensure that upgrades are correct and result in foreseen changes, formal models and analysis methods for runtime software evolution are needed.

This paper presents a modeling language which supports the runtime evolution of distributed object-oriented systems. The language extends Creol [19], an executable formalism in which distributed concurrent objects communicate by asynchronous method calls and futures [10, 16, 26, 8, 21], with *dynamic class operations.* These may introduce new functionality and interfaces for classes, change data structures and implementations for existing functionality, and remove legacy code. Dynamic class operations provide a *modular* form of software evolution because upgrades to a class *C* apply to all existing instances of *C* and of its subclasses. Compared to previous approaches [22, 20, 15, 7], our approach supports the gradual upgrade of active objects and upgrade operations propagate asynchronously through the system. It is a challenge for formal methods to reason about the runtime evolution of distributed systems. This paper focuses on ensuring type safety at runtime as class definitions evolve. For example, if a class is asynchronously extended with a new method at runtime, calls to that method on an existing object must not occur before the method is available, new data structures must be available before methods attempt to use them, and fields in an object cannot be removed while methods may still access them.

To this end, an operational semantics and type system for the proposed dynamic class operations are introduced and integrated with the semantics and type system of Creol. Creol is type-safe in the sense that runtime type errors do not occur for well-typed programs; in particular, method binding always succeeds. We show that well-typed dynamic class operations maintain this property. As classes gradually evolve, a type-safe upgrade of one class may require that an upgrade of another class has already been applied; e.g., when new code contains calls to methods introduced in a previous upgrade. Upgrades may be arbitrarily delayed in the asynchronous setting, so upgrades injected into the system in one order may be applied in another. This causes a discrepancy between the static system view, as provided by a typing environment for dynamic class operations, and the situation at runtime. We develop a type and effects system [25, 3] to analyze dynamic class upgrades and to automatically infer and collect dependencies between class upgrades. Thus, the dependencies of an upgrade operation need not be provided by the modeler. A characterizing feature of our approach is that the dependencies inferred during type analysis are imposed as constraints on the applicability of a particular upgrade at runtime. This may delay certain upgrade operations at runtime to ensure that execution remains type safe. This paper presents the proposed dynamic class operations for a kernel language, but the approach is supported in the complete Creol language. The type system and operational semantics of this paper have been implemented and integrated with Creol's execution platform.

Paper overview. Sect. 10.2 presents the kernel language, its type system, and semantics. Sect. 10.3 provides an example of dynamic class upgrades. Sect. 10.4 introduces operations for dynamic class upgrades, their type system, and semantics. Sect. 10.5 discusses related work, and Sect. 10.6 concludes the paper.

10.2 A Language for Distributed Concurrent Objects

A kernel language for distributed concurrent objects is now introduced, similar to, e.g., Featherweight Java [18]. The language targets distributed systems by supporting asynchronous method calls and futures (i.e., returns from asynchronous calls). In contrast to Java each concurrent object encapsulates its state; i.e., all external manipulation of the object state is through calls to the object's methods. In addition, objects execute concurrently: each object has a processor which executes the processes of that object. Processes in different objects execute in parallel. A process corresponds to the activation of one of the object's method. Only one process may be active in an object at a time; the other processes in the object are *suspended*. We distinguish between *blocking* a process and *releasing* a process. Blocking is used for synchronization and causes the execution of the process to stop, but does not allow a suspended process resume. Releasing a process suspends the execution of that process and lets another (suspended) process resume. Thus, if a process is blocked there is no execution in the object, whereas if a process is released another process in the object may execute. Although processes need not terminate, the execution of several processes may be combined using *release points* within method bodies. At a release point, the active process may be released and a suspended process may resume.

Method calls are asynchronous and the result of a call is stored in a future [10, 16, 26, 8, 21], which may be read or polled. Return values are accessed by need; i.e., the execution blocks if attempting to read from a future without a return value. In contrast to read, polling a future never blocks. The scheduling of processes at release points is influenced by await-statements with Boolean guards, including the polling of futures. If a guard evaluates to false, the process is released. Only a process whose guard evaluates to true may resume execution. Remark that this use of release points make it straightforward to combine active (i.e., nonterminating) and reactive processes in an object. Thus, an object may behave both as a client and as a server while abstracting from the exact interleaving of these roles.

The behavior of an object may depend on its context of interaction; we let object variables (references) be typed by *interfaces*. These contain method signatures and provide context-dependent encapsulation, as different sets of methods may be available through different interfaces. Variables typed by different interfaces may refer to the same object. A class *implements* an interface *I* if its instances may be typed by *I*. A class may implement several interfaces and different classes may provide different implementations of the same interface. Reasoning control is ensured by substitutability at the level of interfaces: *an object supporting an interface I may be replaced by another object supporting I or a subinterface of I* in a context depending on *I*. This substitutability is reflected in the semantics by the fact that late binding applies to all external method calls, as the runtime class of the object reference is not in general statically

$$\begin{array}{lcl@{\qquad}lcl}
P & ::= & \overline{D}\,\overline{L}\,\{\overline{T\,x};sr\} & D & ::= & \textsf{interface}\ I\ \textsf{extends}\ \overline{I}\ \{\overline{M_s}\} \\
sr & ::= & s;\textsf{return}\ e & L & ::= & \textsf{class}\ C\ \textsf{extends}\ \overline{C}\ \textsf{implements}\ \overline{I}\ \{\overline{T\,f};\overline{M}\} \\
v & ::= & f \mid x & M & ::= & M_s\{\overline{T\,x};sr\} \\
b & ::= & \textsf{true} \mid \textsf{false} \mid v & e & ::= & v \mid \textsf{new}\ C() \mid e.\textsf{get} \mid e!m(\overline{e}) \mid \textsf{null} \\
T & ::= & I \mid \textsf{bool} \mid \textsf{fut}(T) & s & ::= & v := e \mid \textsf{await}\ g \mid \textsf{skip} \mid s;s \mid \textsf{if}\ g\ \textsf{then}\ s\ \textsf{fi} \mid \textsf{release} \\
M_s & ::= & T\ m\ (\overline{T\,x}) & g & ::= & b \mid v? \mid g \wedge g
\end{array}$$

Figure 10.1: The language syntax. Variables v are fields (f) or local variables (x), C is a class name, and I an interface name.

known.

Syntax. The syntax of Creol is given in Fig. 10.1. We emphasize the differences with Java. A program P is a list of interface and class definitions, followed by a method body. In order to illustrate the generality of the dynamic class construct, we let a class inherit from a list of superclasses (which may be just Object), extending these with additional fields $\overline{f}$ and methods $\overline{M}$. *Expressions e* are standard apart from the asynchronous method call $e!m(\overline{e})$ and the (blocking) read operation v.get. *Statements s* are standard apart from release points await g and release. *Guards g* are conjunctions of Boolean expressions b and polling operations $v?$ on futures v. When the guard in an await statement evaluates to false, the statement gets preceded by a release, otherwise it becomes a skip. The release statement suspends the active process and another suspended process may be rescheduled.

10.2.1 Typing

Type analysis is done by a type and effect system [25, 3] in the context of a *mapping family*, defined as follows:

Definition 1 Let n be a name, d a declaration, $i \in I$ a mapping index, and $[n \mapsto_i d]$ the binding of n to d indexed by i. A *mapping family* Γ is built from the empty mapping family $\emptyset$ and indexed bindings by the constructor $+$. The *extraction* of an indexed mapping Γ_i from Γ and *application* for the indexed mapping Γ_i, are defined as follows

$$\begin{array}{lcl}
\emptyset_i & = & \varepsilon \\
(\Gamma + [n \mapsto_{i'} d])_i & = & \textbf{if}\ (i = i')\ \textbf{then}\ \Gamma_i + [n \mapsto_i d]\ \textbf{else}\ \Gamma_i \\
\\
\varepsilon(n) & = & \perp \\
(\Gamma_i + [n \mapsto_i d])(n') & = & \textbf{if}\ (n = n')\ \textbf{then}\ d\ \textbf{else}\ \Gamma_i(n').
\end{array}$$

The typing context uses four indexes; the mappings Γ_I and Γ_C map interface and class names to interface and class declarations, and Γ_V maps program variable names to types. In the absence of class upgrades, Γ_I and Γ_C correspond to *static tables*. The subtype relation $T_1 \preceq T_2$ is defined by interface inheritance. Declarations may only extend Γ_V and statements may not update Γ_V. Some auxiliary functions are defined on a mapping family Γ. The field declarations in a class C and its superclasses are collected by $attr(C, \Gamma)$ and $implements(C, I, \Gamma)$ matches signatures for methods declared in an interface I to those in C, in order to check that

$$\overset{(\textsc{Poll})}{\frac{\Gamma \vdash v : \mathsf{fut}(T)\,\langle\Sigma\rangle}{\Gamma \vdash v? : \mathsf{bool}\,\langle\Sigma\rangle}} \qquad \overset{(\textsc{Get})}{\frac{\Gamma \vdash v : \mathsf{fut}(T)\,\langle\Sigma\rangle}{\Gamma \vdash v.\mathsf{get} : T\,\langle\Sigma\rangle}} \qquad \overset{(\textsc{New})}{\frac{\exists T' \in \mathit{interfaces}(\Gamma_C(C)) \cdot T' \preceq T}{\Gamma \vdash \mathsf{new}\ C(\,) : T\,\langle(C, \mathit{curr}(C,\Gamma))\rangle}} \qquad \overset{(\textsc{Skip})}{\frac{}{\Gamma \vdash \mathsf{skip}}}$$

$$\overset{(\textsc{IntCall})}{\frac{\begin{array}{c}\Gamma \vdash \overline{e} : T\,\langle\Sigma\rangle \\ \exists C \in \mathit{matchint}(m, T \to T', \Gamma_{\mathrm{V}}(\mathsf{this}), \Gamma_C)\end{array}}{\Gamma \vdash \mathsf{this}!m(\overline{e}) : \mathsf{fut}(T')\,\langle\Sigma \cup (C, \mathit{curr}(C,\Gamma))\rangle}} \qquad \overset{(\textsc{ExtCall})}{\frac{\begin{array}{c}\Gamma \vdash \overline{e} : T\,\langle\Sigma_1\rangle \quad \Gamma \vdash e : I\,\langle\Sigma_2\rangle \\ \mathit{matchext}(m, T \to T', I, \Gamma_I)\end{array}}{\Gamma \vdash e!m(\overline{e}) : \mathsf{fut}(T')\,\langle\Sigma_1 \cup \Sigma_2\rangle}} \qquad \overset{(\textsc{Null})}{\frac{I \in \mathit{dom}(\Gamma_I)}{\Gamma \vdash \mathsf{null} : I}}$$

$$\overset{(\textsc{Var})}{\frac{\Gamma(v) = T}{\Gamma \vdash v : T\,\langle[\![v]\!]\rangle}} \qquad \overset{(\textsc{Assign})}{\frac{\Gamma \vdash e : T'\,\langle\Sigma\rangle \qquad T' \preceq \Gamma_{\mathrm{V}}(v)}{\Gamma \vdash v := e\,\langle[\![v]\!] \cup \Sigma\rangle}} \qquad \overset{(\textsc{And})}{\frac{\Gamma \vdash g_1 : \mathsf{bool}\,\langle\Sigma_1\rangle \quad \Gamma \vdash g_2 : \mathsf{bool}\,\langle\Sigma_2\rangle}{\Gamma \vdash g_1 \wedge g_2 : \mathsf{bool}\,\langle\Sigma_1 \cup \Sigma_2\rangle}}$$

$$\overset{(\textsc{Await})}{\frac{\Gamma \vdash g : \mathsf{bool}\,\langle\Sigma\rangle}{\Gamma \vdash \mathsf{await}\ g\,\langle\Sigma\rangle}} \qquad \overset{(\textsc{Release})}{\frac{}{\Gamma \vdash \mathsf{release}}} \qquad \overset{(\textsc{Composition})}{\frac{\Gamma \vdash s\,\langle\Sigma_1\rangle \quad \Gamma \vdash s'\,\langle\Sigma_2\rangle}{\Gamma \vdash s; s'\,\langle\Sigma_1 \cup \Sigma_2\rangle}} \qquad \overset{(\textsc{Conditional})}{\frac{\Gamma \vdash b : \mathsf{bool}\,\langle\Sigma_1\rangle \quad \Gamma \vdash s\,\langle\Sigma_2\rangle}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{fi}\,\langle\Sigma_1 \cup \Sigma_2\rangle}}$$

$$\overset{(\textsc{Method})}{\frac{\begin{array}{c}\Gamma' = \Gamma + [\overline{x} \mapsto_v \overline{T}] + [\overline{x'} \mapsto_v \overline{T'}] \\ \Gamma' \vdash e : T'\,\langle\Sigma_1\rangle \quad \Gamma' \vdash s\,\langle\Sigma_2\rangle\end{array}}{\Gamma \vdash T'\ m\ (\overline{T\ x})\{\overline{T'\ x'}; s; \mathsf{return}\ e\}\,\langle\Sigma_1 \cup \Sigma_2\rangle}} \qquad \overset{(\textsc{Program})}{\frac{\begin{array}{c}\Gamma + [\overline{x} \mapsto_v \overline{T}] \vdash s \\ \forall L \in \overline{L} \cdot \Gamma_I + \Gamma_C + \Gamma_d^L \vdash L\end{array}}{\Gamma_I + \Gamma_C + \bigcup_{L \in \overline{L}} \Gamma_d^L \vdash \overline{L}\ \{\overline{T\ x}; s; \mathsf{return}\ \mathit{true}\}}}$$

$$\overset{(\textsc{Class})}{\frac{\forall M \in \overline{M}\ \cdot\ \Gamma + [\mathsf{this} \mapsto_v C] + [\mathit{attr}(C,\Gamma)] \vdash M\,\langle\Sigma^M\rangle \quad \forall I \in \overline{I} \cdot \mathit{implements}(C, I, \Gamma)}{\Gamma + [\langle C, 0\rangle \mapsto_d \bigcup_{M \in \overline{M}} \Sigma^M] \vdash \mathsf{class}\ C\ \mathsf{extends}\ \overline{C}\ \mathsf{implements}\ \overline{I}\ \{\overline{T\ f}; \overline{M}\}}}$$

Figure 10.2: The type and effect system. Judgments for the Boolean constants true and false are similar to (Release). We omit empty effects; e.g., $\Gamma \vdash e\,\langle\emptyset\rangle$ is written $\Gamma \vdash e$.

the class provides method bodies for the method declarations of its interfaces. We assume for simplicity that variable declarations $\overline{T\ x}$ are well-typed and denote by $[\overline{x} \mapsto_v \overline{T}]$ the associated mapping (built from the bindings $[x \mapsto_v T]$).

For the purposes of dynamic upgrades, there is a mapping of *dependencies* $\Gamma_d : \mathsf{Dep} \to \mathsf{Set}[\mathsf{Dep}]$, where the type Dep consist of pairs of class names and natural numbers. An upgrade of a class C can be uniquely identified by a natural number; e.g., $\langle C, 5\rangle$ represents the fifth upgrade of C. Elements in $\Gamma_d(\langle C, u\rangle)$ will represent classes on which an upgrade u of a class C depends; these dependencies are inferred from the current class table by the type analysis, and exploited for dynamic classes in Sect. 10.4.

The type rules are given in Fig. 10.2. Judgments have the form $\Gamma \vdash e : T\,\langle\Sigma\rangle$ and $\Gamma \vdash s\,\langle\Sigma\rangle$, where Γ is the typing environment and $\Sigma : \mathsf{Set}[\mathsf{Dep}]$ the effect. To simplify the presentation, we assume that method declarations in interfaces are unique and well-typed and omit the analysis of interfaces. Furthermore, the (straightforward) definitions of auxiliary functions on Γ are omitted.

$$
\begin{array}{rcl@{\qquad}rcl}
\mathit{config} & ::= & \varepsilon \mid \mathit{class} \mid \mathit{object} \mid \mathit{msg} \mid \mathit{config}\ \mathit{config} & o & ::= & (\mathit{oid}, C\#n) \\
\mathit{class} & ::= & (C\#n, \mathit{vs}, \mathit{impl}, \mathit{inh}, \mathit{ob}, \mathit{fds}, \mathit{mtds}) & \mathit{fds} & ::= & \overline{T\ v\ \mathit{val}} \\
\mathit{object} & ::= & (o, \mathit{pv}, \mathit{processQ}, \mathit{fds}, \mathit{active}) & \mathit{active} & ::= & \mathit{process} \mid \mathsf{idle} \\
\mathit{processQ} & ::= & \varepsilon \mid \mathit{process} \mid \mathit{processQ}\ \mathit{processQ} & \mathit{mc} & ::= & \mathit{oid}.m(\overline{\mathit{val}}) \\
\mathit{msg} & ::= & (\mathit{fid}, \mathit{mc}, \mathit{mode}, \mathit{val}) \mid (\mathit{bind}, \overline{C}, \mathit{fid}, \mathit{mc}) & \mathit{val} & ::= & \mathit{oid} \mid \mathit{fid} \mid \mathsf{null} \mid b \\
 & & \mid (\mathit{bound}, o, \mathit{process}) & \mathit{mtd} & ::= & T\ m(\overline{T\ x})\{\mathit{process}\} \\
\mathit{process} & ::= & (\mathit{fds}, \mathit{sr}) \mid \mathsf{error} & \mathit{mtds} & ::= & \varepsilon \mid \mathit{mtd} \mid \mathit{mtds}\ \mathit{mtds}
\end{array}
$$

Figure 10.3: Syntax for runtime configurations; *oid* and *fid* are object and future identifiers.

The rules (Poll) and (Get) for operations on futures convert types from T to $\mathsf{fut}(T)$. Let $\mathit{interfaces}(\Gamma_C(C))$ denote the interfaces of C as declared in Γ_C. Rule (New) shows the connection between the type of the variable and the interfaces of the class; the typing of a class instance depends on the context. The function $\mathit{curr}(C,\Gamma)$ identifies the current version number of a class C. In (IntCall) the auxiliary predicate *matchint*, given a method name, signature, and class, checks that an internal invocation may be bound in the class mapping. Similarly, in (ExtCall), *matchext* checks that the interface of the callee can bind the external call. Any interface can type null in (Null). For a variable v, let $[\![v]\!] : \mathsf{Dep}$ denote the class in which v is declared and its version number (which is easily retrieved from the typing environment). The effect of the analysis of expressions and guards is a set of dependencies to versions of the current class and its superclasses.

In rule (Method), local declarations extend the typing environment used for statements in the method body. The dependencies from different statements are accumulated in (Composition). The effect of the analysis of a method is the set of all dependencies from the body of the method. In (Class), this is bound to the class name, the context is extended with fields, and each method is typechecked. For each method M in rule (Class), the dependencies accumulated by the analysis of M are stored in the effect Σ^M. After the analysis of a class C, the dependency mapping Γ_d maps the dependencies of the initial version of C to the dependencies accumulated from the type analysis of the class; i.e., $\langle C,0\rangle \mapsto_d \bigcup_{M\in\overline{M}} \Sigma^M$. In (Program), a program is type checked in the context $\Gamma_I + \Gamma_C$. Here, Γ_d^L denotes the dependency mapping derived for class L in the program.

10.2.2 Context-Reduction Semantics

The semantics is given by a small-step reduction relation on *configurations* of objects, classes, and futures (see Fig. 10.3). To accommodate upgrades in Sect. 10.4, stage and version numbers are introduced in the semantics. A *class* has an id (i.e., a name and a *stage number* n: Nat, which changes when the class or one of its superclasses is upgraded), a *version number* vs: Nat (which changes only when the class itself is upgraded), a list of interfaces, a list of superclasses, a set of object ids, a set of fields with default values, and a set of methods. Default values for types are given by a function *default* (e.g., $\mathit{default}(I) = \mathsf{null}$, $\mathit{default}(\mathsf{bool}) = \mathsf{false}$, and $\mathit{default}(!T) = \mathsf{null}$). An *object* has an id *oid*, a class with a stage number $C\#n$, a process version set $pv : \mathsf{Set}[\mathit{fid} \times \mathsf{Nat}]$, a queue pq of suspended processes, fields *fds*, and an active process. In an object o, pv tracks the stage of the class for (pending) method activations on o: these may be either internal calls or incoming requests, and are removed upon process completion. The idle process indicates that no method is active in the object and error that method binding has failed. A *future* $(\mathit{fid}, \mathit{mc}, \mathit{mode}, \mathit{val})$ captures the state of a method call: initially *sleeping*, then *active*, and finally, it becomes *completed* and stores the result from the call. Let $\mathit{mode} \in \{\mathsf{s}, \mathsf{a}, \mathsf{c}\}$ represent these three states. The *initial configuration* of a program $\overline{L}\ \{\overline{T\ x}; sr\}$ has classes and one object $(o, \emptyset, \varepsilon, \varepsilon, (\overline{T\ x\ \mathit{default}(T)}), sr)$.

Reduction takes the form of a relation $\mathit{config} \rightarrow \mathit{config}'$. The main rules are given in Fig. 10.4. The context reduction semantics decomposes a statement into a reduction context and a redex, and reduces the redex [14]. *Reduction contexts* are method bodies M, statements S, expressions E, and guards G with a single hole denoted by $\bullet$:

$$\begin{array}{lcl@{\qquad}lcl}
M & ::= & \bullet \mid S; \mathsf{return}\ e \mid \mathsf{return}\ E & S & ::= & \bullet \mid v := E \mid S; s \mid \mathsf{if}\ G\ \mathsf{then}\ s_1\ \mathsf{fi} \\
E & ::= & \bullet \mid E.\mathsf{get} \mid E!m(\overline{e}) \mid \mathit{oid}!m(\overline{\mathit{val}}, E, \overline{e}) & G & ::= & \bullet \mid E? \mid G \wedge g \mid b \wedge G
\end{array}$$

Redexes reduce in their respective contexts; i.e., body-redexes in M, stat-redexes in S, expr-redexes in E, and guard-redexes in G. Redexes are defined as follows:

$$\begin{array}{lcl}
\mathit{body\text{-}redexes} & ::= & \mathsf{return}\ \mathit{val} \\
\mathit{stat\text{-}redexes} & ::= & x := \mathit{val} \mid f := \mathit{val} \mid \mathsf{await}\ g \mid \mathsf{skip}; s \mid \mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ s\ \mathsf{fi} \mid \mathsf{release} \\
\mathit{expr\text{-}redexes} & ::= & x \mid f \mid \mathit{fid}.\mathsf{get} \mid \mathit{oid}!m(\overline{\mathit{val}}) \mid \mathsf{new}\ C() \\
\mathit{guard\text{-}redexes} & ::= & \mathit{fid}? \mid b \wedge g
\end{array}$$

Filling the hole of a context M with a redex r is denoted $M[r]$. Before evaluating the expression e in the method body $s; \mathsf{return}\ e$, the body will be reduced to $\mathsf{skip}; \mathsf{return}\ e$. For simplicity, we elide the skip and write just $\mathsf{return}\ e$.

The funtion *lookup* returns a process resulting from the matching of a method name m and signature with a set of method definitions.

Definition 2 Let m be a method name, $\overline{T} \rightarrow T$ a signature and *mtds* a set of method definitions. Define $\mathit{lookup}(m(\overline{\mathit{val}}), \overline{T} \rightarrow T, \mathit{fid}, \mathit{mtds})$ recursively as follows:

$$\begin{array}{l}
\mathit{lookup}(m(\overline{\mathit{val}}), \overline{T} \rightarrow T, \mathit{fid}, \varepsilon) = \mathsf{error} \\
\mathit{lookup}(m(\overline{\mathit{val}}), \overline{T} \rightarrow T, \mathit{fid}, (T\ m(\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'}; sr\})\ \mathit{mtds}) = \\
\qquad (\mathsf{fut}(T)\ \mathit{destiny}\ \mathit{fid}; T\ \mathit{return}\ \mathit{default}(T); \overline{T}\ \overline{x}\ \overline{\mathit{val}}; \overline{T'}\ \overline{x'}\ \mathit{default}(\overline{T'}); sr) \\
\mathit{lookup}(m(\overline{\mathit{val}}), \overline{T} \rightarrow T, \mathit{fid}, \mathit{mtd}\ \mathit{mtds}) = \\
\qquad \mathit{lookup}(m(\overline{\mathit{val}}), \overline{T} \rightarrow T, \mathit{fid}, \mathit{mtds})\ \mathit{otherwise}
\end{array}$$

Statements. In (Red-Assign1) and (Red-Assign2), values are assigned to local and program variables. In (Red-Cond1), the boolean condition evaluates to true so the evaluation of s is choosen. Otherwise, evaluate s' (Red-Cond2).

Expressions and guards. In (Red-Call1) and (Red-Call2), external and internal asynchronous calls add a sleeping future to the configuration, returning its id to the caller. Note that an internal call extends the process version set with a pair consisting of the new future and the current stage number. This is because the asynchronous call to an internal method creates an obligation for

(RED-ASSIGN1)

$$\begin{array}{l}(o,pv,pq,fds,(l,M[x := val]))\\ \rightarrow (o,pv,pq,fds,(l+[x \mapsto\ val],M[\mathsf{skip}]))\end{array}$$

(RED-ASSIGN2)

$$\begin{array}{l}(o,pv,pq,fds,(l,M[f := val]))\\ \rightarrow (o,pv,pq,fds+[f \mapsto\ val],(l,M[\mathsf{skip}]))\end{array}$$

(RED-COND1)

$$\begin{array}{l}(o,pv,pq,fds,(l,M[\mathsf{if\ true\ then}\ s\ \mathsf{else}\ s']))\\ \rightarrow (o,pv,pq,fds,(l,M[s]))\end{array}$$

(RED-COND2)

$$\begin{array}{l}(o,pv,pq,fds,(l,M[\mathsf{if\ false\ then}\ s\ \mathsf{else}\ s']))\\ \rightarrow (o,pv,pq,fds,(l,M[s']))\end{array}$$

(RED-SKIP)

$$\begin{array}{l}(o,pv,pq,fds,(l,M[\mathsf{skip};s]))\\ \rightarrow (o,pv,pq,fds,(l,M[s]))\end{array}$$

(GUARD1)

$$\begin{array}{l}(o,pv,pq,fds,(l,M[\mathsf{true}\wedge G]))\\ \rightarrow (o,pv,pq,fds,(l,M[G]))\end{array}$$

(GUARD2)

$$\begin{array}{l}(o,pv,pq,fds,(l,M[\mathsf{false}\wedge G]))\\ \rightarrow (o,pv,pq,fds,(l,M[\mathsf{false}]))\end{array}$$

(RED-CALL1)

$$\frac{oid \neq \mathsf{this} \qquad fid \text{ is fresh}}{\begin{array}{l}(o,pv,pq,fds,(l,M[oid!m(\overline{val})]))\\ \rightarrow (o,pv,pq,fds,(l,M[fid]))\\ (fid,oid.m(\overline{val}),\mathsf{s},\mathsf{null})\end{array}}$$

(RED-CALL2)

$$\frac{fid \text{ is fresh}}{\begin{array}{l}((oid,C\#n),pv,pq,fds,(l,M[\mathsf{this}!m(\overline{val})]))\\ \rightarrow ((oid,C\#n),pv\cup\{(fid,n)\},pq,fds,(l,M[fid]))\\ (fid,oid.m(\overline{val}),\mathsf{s},\mathsf{null})\end{array}}$$

(RED-NEW)

$$\frac{oid \text{ is fresh} \qquad fds'' = attr(C\#n)}{\begin{array}{l}(o,pv,pq,fds,(l,M[\mathsf{new}\ C()]))\ (C\#n,vs,impl,inh,ob,fds',mtds)\\ \rightarrow (o,pv,pq,fds,(l,M[oid]))\ (C\#n,vs,impl,inh,(ob\cup\{oid\}),fds',mtds)\\ ((oid,C\#n),\varepsilon,\varepsilon,fds'',(\varepsilon,\mathsf{skip}))\end{array}}$$

(RED-POLL)

$$\frac{b = (m \equiv \mathsf{c})}{\begin{array}{l}(o,pv,pq,fds,(l,M[fid?]))\ (fid,mc,m,val)\\ \rightarrow (o,pv,pq,fds,(l,M[b]))\ (fid,mc,m,val)\end{array}}$$

(RED-AWAIT)

$$\begin{array}{l}(o,pv,pq,fds,(l,M[\mathsf{await}\ g]))\\ \rightarrow (o,pv,pq,fds,(l,M[\mathsf{if}\ g\ \mathsf{then}\\ \qquad \mathsf{skip\ else\ release;\ await}\ g\ \mathsf{fi}]))\end{array}$$

(RED-BOUND)

$$\begin{array}{l}((oid,C),pv,pq,fds,\mathsf{idle})\\ (bound,oid,process)\\ \rightarrow ((oid,C),pv,pq :: process,fds,\mathsf{idle})\end{array}$$

(RED-RELEASE)

$$\begin{array}{l}(o,pv,pq,fds,(l,M[\mathsf{release}]))\\ \rightarrow (o,pv,pq :: (l,M[\mathsf{skip}]),fds,\mathsf{idle})\end{array}$$

(RED-RETURN)

$$\frac{l(\mathsf{destiny}) = fid \qquad pv' = pv\setminus\{(fid,n)\}}{\begin{array}{l}(o,pv,pq,fds,(l,\mathsf{return}\ val : T))\ (fid,oid.m(\overline{val}),\mathsf{a},\mathsf{null})\\ \rightarrow (o,pv',pq,fds,\mathsf{idle})\ (fid,oid.m(\overline{val}),\mathsf{c},val)\end{array}}$$

(RED-RESCHEDULE)

$$\begin{array}{l}(o,pv,p :: pq,fds,\mathsf{idle})\\ \rightarrow (o,pv,pq,fds,p)\end{array}$$

(RED-BIND1)

$$\begin{array}{l}((oid,C\#n),pv,pq,fds,p)\ (fid,oid.m(\overline{val}),\mathsf{s},\mathsf{null})\\ \rightarrow ((oid,C\#n),pv\cup\{(fid,n)\},pq,fds,p)\\ \quad (fid,oid.m(\overline{val}),\mathsf{a},\mathsf{null})\ (bind,C\#n,fid,oid.m(\overline{val}))\end{array}$$

(RED-GET)

$$\begin{array}{l}(o,pv,pq,fds,(l,M[fid.\mathsf{get}]))\\ (fid,mc,\mathsf{c},val)\\ \rightarrow (o,pv,pq,fds,(l,M[val]))\end{array}$$

(RED-BIND2)

$$\frac{lookup(m(\overline{val}),sig(m(\overline{val}),fid),fid,mtds) = \mathsf{error}}{\begin{array}{l}(bind,(C\#n;\overline{cid}),fid,oid.m(\overline{val}))(C\#n',vs,impl,inh,ob,fds,mtds) \rightarrow\\ (bind,(\overline{inh};\overline{cid}),fid,oid.m(\overline{val}))\ (C\#n',vs,impl,inh,ob,fds,mtds)\end{array}}$$

(RED-CONTEXT)

$$\frac{config \rightarrow config'}{\begin{array}{l}config\ config''\\ \rightarrow config'\ config''\end{array}}$$

(RED-BIND4)

$$\frac{\begin{array}{c}process \neq \mathsf{error} \qquad n \leq n'\\ lookup(m(\overline{val}),sig(m(\overline{val}),fid),fid,mtds) = process\end{array}}{\begin{array}{l}(bind,C\#n;\overline{cid},fid,oid.m(\overline{val}))\ (C\#n',vs,impl,inh,ob,fds,mtds)\\ \rightarrow (bound,oid,process)\ (C\#n',vs,impl,inh,ob,fds,mtds)\end{array}}$$

(RED-BIND3)

$$\begin{array}{l}(bind,\varepsilon,fid,oid.m(\overline{val}))\\ \rightarrow (bound,oid,\mathsf{error})\end{array}$$

Figure 10.4: The context reduction semantics.

the object to keep this method available until the call has been executed. In (RED-GET), a read on a future variable in the active process only reduces if the corresponding future is in completed mode. Otherwise, the process is blocked. In (RED-NEW), a new instance of a class C is introduced into the configuration (with fields collected from C and its superclasses using $attr(C)$). In (RED-POLL), a future variable is polled to see if a call has been executed.

Release and rescheduling. Guards determine whether a process should be released. In (RED-AWAIT), a process at a release point proceeds if its guard is true and releases otherwise. When a process is released, its guard is reused to reschedule the process. When an active process is released in (RED-RELEASE) or terminates, it is replaced by the idle process, which allows a process from the process queue to be scheduled for execution in (RED-RESCHEDULE).

Method invocation, binding, and return. A method call results in an activation on the callee's process queue. As the call is asynchronous, there is a delay between the call and its activation, represented by the sleeping mode of a future. After the call, (RED-BIND1) creates a bind request to the callee's class and the future changes its mode to active, preventing multiple activations. Note that the bind request extends the process version set with a pair consisting of the new future and the current stage number of the object, similar to an invocation for an internal call. The process version set influences the applicability of upgrades, delaying those upgrades that may introduce errors into the nonterminated processes. (RED-BIND2) traverses the implicit inheritance tree until binding fails in (RED-BIND3) or succeeds in (RED-BIND4). Successful binding results in a *bound* message to the callee, which is loaded into the process queue in (RED-BOUND). When the process terminates, the result is stored by (RED-RETURN) in the future identified by the destiny variable. This future changes its mode to completed and the active process becomes idle. When a process terminates, its return value is placed in the associated future and the future id is removed from the process version set pv by (RED-RETURN). Finally, (RED-CONTEXT) reduces subconfigurations.

Adapting the type system to runtime configurations, we let $\Delta \vdash_R config$ ok denote that *config* is well-typed. The initial state of a well-typed program is well-typed, and type soundness can be established for the type system and reduction semantics of this paper (the details of the proof are given in Appendix 10.A):

Theorem 1 *If* $\Delta \vdash_R config$ ok *and* $config \rightarrow config'$*, then there is an extension* Δ' *of* Δ *such that* $\Delta' \vdash_R config'$ ok

10.3 Example of Dynamic Class Extensions

Let an interface Account provide basic banking services; e.g., depositing money and receiving the balance for an account. Class BankAccount implements Account; an internal method increaseBalance is called by method deposit. The comment *V:0* indicates that this is class version 0.

```
class BankAccount implements Account {                                    -- V:0
  Nat bal:=0;
  Bool increaseBalance (Nat sum) { bal := bal + sum; return true }
  Nat balance ( ) { return bal }
  Bool deposit (Nat sum) { return this !increaseBalance(sum) }}
```

By *dynamically extending* the class with new methods transfer and withdraw, money can be transferred to a receiver account or withdrawn. To log transactions, a general method modifyBalance will modify the balance of the account and log the transaction:

```
class BankAccount implements Account {                                    -- V:1
  Nat bal:=0; Log l;
  Nat modifyBalance (Int sum) {Nat w; w := 0; l:= new Log(); bal := bal + sum;
    l.addlog(this, sum); if sum < 0 then w := −sum fi; return w }
  Bool increaseBalance(Nat sum) {bal := bal + sum; return true }
  Nat balance ( ) { return bal }
  Bool deposit (Nat sum) { fut⟨Nat⟩ w; w:=this!modifyBalance(sum); return true }
  Nat withdraw (Nat sum ) { await sum ≤ bal; return this!modifyBalance(−sum) }
  Bool transfer (Nat sum, Account acc) { fut⟨Nat⟩ w; await sum ≤ bal;
    w:=this!modifyBalance(−sum); return acc!deposit(w.get()) }}
```

In this class upgrade, the class is extended with a new field l, new methods modifyBalance, withdraw, and transfer. Furthermore, deposit is redefined to use the internal method modifyBalance. (Remark that allowing asynchronous calls as statements in the language would remove the need for the future w in deposit.) However, the new methods withdraw and transfer are only known internally in the class. To *export* these methods the class is extended with a new interface TransferAcc which provides methods transfer and withdraw with appropriate signatures, after which transfer and withdraw may be invoked on pointers typed by TransferAcc. If we can type check that BankAccount implements TransferAcc, it is type-safe to bind a pointer typed by TransferAcc to an instance of BankAccount and call transfer and withdraw on this object:

```
class BankAccount implements Account, TransferAcc {                       -- V:2
  Nat bal:=0; Log l;
  Nat modifyBalance (Int sum) { Nat w; w := 0; l:= new Log(); bal := bal + sum;
    l.addlog(this, sum); if sum < 0 then w := −sum fi; return w }
  Bool increaseBalance(Nat sum) {bal := bal + sum; return true }
  Nat balance ( ) { return bal }
  Bool deposit (Nat sum) { fut⟨Nat⟩ w; w:=this!modifyBalance(sum); return true }
  Nat withdraw (Nat sum ) { await sum ≤ bal; return this!modifyBalance(−sum) }
  Bool transfer (Nat sum, Account acc) { fut⟨Nat⟩ w; await sum ≤ bal;
    w:=this!modifyBalance(−sum); return acc!deposit(w.get()) }}
```

As method increaseBalance is now redundant, it is now safe to *dynamically simplify* class BankAccount by removing this method. After successful upgrades, the following class replaces the initial runtime class definition:

```
class BankAccount implements Account, TransferAcc {                       -- V:3
  Nat bal:=0; Log l;
  Nat modifyBalance (Int sum) { Nat w; w := 0; l:= new Log(); bal := bal + sum;
    l.addlog(this, sum); if sum < 0 then w := −sum fi; return w }
  Nat balance ( ) { return bal }
  Bool deposit (Nat sum) { fut⟨Nat⟩ w; w:=this!modifyBalance(sum); return true }
  Nat withdraw (Nat sum ) { await sum ≤ bal; return this!modifyBalance(−sum) }
  Bool transfer (Nat sum, Account acc) { fut⟨Nat⟩ w; await sum ≤ bal;
w:=this!modifyBalance(−sum); return acc!deposit(w.get()) }}
```

These dynamic upgrades are here realized by three upgrade messages added to the running system: upgrading BankAccount with the redefinition of deposit and the new methods modifyBalance, withdraw and transfer; exporting new functionality by extending BankAccount with the TransferAcc interface; and removing the redundant method increaseBalance. A type-safe introduction of these upgrades in a distributed system requires a combination of type checking and careful timing at runtime. The addition of the new interface requires the presence of methods withdraw and transfer, which means that the first upgrade of BankAccount must already have occurred. Moreover, the class of l must implement Log. There are similar dependencies for removing method increaseBalance: the redefinition of deposit must occur before the class simplify upgrade, otherwise method binding may fail. Furthermore, we must ensure that the old definition of deposit is not a process in any runtime object for similar reasons.

10.4 Dynamic Classes

Software evolution in a running system may be perceived as a series of operations injected into the system, which modify the classes and the class hierarchy. An upgrade U is any dynamic class operation, as given by the following syntax:

$$\begin{array}{lll} U ::= & \textsf{new-class}\ C\ \textsf{extends}\ \overline{C}\ \textsf{implements}\ \overline{I}\ \{\overline{T\ f};\overline{M}\} & \mid \textsf{new-interface}\ I\ \textsf{extends}\ \overline{I}\ \{\overline{M_s}\} \\ \mid & \textsf{update}\ C\ \textsf{extends}\ \overline{C}\ \textsf{implements}\ \overline{I}\ \{\overline{T\ f};\overline{M}\} & \mid \textsf{simplify}\ C\ \textsf{retract}\ \overline{C}\ \{\overline{T\ f};\overline{M}\} \end{array}$$

A *class addition* adds the representation of the new class to the system, an *interface addition* extends the type system, a *class update* extends an existing class with new fields and methods and redefines existing methods in the class, and a *class simplification* removes redundant superclasses, fields, and methods from an existing class. Upgrades propagate asynchronously at runtime. They first change classes, then subclasses, and eventually the objects of those classes.

10.4.1 Typing of Dynamic Classes

Dynamic class operations are type checked in a sequence of typing environments $\Gamma^0, \Gamma^1, \ldots$, which extend each other; Γ^0 is the typing environment for the original program and Γ^i the current static view of the system. We describe the construction of Γ^{i+1} for the next well-typed upgrade U. The type system for judgments $\Gamma^{i+1} \vdash U$ is shown in Fig. 10.5, extending the system in Fig. 10.2. For simplicity, we omit the analysis of new-interface and focus on class updates. We assume that new interfaces are well-typed and that Γ^i_I are correctly extended for each update. (As before, we omit the straightforward analysis of superinterfaces and method signatures.) Rule (New-Class) for class additions requires a fresh name, type checks like a class in the original program, and extends Γ^i_C. Remark that the version number of the new class is different from that of the program's original classes. This reflects the fact that the new class may depend on other dynamic changes to the system. For a new class, we remove the dependency to the class itself; i.e., $(C,0)$ is removed from the dependency mapping.

$$\textsc{(New-Class)}\quad \frac{C \notin dom(\Gamma^i_C) \quad \Gamma' = [C \mapsto_C (\overline{I}, \overline{C}, \overline{T\ f}, \overline{M})] \quad \forall I \in \overline{I} \cdot implements(C, I, \Gamma + \Gamma') \\ \forall M \in \overline{M}\ \cdot\ \Gamma^i + \Gamma' + [\mathsf{this} \mapsto_v C] + [attr(C, \Gamma^i + \Gamma')] \vdash M \langle \Sigma^M \rangle}{\Gamma^i + \Gamma' + [\langle C, 1\rangle \mapsto_d \bigcup_{M \in \overline{M}} \Sigma^M \setminus \{(C,0)\}] \vdash \mathsf{new\text{-}class}\ C\ \mathsf{extends}\ \overline{C}\ \mathsf{implements}\ \overline{I}\ \{\overline{T\ f}; \overline{M}\}}$$

$$\textsc{(Class-Extend)}\quad \frac{\Gamma^i_C(C) = (\overline{I}_1, \overline{C}_1, \overline{T_1\ f_1}, \overline{M}_1) \quad \Gamma' = [C \mapsto_C (\overline{I}_1; \overline{I}, \overline{C}_1; \overline{C}, \overline{T_1\ f_1}; \overline{T\ f}, (\overline{M}_1 \oplus \overline{M}))] \\ vs = curr(C, \Gamma^i_d) \quad refines(\overline{M}, \overline{M}_1) \quad \forall I \in \overline{I} \cdot implements(C, I, \Gamma^i + \Gamma') \\ \forall M \in \overline{M}\ \cdot\ \Gamma^i + \Gamma' + [\mathsf{this} \mapsto_v C] + [attr(C, \Gamma^i + \Gamma')] \vdash M \langle \Sigma^M \rangle}{\Gamma^i + \Gamma' + [\langle C, vs+1\rangle \mapsto_d \bigcup_{M \in \overline{M}} \Sigma^M \cup \{(C, vs)\}] \vdash \mathsf{update}\ C\ \mathsf{extends}\ \overline{C}\ \mathsf{implements}\ \overline{I}\ \{\overline{T\ f}; \overline{M}\}}$$

$$\textsc{(Class-Simplify)}\quad \frac{\Gamma^i_C(C) = (\overline{I}_1, \overline{C}_1, \overline{T_1\ f_1}, \overline{M}_1) \quad \Gamma' = [C \mapsto_C (\overline{I}_1, (\overline{C}_1 \setminus \overline{C}), (\overline{T_1\ f_1} \setminus \overline{T\ f}), (\overline{M}_1 \setminus \overline{M}))] \\ \overline{D} = \{C\} \cup below(C, \Gamma^i_C)\} \quad dep = \bigcup_{D \in \overline{D}} \{(D, curr(D, \Gamma^i_d))\} \quad vs = curr(C, \Gamma^i_d) \\ \forall D \in \overline{D} \cdot \Gamma^i + \Gamma' + [\mathsf{this} \mapsto_v D] + [attr(D, \Gamma^i + \Gamma')] \vdash (\Gamma^i_C + \Gamma')(D).mtds \\ \forall D \in \overline{D} \wedge\ \forall I \in \Gamma^i_C(D).impl \cdot implements(D, I, \Gamma^i + \Gamma')}{\Gamma^i + \Gamma' + [\langle C, vs+1\rangle \mapsto_d dep \cup \{(C, vs)\}] \vdash \mathsf{simplify}\ C\ \mathsf{retract}\ \overline{C}\ \{\overline{T\ f}; \overline{M}\}}$$

Figure 10.5: The type system for dynamic class extensions. Judgments have the form $\Gamma^{i+1} \vdash U$, where Γ^i is the current typing environment before the operation U.

Rule (Class-Extend) for the extension of a class C obeys a substitutability discipline captured by the predicate $refines(\overline{M}, \overline{M}_1)$; if $M \in \overline{M}$ redefines $M_1 \in \overline{M}_1$, the signature of M must be a subtype of the signature of M_1. The extended class replaces the definition of C in Γ^i_C by the binding Γ'. When retrieving the old version of the class from Γ^i_C, we represent the class compactly as a tuple and we denote by $\overline{M}_1 \oplus \overline{M}$ the union operation which retains methods in $\overline{M}$ in case of name conflicts. The function $curr(C, \Gamma^i_d)$ identifies the current version number of a class C by inspecting the dependency mapping. The new features of the class extension are type checked in a similar way as rule (Class) and the resulting dependencies, accumulated by the type analysis of methods, are bound to the new version $curr(C, \Gamma^i_d) + 1$ of the class in Γ^{i+1}_d. Moreover, in order to ensure that multiple upgrades to the same class occur in a correct order, the current version of the class is also included in this mapping.

In rule (Class-Simplify), which removes features from a class C, the simplification is restricted to superclasses, fields, and methods which are not statically needed in Γ^i. To verify this requirement, it is necessary to type check the new version of the class as well as its subclasses, identified by the function $below(C, \Gamma^i_C)$, in the updated typing environment Γ^{i+1}. The dependencies of the simplification operation are the current versions of the subclasses and of the class itself. As effects are not needed for the construction of the dependency mapping (the simplification only affects subclasses), for brevity, we elide the effects of methods and denote by $\Gamma(C).mtds$ and $\Gamma(C).impl$ the methods and interfaces of a class C and type check each single method and interface similar to rule (Class).

In the asynchronous setting of distributed concurrent objects, upgrades may be delayed and even bypass each other. Hence, the system reflected by the current typing environment Γ^i may differ considerably from the running system. To ensure that the execution is type safe, we

(DEP)
$$\frac{hd \in \{new, ext\} \qquad vs \geq n}{\begin{array}{l}(hd, C, impl, inh, fds, mtds, ((C', n) \cup dep))\\ \quad (C'\#n', vs, impl', inh', ob, fds', mtds')\\ \to (hd, C, impl, inh, fds, mtds, dep)\\ \quad (C'\#n', vs, impl', inh', ob, fds', mtds')\end{array}}$$

(NEW-CLASS)
$$\begin{array}{l}(new, C, impl, inh, fds, mtds, \emptyset)\\ \to (C\#1, 1, impl, inh, \varepsilon, fds, mtds)\end{array}$$

(DEP-SIMPLIFY)
$$\frac{\begin{array}{c}dep = (C', n) \cup dep' \qquad vs \geq n\\ odep' = odep \cup \{(o, C'\#n') | o \in ob\}\end{array}}{\begin{array}{c}(simp, C, inh, fds, mtds, dep, odep)\\ (C'\#n', vs, impl, inh', ob, fds', mtds')\\ \to (simp, C, inh, fds, mtds, dep', odep'\})\\ (C'\#n', vs, impl, inh', ob, fds', mtds')\end{array}}$$

(DEP-OBJECT)
$$\frac{\begin{array}{c}\mathsf{oldest}(pv) \geq n'\\ odep = \{(oid, C'\#n')\} \cup odep'\end{array}}{\begin{array}{c}(simp, C, inh, fds, mtds, dep, odep)\\ ((oid, C'\#n), pv, pq, fds, active)\\ \to (simp, C, inh, fds, mtds, dep, odep')\\ ((oid, C'\#n), pv, pq, fds, active)\end{array}}$$

(EXTEND-CLASS)
$$\begin{array}{l}(ext, C, impl, inh, fds, mtds, \emptyset)\\ (C\#n, vs, impl', inh', ob, fds', mtds')\\ \to (C\#(n+1), vs+1, impl'; impl,\\ inh'; inh, ob, fds'; fds, mtds' \oplus mtds)\end{array}$$

(SIMPLIFY-CLASS)
$$\begin{array}{l}(simp, C, inh, fds, mtds, \emptyset, \emptyset)\\ (C\#n, vs, impl, inh', ob, fds', mtds')\\ \to (C\#(n+1), vs+1, impl, inh' \setminus inh,\\ \quad ob, fds' \setminus fds, mtds' \setminus mtds)\end{array}$$

(CLASS-INH)
$$\frac{n'' > n'}{\begin{array}{c}(C'\#n'', vs', impl', inh', ob', fds', mtds')\\ (C\#n, vs, impl, (\overline{cid}; C'\#n'; \overline{cid}'), ob, fds, mtds)\\ \rightharpoonup (C'\#n'', vs', impl', inh', ob', fds', mtds')\\ (C\#n+1, vs, impl, (\overline{cid}; C'\#n''; \overline{cid}'), ob, fds, mtds)\end{array}}$$

(RED-CONTEXT2)
$$\frac{config \rightharpoonup config'}{\begin{array}{c}config\ config''\\ \rightharpoonup config'\ config''\end{array}}$$

(OBJ-STATE)
$$\frac{n' > n \quad fds' = \mathsf{transf}(fds, attr(C))}{\begin{array}{c}((oid, C\#n), pv, pq, fds, \mathsf{idle})\\ (C\#n', vs, impl, inh, ob, fds, mtds)\\ \rightharpoonup ((oid, C\#n'), pv, pq, fds', \mathsf{idle})\\ (C\#n', vs, impl, inh, ob, fds, mtds)\end{array}}$$

(UPGRADE)
$$config \xrightarrow{upg} config\ upg$$

(RED)
$$\frac{\begin{array}{c}config_1 \rightharpoonup ! config'_1\\ config'_1 \to config_2\end{array}}{config_1 \longrightarrow_{up} config_2}$$

Figure 10.6: The context reduction semantics for class upgrades.

exploit the dependency mapping of Γ^i to impose constraints on the applicability of the i'th upgrade at runtime. The constraints ensure that if one upgrade depends on another, they will be applied in the correct order, otherwise, they may be applied in any order, or in parallel.

10.4.2 Semantics for Dynamic Classes

We extend the runtime syntax of Figure 10.3 with upgrade messages as follows:

$$\begin{array}{rcl@{\qquad}rcl} \mathit{upg} & ::= & (\mathit{new},C,\mathit{impl},\mathit{inh},\mathit{fds},\mathit{mtds},\mathit{dep})\,|\,(\mathit{ext},C,\mathit{impl},\mathit{inh},\mathit{fds},\mathit{mtds},\mathit{dep}) & \mathit{dep} & ::= & \overline{(C,n)} \\ & | & (\mathit{simp},C,\mathit{inh},\mathit{fds},\mathit{mtds},\mathit{dep},\mathit{odep})\,|\,\ldots & \mathit{odep} & ::= & \overline{(o,C\#n)} \end{array}$$

An upgrade message for a new class or class extension has a class name C, a list *impl* of interfaces, a list *inh* of superclasses, a list *fds* of new fields, a set *mtds* of new (or redefined) methods, and a set *dep* of constraints to classes in the runtime system. For the simplification of classes, *inh*, *fds* and *mtds* are the superclasses, fields and methods to be removed, respectively. For simplification, applicability does not only depend on the class constraints but also propagates to the state of runtime objects, as there may exist processes in or communication between objects in the runtime environment that uses fields or methods to be removed. Thus, in addition to class constraints, a class simplification message includes a set *odep* of constraints on objects, which is initially empty but gradually extended during the verification of class constraints. If a message injected into the runtime configuration is well-typed in an environment Γ^i, then *dep* is $\Gamma^i_d(\langle C, \mathit{curr}(C,\Gamma^i_d)\rangle)$. Thus, the static dependencies of the current upgrade are introduced into the runtime configuration.

The semantics for dynamic class operations extends the reduction system of Fig. 10.4 with the rules given in Fig. 10.6. A reduction step in the extended system takes the form $\mathit{config}_1 \longrightarrow_{up} \mathit{config}_2$ in (Red), where $\mathit{config}_1 \rightharpoonup ! \mathit{config}'_1$ reduces config_1 to *normal form* by the relation $\rightharpoonup$, which consists of the two rules (Class-Inh) and (Obj-State), before the relation $\rightarrow$ applies. The $\rightharpoonup$ relation is similar to equational reduction in Maude [9] and abstracts from locking disciplines that would otherwise be needed, as explained below.

Dynamic class operations are initiated by injecting a message *upg* into the configuration by (Upgrade). For the *extension* of a class C, this message is $(\mathit{ext},C,\mathit{impl},\mathit{inh},\mathit{fds},\mathit{mtds},\mathit{dep})$ which cannot be applied unless the constraints in *dep* are satisfied, checked by (Dep). Thus, the upgrade is delayed at runtime until other upgrades have been applied. When the constraints are satisfied, the superclasses, fields, and methods of the runtime class definition are extended and the stage and version numbers increased in (Extend-Class). (For the operator $\oplus$, see Sect. 10.4.1.) Similarly, (New-Class) creates a new runtime class when the constraints are satisfied. For the *simplification* of a class C, the message is $(\mathit{simp},C,\mathit{inh},\mathit{fds},\mathit{mtds},\mathit{dep},\emptyset)$. When verifying class constraints in (Dep-Simplify), the set *ob* of instances of a class is used for stage constraints in *odep*. These are checked in (Dep-Object). To guarantee that an object is of stage n, we must ensure that all processes stemming from older versions of the class have completed and that there are no pending calls from such processes to local methods (which could be scheduled for removal). Rule (Dep-Object) compares stage constraints to the *oldest* stage number in the object's process version set *pv*. When no unsatisfied dependencies remain, the simplification can be applied in (Simplify-Class).

Updating the object state. When an object's class or superclass has been upgraded, the object's state must be updated *before* new code is allowed to execute. New instances of a class automatically get the new fields, but the upgrade of existing instances must be closely controlled; errors may occur if new or redefined methods, which rely on fields that are not yet available in the object, were executed. With recursive or nonterminating methods objects cannot generally be expected to reach a state without pending processes, even if the activation of new method calls were postponed as suggested in [2]. Consequently, it is too restrictive to wait for the completion of all processes before applying an upgrade. However, objects may reach *quiescent* states when the processor has been released and before any pending process has been activated. Quiescent states are those in which the active process is idle. Any object which does not deadlock will eventually reach a quiescent state. In our language, nonterminating activity is defined by recursion, which ensures at least one quiescent state in each cycle. Consequently, the object state will be upgraded in quiescent states.

Class upgrades propagate to objects in two steps. When a class C is upgraded in (EXTEND-CLASS) or (SIMPLIFY-CLASS), both its stage and version numbers increase. In order to notify objects of this change, the stage change propagates in rule (CLASS-INH) to the subclasses of C, and the subclasses recursively increment their stage numbers. Since this notification is given priority, the object gets an upgrade the next time it interacts with a class in rule (RED-BIND4). Before the new process is activated, the active process must become idle, in which case (OBJ-STATE) applies. The *transf* function returns the new state, retaining the values of old fields.

The reduction $\longrightarrow_{up}$ in (RED) reduces a subconfiguration by $\rightharpoonup$ to its normal form before a $\rightarrow$ rewrite, simulating locking the object. Thus, the use of the $\rightharpoonup$ relation abstracts from two locking disciplines; one to deny access to classes for *bind* messages while (CLASS-INH) is applicable and the other to delay processing *bind* messages from a callee by an upgraded class until the callee's state has been updated. A class may be upgraded several times before the object reaches a quiescent state, so the object may miss some upgrades. However a single state update suffices to ensure that the object, once upgraded, is a complete instance of the present version of its class. Extending Theorem 1, type soundness holds for the dynamic class system (the details of the proof are given in Appendix 10.A):

Theorem 2 (Subject reduction) *Let P be a well-typed program with initial configuration init and let $U_1,\ldots,U_n$ be a series of well-typed dynamic class operations with runtime representation upg_i for U_i. If init $\longrightarrow_{up}$config and upg_{i+1} is injected in the runtime configuration after upg_i for all $i < n$, then there is a typing context Δ such that $\Delta \vdash_R$ config* ok.

Proof (sketch). The proof is by induction over the number of reduction steps and then by cases. We show that injecting upg_i maintains the well-typedness of the configuration. Furthermore, we show that the dependencies provided by the static analysis enforce an ordering of upgrades such that new definitions give well-typed configurations. Especially, existing processes as well as new processes and fields in runtime objects are well-typed after the possible reductions. ∎

10.5 Related Work

For many modern distributed applications, system availability during reconfiguration is crucial. Among dynamic or online upgrade solutions, version control systems aim at modular evolution; some keep multiple co-existing versions of a class or schema [6,5,4,13,15,17], others apply a global update or "hot-swapping" [22,20,2,7]. The approaches differ for active behavior, which may be disallowed [22,20,15,7], delayed [2], or supported [24,17]. Hjálmtýsson and Gray [17] propose proxy classes and reference indirection for C++, with multiple versions of each class.

Old instances are not upgraded, so their activity is not interrupted. Existing approaches for Java, using proxies [22] or modifying the Java virtual machine [20], use global upgrade and do not apply to active objects.

Automatic upgrades by lazy global update has been proposed for distributed objects [2] and persistent object stores [7], in which instances of upgraded classes are upgraded, but inheritance and (nonterminating) active code are not addressed, limiting the effect and modularity of the class upgrade. Remark that the use of recursion instead of loops in our approach guarantees that all non-blocked processes will eventually reach a quiescent state. In [7] the ordering of upgrades is serialized and in [20] invalid upgrades raise exceptions.

It is interesting to apply formal techniques to systems for software evolution. Formalizations of runtime upgrade mechanisms are less studied, but type soundness results exist for imperative [24], functional [5], and object-oriented [12, 6] languages. These approaches consider the upgrade of single type declarations, procedures, objects, or components in the sequential setting. For example, Fickle's type system guarantees type soundness when an object's class changes by an explicit program instruction [12].

In a recent upgrade system for (sequential) C [24], type-safe updates of type declarations and procedures may occur at annotated points identified by static analysis. However, the approach is synchronous as upgrades which cannot be applied immediately will fail. Closer to our work, UpgradeJ [6] uses an incremental type system in which class versions are only typechecked once and the class table is updated to reflect class changes. Our type system is incremental in this sense for new classes and class extensions, but class simplification requires rechecking the subclasses of the modified class. In contrast to our work, UpgradeJ is synchronous and uses explicit upgrade statements in programs. Upgrades only affect the class hierarchy and not running objects. In particular, multiple versions of a class will coexist and the programmer must explicitly refer to the different class versions in the code, for example using instanceOf to identify the class version of an object at runtime. Compared to previous work by the authors [27], dynamic classes allow much more flexible runtime upgrades, including simplification operations which necessitate additional runtime overhead, yet the type system has been simplified. However, we do not support the removal of interfaces from classes. This would increase the runtime overhead significantly, as all objects with fields typed by that particular interface would need to be inspected.

10.6 Conclusion

This paper presents a kernel language for distributed concurrent objects in which programs may evolve at runtime by means of dynamic class operations. The granularity of this mechanism for reprogramming fits well with object orientation and it is modular as a single upgrade may affect a large number of running objects. The mechanism does not impose any particular requirements on the developers of initial applications and provides a fairly flexible mechanism for program evolution. It supports not only the addition of new interfaces and classes to running programs, but also the extension, redefinition, and simplification of existing classes. The dynamic class operations proposed in this paper integrate naturally with the concurrency model adapted in the Creol language and a prototype implementation has been integrated with Creol's execution platform.

The paper presents a type and effect system for dynamic class operations. A characteristic feature of our approach is that the type analysis identifies dependencies between different upgrades which are exploited at runtime to impose constraints on the runtime applicability of a particular upgrade in the asynchronous distributed setting, and suffice to guarantee type soundness. We currently investigate the application of other formal methods to dynamic class operations. In particular, lazy behavioral subtyping [11] seems applicable in order to extend the scope of object-oriented program logics to runtime program evolution.

References

[1] *Proceedings of the General Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*. USENIX, 2003.

[2] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In D. Thomas, editor, *Proc. 20th European Conference on Object-Oriented Programming (ECOOP'06)*, volume 4067 of *Lecture Notes in Computer Science*, pages 452–476. Springer-Verlag, 2006.

[3] T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.

[4] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[5] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating. In *Proc. 2nd Intl. Workshop on Unanticipated Software Evolution (USE)*, April 2003.

[6] G. Bierman, M. Parkinson, and J. Noble. UpgradeJ: Incremental typechecking for class upgrades. In *Proc. 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *Lecture Notes in Computer Science*, pages 235–259. Springer-Verlag, 2008.

[7] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In R. Crocker and G. L. S. Jr., editors, *Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, pages 403–417. ACM Press, 2003.

[8] D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer-Verlag, 2005.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.

[10] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, Mar. 2007.

[11] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. In J. Cuellar and T. Maibaum, editors, *Proc. 15th International Symposium on Formal Methods (FM'08)*, volume 5014 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, May 2008.

[12] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: Fickle$_{\mathrm{II}}$. *ACM Transactions on Programming Languages and Systems*, 24(2):153–191, 2002.

[13] D. Duggan. Type-Based hot swapping of running modules. In C. Norris and J. J. B. Fenwick, editors, *Proc. 6th Intl. Conf. on Functional Programming (ICFP'01)*, volume 36, 10 of *ACM SIGPLAN notices*, pages 62–73, New York, Sept. 3–5 2001. ACM Press.

[14] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[15] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, 1996.

[16] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.

[17] G. Hjálmtýsson and R. S. Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proc. USENIX Tech. Conf. (USENIX '98)*, May 1998.

[18] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[19] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[20] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In E. Bertino, editor, *Proc. 14th European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, pages 337–361. Springer-Verlag, June 2000.

[21] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364:338–356, 2006.

[22] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. In *Proc. International Conference on Software Maintenance (ICSM'02)*, pages 649–658. IEEE Computer Society Press, Oct. 2002.

[23] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. D. Silva, G. R. Ganger, O. Krieger, M. Stumm, M. A. Auslander, M. Ostrowski, B. S. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *USENIX Annual Technical Conference, General Track* [1], pages 141–154.

[24] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and predictable dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 29(4):22, 2007.

[25] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[26] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. Object oriented programming, systems, languages, and applications (OOPSLA'05)*, pages 439–453, New York, NY, USA, 2005. ACM Press.

[27] I. C. Yu, E. B. Johnsen, and O. Owe. Type-safe runtime class upgrades in Creol. In R. Gorrieri and H. Wehrheim, editors, *Proc. 8th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 of *Lecture Notes in Computer Science*, pages 202–217. Springer-Verlag, June 2006.

10.A Typing Rules for Runtime Configurations

We now adapt the typing rules to the syntax of runtime configurations (given in Fig. 10.3). Since that classes, objects, and futures have unique names in the operational semantics, we can without loss of generality assume that all program variables have distinct names at runtime. We call a sequence of reductions of an initial state according to the rules of the operational semantics a *run*. We now show that a run from a well-typed initial configuration will maintain well-typed configurations.

Auxiliary functions. The following definitions define the auxiliary functions for runtime type checking in terms of the current runtime configuration:

In order to successfully bind method calls, the types of the formal parameters of the declared method in a class (when the call is local) or in an interface (when the call is external) must correspond to the types of the actual parameters of the call. This is verified by a predicate *match*.

Definition 3 Let m be a method name, $\overline{T}$ and T be types, $\overline{C}$ a list of classes, $\overline{I}$ a list of interfaces and Δ the current runtime configuration. Define $\mathit{match}(C, m, \overline{T} \to T, \Delta)$ and $\mathit{match}(I, m, \overline{T} \to T, \Delta)$ by

$$\begin{aligned}
&\mathit{match}(\varepsilon, m, \overline{T} \to T, \Delta) = \mathit{false}\\
&\mathit{match}(C;\overline{C}, m, \overline{T} \to T, \Delta) = \Delta(C) = (C\#n, vs, impl, inh, ob, fds, (T\ m(\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'}; sr\})\ mtds)\\
&\mathit{match}(C;\overline{C}, m, \overline{T} \to T, \Delta) = \text{let } (C\#n, vs, impl, inh, ob, fds, mtds) = \Delta(C) \text{ in}\\
&\qquad\qquad \mathit{match}(inh;\overline{C}, m, \overline{T} \to T, \Delta)\ \mathit{otherwise}
\end{aligned}$$

$$\begin{aligned}
&\mathit{match}(\varepsilon, m, \overline{T} \to T, \Delta) = \mathit{false}\\
&\mathit{match}(I;\overline{I}, m, \overline{T} \to T, \Delta) = \Delta(I) = (I, inh, (T\ m(\overline{T\ x}))\ mtds_s)\\
&\mathit{match}(I;\overline{I}, m, \overline{T} \to T, \Delta) = \text{let } (I, inh, mtds_s) = \Delta(I) \text{ in}\\
&\qquad\qquad \mathit{match}(inh;\overline{I}, m, \overline{T} \to T, \Delta)\ \mathit{otherwise}
\end{aligned}$$

We define the functions for collecting fields defined in a class C and in the superclasses of C, for finding the oldest process from a process version set pv and for matching signatures for methods declared in an interface to those in a class implementing the interface. These functions are defined by *attr*, *oldest* and *implements*, respectively.

Definition 4 Let $C;\overline{C}$ be a list of class names, I an interface name, $(fid, n) \cup pv$ a process version set and Δ, the runtime configuration. Define *attr*, *oldest* and *implements* as follows:

$$\begin{aligned}
&\mathit{attr}(C;\overline{C},\Delta) = \mathsf{let}\ (C\#n, vs, impl, inh, ob, fds, mtds) = \Delta(C)\\
&\qquad\qquad \mathsf{in}\ \mathit{mapping}(\mathit{attr}(inh;\overline{C},\Delta), fds)
\end{aligned}$$

$$\begin{aligned}
&\mathit{oldest}((\mathit{fid},n)) = n\\
&\mathit{oldest}((\mathit{fid},n)\cup(\mathit{fid}',n')\cup pv) = \mathit{oldest}((\mathit{fid},n)\cup pv)\ \mathsf{if}\ n \le n'\\
&\mathit{oldest}((\mathit{fid},n)\cup(\mathit{fid}',n')\cup pv) = \mathit{oldest}((\mathit{fid}',n')\cup pv)\ \mathit{otherwise}
\end{aligned}$$

$$\begin{aligned}
&\mathit{implements}(C,I,\Delta) = \mathsf{let}\ (I, inh, (T\ m(\overline{T\ x}))\ mtds_s) = \Delta(I)\ \mathsf{in}\\
&\qquad\qquad \mathit{matching}(C,(I,inh,(T\ m(\overline{T\ x}))\ mtds_s),\Delta)
\end{aligned}$$

$$\begin{aligned}
&\mathit{mapping}(T\ v\ val, \overline{fds}) = [v \mapsto\ T] + \mathit{mapping}(\overline{fds})\\
&\mathit{matching}(C,(I,inh,(T\ m(\overline{T\ x}))\ mtds_s),\Delta) = \mathit{match}(C,m,\overline{T}\to T,\Delta)\wedge\\
&\qquad\qquad \mathit{matching}(C,(I,inh,mtds_s))
\end{aligned}$$

For simplicity, we extend the subtype relation on interfaces such that a class of name C is a subtype of an interface I, written $C \preceq I$, if $\mathit{implements}(C,I,\Delta)$.

Typing rules for runtime configurations. Let $\Delta \vdash_R \mathit{config}\ \mathsf{ok}$ express that the configuration *config* is well-typed in the typing context Δ. The typing rules for runtime configurations are given in Figs. 10.7 and 10.8. Each class of the program is given a runtime representation with version and stage number 0. The initial method body of the program is represented by an object with one active process. We can prove that this object is well-typed (Lemma 3) and show that the well-typedness of runtime configuration is preserved by reductions (Theorem 4).

Lemma 3 *The initial state of any well-typed program is well-typed. Let Δ be the typing context including the interface and class mappings from the static typing environment. If $\Gamma \vdash \overline{D}\,\overline{L}\ \{\overline{T\ x}; s\}$ then $\Delta \vdash_R \overline{L}\ (o, \emptyset, \emptyset, \emptyset, (\overline{T\ x\ default(T)}; s; \mathsf{return}\ true))\ \mathsf{ok}$.*

Proof. We need to show that the representations of classes are well-typed and that

$$\vdash_R (\overline{T\ x\ default(T)}; s; \mathsf{return}\ true))\ \mathsf{ok}.$$

For a class C defined by $(C\#0, 0, impl, inh, ob, fds, mtds)$, we know from static typing that $\forall I \in impl \cdot \mathit{implements}(C,I,\Gamma)$ and $\forall mtd \in mtds\ \cdot\ \Gamma + [\mathsf{this}\mapsto_v C] + [\mathit{attr}(C,\Gamma)] \vdash mtd$. It follows that $\forall mtd \in mtds\ \cdot\ \Delta + [\mathsf{this}\mapsto_v C] + [\mathit{attr}(C,\Delta)] \vdash_R mtd$ and consequently that $\Delta \vdash_R (C\#0, 0, impl, inh, ob, fds, mtds)$.

For the initial process $(\overline{T\ x\ default(T)}; s; \mathsf{return}\ true)$, it is obvious that $\Delta \vdash_R \overline{T\ x\ default(T)}\ \mathsf{ok}$ and since $\Gamma + [\overline{x} \to \overline{T}] \vdash s$ and $\Gamma + [\overline{x} \to \overline{T}] \vdash \mathsf{return}\ true$, it follows that $\Delta + [\overline{x} \to \overline{T}] \vdash_R s$ and $\Delta + [\overline{x} \to \overline{T}] \vdash_R \mathsf{return}\ true$. Thus $\vdash_R (\overline{T\ x\ default(T)}; s; \mathsf{return}\ true))\ \mathsf{ok}$. ■

Theorem 4 (Subject Reduction) *If $\Delta \vdash_R \mathit{config}\ \mathsf{ok}$ and $\mathit{config} \to \mathit{config}'$, then there is an extension Δ' of Δ such that $\Delta' \vdash_R \mathit{config}'\ \mathsf{ok}$*

Proof. The proof is by induction over the reduction rules.

$$\textsc{(State1)}\ \frac{\Delta(v) = T \quad \Delta \vdash_R val : T}{\Delta \vdash_R T\ v\ val\ \mathsf{ok}} \qquad \textsc{(Var)}\ \frac{\Delta(v) \preceq T}{\Delta \vdash_R v : T} \qquad \textsc{(Const)}\ \frac{\Delta(val) \preceq T}{\Delta \vdash_R val : T}$$

$$\textsc{(Configurations)}\ \frac{\Delta \vdash_R config\ \mathsf{ok} \quad \Delta \vdash_R config'\ \mathsf{ok}}{\Delta \vdash_R config\ config'\ \mathsf{ok}}$$

$$\textsc{(Process)}\ \frac{\begin{array}{c}\Delta' = \Delta + [\overline{x} \rightarrow \overline{T}] \quad \Delta(return) = T \\ \Delta' \vdash_R \overline{T\ x\ val}\ \mathsf{ok} \quad \Delta' \vdash_R s\ \mathsf{ok} \\ \Delta'(destiny) = \mathsf{fut}(T) \quad \Delta' \vdash_R e : T\end{array}}{\Delta \vdash_R (\overline{T\ x\ val}, s; \mathsf{return}\ e)\ \mathsf{ok}}$$

$$\textsc{(Process-Queue)}\ \frac{\Delta \vdash_R processQ\ \mathsf{ok} \quad \Delta \vdash_R processQ'\ \mathsf{ok}}{\Delta \vdash_R processQ\ processQ'\ \mathsf{ok}} \qquad \textsc{(Empty)}\ \frac{}{\Delta \vdash_R \varepsilon\ \mathsf{ok}}$$

$$\textsc{(Object)}\ \frac{\begin{array}{c}\Delta_{old}(o) = oldest(pv) \quad \Delta \vdash_R fds\ \mathsf{ok} \quad \Delta \vdash_R pv\ \mathsf{ok} \\ \Delta \vdash_R processQ\ \mathsf{ok} \quad \Delta \vdash_R active\ \mathsf{ok}\end{array}}{\Delta \vdash_R (o, pv, processQ, fds, active)\ \mathsf{ok}}$$

$$\textsc{(Bind)}\ \frac{\begin{array}{c}\Delta(fid) = \mathsf{fut}(T) \quad \Delta \vdash_R \overline{val} : \overline{T} \\ match(\overline{C}, m, \overline{T} \rightarrow T, \Delta)\end{array}}{\Delta \vdash_R (bind, \overline{C}, fid, oid.m(\overline{val}))\ \mathsf{ok}}$$

$$\textsc{(Method)}\ \frac{\Delta + [return \rightarrow T] + [\overline{x \rightarrow T}] \vdash_R process\ \mathsf{ok}}{\Delta \vdash_R T\ m(\overline{T\ x})\{process\}\ \mathsf{ok}} \qquad \textsc{(Bound)}\ \frac{\Delta + [attr(\Delta(oid), \Delta)] \vdash_R process\ \mathsf{ok}}{\Delta \vdash_R (bound, oid, process)\ \mathsf{ok}}$$

$$\textsc{(Constraints1)}\ \frac{\Delta(fid) = \mathsf{fut}(T) \quad \Delta(n) = Nat}{\Delta \vdash_R (fid, n)\ \mathsf{ok}} \qquad \textsc{(Constraints2)}\ \frac{\Delta \vdash_R pv1\ \mathsf{ok} \quad \Delta \vdash_R pv2\ \mathsf{ok}}{\Delta \vdash_R pv1 \cup pv2\ \mathsf{ok}}$$

$$\textsc{(Methods)}\ \frac{\Delta \vdash_R mtds\ \mathsf{ok} \quad \Delta \vdash_R mtds'\ \mathsf{ok}}{\Delta \vdash_R mtds\ mtds'\ \mathsf{ok}} \qquad \textsc{(State2)}\ \frac{\Delta \vdash_R fds\ \mathsf{ok} \quad \Delta \vdash_R fds'\ \mathsf{ok}}{\Delta \vdash_R fds\ fds'\ \mathsf{ok}}$$

Figure 10.7: The typing rules for runtime configurations.

Guards. We have a well-typed guard *fid*? and we apply rule (RED-POLL). There must be a future with id *fid* which is either completed ($m \equiv \mathsf{c}$), in which case *fid*? reduces to *true*, or not, in which case *fid*? reduces to *false*. Since $\vdash_R true : \mathsf{bool}$ and $\vdash_R false : \mathsf{bool}$, the new state is well-typed. For composed guards $b \wedge g$, we have $\Delta \vdash_R b \wedge G : \mathsf{bool}$. There are two cases. First, if b reduces to *false* by (GUARD1) we get $\Delta \vdash_R false : \mathsf{bool}$. Second, if b reduces to *true* we get $\Delta \vdash_R g : \mathsf{bool}$. In both cases, the resulting configuration is well-typed in Δ.

Expressions. First consider a variable v in $(o, pv, pq, fds, (l, M[v]))$. Since the configuration is well-typed, v must be bound to a value *val* in a well-typed binding $\Delta \vdash_R T\ v\ val\ \mathsf{ok}$. Then $\Delta(val) \preceq T$, and $\Delta \vdash_R (o, pv, pq, fds, (l, M[val]))\ \mathsf{ok}$.

Consider $\mathsf{new}\ C()$ with $\Delta \vdash_R config\ (o, pv, pq, fds, (l, M[\mathsf{new}\ C()]))\ \mathsf{ok})$. We have $\Delta \vdash_R \mathsf{new}\ C() : T$ and a reduction by rule (RED-NEW) to a fresh object identifier *oid*. Let $\Delta' = \Delta[oid \mapsto C]$. Since *oid* is fresh, $oid \notin dom(\Delta)$ and $\Delta' \vdash_R config\ \mathsf{ok}$. By rule (VAR), $\Delta' \vdash_R oid : T$, so $\Delta' \vdash_R (o, pv, pq, fds, (l, M[oid]))\ \mathsf{ok}$.

For $fid.\mathsf{get}$, we have $\Delta \vdash_R fid.\mathsf{get} : T$ such that $\Delta(fid) = \mathsf{fut}(T)$, $\Delta \vdash_R (fid, mc, \mathsf{c}, val)\ \mathsf{ok}$, and a reduction by (RED-GET). Since the future is well-typed, $\Delta \vdash_R val : T$, and
$\Delta \vdash_R (o, pv, pq, fds, (l, M[val]))\ \mathsf{ok}$.

For $oid!m(\overline{val})$ with $\Delta \vdash_R config\ (o, pv, pq, fds, (l, M[oid!m(\overline{val})]))\ \mathsf{ok})$.

(CLASS)
$$\frac{\begin{array}{c} version(\Delta(C)) = vs \\ \forall I \in impl \cdot implements(C,I,\Delta) \\ \Delta + [\mathsf{this}\mapsto_v C] + [attr(C,\Delta)] \vdash_R mtds\ \mathsf{ok} \end{array}}{\Delta \vdash_R (C\#n, vs, impl, inh, ob, fds, mtds)\ \mathsf{ok}}$$

(AND)
$$\frac{\begin{array}{c} \Delta \vdash_R g_1 : \mathsf{bool} \\ \Delta \vdash_R g_2 : \mathsf{bool} \end{array}}{\Delta \vdash_R g_1 \wedge g_2 : \mathsf{bool}}$$

(COMPOSITION)
$$\frac{\begin{array}{c} \Delta \vdash_R s\ \mathsf{ok} \\ \Delta \vdash_R s'\ \mathsf{ok} \end{array}}{\Delta \vdash_R s; s'\ \mathsf{ok}}$$

(FUTURE)
$$\frac{\begin{array}{c} \Delta(fid) = \mathsf{fut}(T) \\ match(\Delta(oid), m, \overline{T} \rightarrow T, \Delta) \\ \Delta \vdash_R \overline{val} : \overline{T} \qquad mode = c \Rightarrow \Delta \vdash_R val : T \end{array}}{\Delta \vdash_R (fid, oid.m(\overline{val}), mode, val)\ \mathsf{ok}}$$

(ASSIGN)
$$\frac{\begin{array}{c} \Delta \vdash_R e : T' \\ T' \preceq \Gamma_V(v) \end{array}}{\Delta \vdash_R v := e\ \mathsf{ok}}$$

(POLL)
$$\frac{\Delta \vdash_R v : \mathsf{fut}(T)}{\Delta \vdash_R v? : \mathsf{bool}}$$

(EXTCALL)
$$\frac{\begin{array}{c} \Delta \vdash_R \overline{e} : T \quad \Delta \vdash_R e : I \\ match(I, m, \overline{T} \rightarrow T, \Delta) \end{array}}{\Delta \vdash_R e!m(\overline{e}) : \mathsf{fut}(T')}$$

(INTCALL)
$$\frac{\begin{array}{c} \Delta \vdash_R \overline{e} : T \\ match(\Delta(\mathsf{this}), m, \overline{T} \rightarrow T, \Delta) \end{array}}{\Delta \vdash_R \mathsf{this}!m(\overline{e}) : \mathsf{fut}(T')}$$

(AWAIT)
$$\frac{\Delta \vdash_R g : \mathsf{bool}}{\Delta \vdash_R \mathsf{await}\ g\ \mathsf{ok}}$$

(NEW)
$$\frac{\exists T' \in interfaces(\Delta(C)) \cdot T' \preceq T}{\Delta \vdash_R \mathsf{new}\ C() : T}$$

(CONDITIONAL)
$$\frac{\Delta \vdash_R b : \mathsf{bool} \quad \Delta \vdash_R s'\ \mathsf{ok} \quad \Delta \vdash_R s'\ \mathsf{ok}}{\Delta \vdash_R \mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ s'\ \mathsf{fi}\ \mathsf{ok}}$$

(GET)
$$\frac{\Delta(v) = \mathsf{fut}(T)}{\Delta \vdash_R v.\mathsf{get} : T}$$

(RELEASE)
$$\frac{}{\Delta \vdash_R \mathsf{release}\ \mathsf{ok}}$$

(NULL)
$$\frac{I \in dom(\Gamma_I)}{\Delta \vdash_R \mathsf{null} : I}$$

(SKIP)
$$\frac{}{\Delta \vdash_R \mathsf{skip}\ \mathsf{ok}}$$

Figure 10.8: The typing rules for runtime configurations.

Case 1 ($oid \neq \mathsf{this}$). We may assume $\Delta \vdash_R oid!m(\overline{val}) : \mathsf{fut}(T)$ and by rule (RED-CALL1) we get a fresh id fid. Let $\Delta' = \Delta[fid \mapsto \mathsf{fut}(T)]$. Obviously, $fid \notin dom(\Delta)$ so $\Delta' \vdash_R config\ \mathsf{ok}$. By (EXTCALL), we know that $\Delta \vdash_R \overline{val} : \overline{T}$ and $\Delta \vdash_R oid : I$ such that $match(I, m, \overline{T} \rightarrow T, \Delta)$. Since $\Delta(oid) \preceq I$, it follows that $match(\Delta(oid), m, \overline{T} \rightarrow T, \Delta)$, and that $\Delta' \vdash_R (fid, oid.m(\overline{val}), \mathsf{c}, \mathsf{null})\ \mathsf{ok}$.

Case 2 ($oid = \mathsf{this}$). We may assume $\Delta \vdash_R this!m(\overline{val}) : \mathsf{fut}(T)$ and by rule (RED-CALL2) we get a fresh id fid. The case is similar to Case 1, except we need to prove that $\Delta' \vdash_R (fid, n)\ \mathsf{ok}$ for the new constraint. This is immediate from (CONSTRAINT1).

Statements. For assignment $v := val$: Since $\Delta \vdash_R v := val\ \mathsf{ok}$, we know that $\Delta(val) \preceq \Delta(v)$. Consequently, $\Delta \vdash_R \Delta(v)\ v\ val\ \mathsf{ok}$, and the configuration is well-typed.

For skip, we have $(o, pv, pq, fds, (l, M[\mathsf{skip}; s]))$ which by (RED-SKIP) reduces to $(o, pv, pq, fds, (l, M[s]))$. Since $\Delta \vdash_R (o, pv, pq, fds, (l, M[\mathsf{skip}; s]))\ \mathsf{ok}$, $\Delta \vdash_R (o, pv, pq, fds, (l, M[s]))\ \mathsf{ok}$ by (COMPOSITION).

For conditionals $\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ s'\ \mathsf{fi}$, we have that $\Delta \vdash_R s\ \mathsf{ok}$ and $\Delta \vdash_R s'\ \mathsf{ok}$ since $\Delta \vdash_R \mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ s'\ \mathsf{fi}\ \mathsf{ok}$. If $b = true$, $\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ s'\ \mathsf{fi}$ reduces to s by (RED-COND1) and if $b = false$, $\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ s'\ \mathsf{fi}$ reduces to s' by (RED-COND2). In both cases, the new configuration is well-typed.

For release points release, (RED-RELEASE) gives us $(o, pv, pq, fds, (l, M[\mathsf{skip}]))$, which is well-typed since $\Delta \vdash_R (o, pv, pq, fds, (l, M[\mathsf{skip}]))$ ok.

For release points await g: By (RED-AWAIT), $(o, pv, pq, fds, (l, M[\mathsf{await}\ g]))$ is reduced to $(o, pv, pq, fds, (l, M[\mathsf{if}\ g\ \mathsf{then\ skip\ else\ release; await}\ g\ \mathsf{fi}]))$. By assumption $\Delta \vdash_R (o, pv, pq, fds, (l, M[\mathsf{await}\ g]))$ ok, so $\Delta \vdash_R g : \mathsf{bool}$. It follows that $\Delta \vdash_R (o, pv, pq, fds, (l, M[\mathsf{if}\ g\ \mathsf{then\ skip\ else\ release; await}\ g\ \mathsf{fi}]))$ ok.

Processes, Objects, Configurations. Process rescheduling. By (RED-RESCHEDULE), $(o, pv, p :: pq, fds, \mathsf{idle})$ reduces to (o, pv, pq, fds, p). By assumption we have $\Delta \vdash_R (o, pv, pq, fds, \mathsf{idle})$ ok, so $\Delta \vdash_R p$ ok and $\Delta \vdash_R pq$ ok, and consequently $\Delta \vdash_R (o, pv, pq, fds, p)$ ok.

Method binding. By (RED-BIND1) we must show that $(bind, C\#n, fid, oid.m(\overline{val}))$ is well-typed. We have that $\Delta \vdash_R (fid, oid.m(\overline{val}), \mathsf{s}, \mathsf{null})$ ok. It follows that $match(\Delta(oid), m, \overline{T} \to T, \Delta)$, $\Delta(fid) = \mathsf{fut}(T)$, and $\Delta \vdash_R \overline{val} : \overline{T}$. Thus, we have $\Delta \vdash_R (bind, C\#n, fid, oid.m(\overline{val}))$.

By (RED-BIND2), we have $match((C;\overline{C}'), m, \overline{T} \to T, \Delta)$ and $lookup(m(\overline{val}), \overline{T} \to T, fid, mtds) =$ error where $mtds$ are the methods defined in C. Let $\overline{C}$ be the superclasses of C, then $match(C, m, \overline{T} \to T, \Delta) = lookup(m(\overline{val}), \overline{T} \to T, fid, mtds) \neq \mathsf{error} \vee match(\overline{C}, m, \overline{T} \to T, \Delta)$. It follows that $match(\overline{C}, m, \overline{T} \to T, \Delta)$, and consequently $\Delta \vdash_R (bind, (\overline{C};\overline{C}'), fid, oid.m(\overline{val}))$.

For (RED-BIND3) we need a term $(bind, \varepsilon, fid, oid.m(\overline{val}))$ such that $\Delta \vdash_R (bind, \varepsilon, fid, oid.m(\overline{val}))$ ok. However, this would require that $match(\varepsilon, m, \overline{T} \to T, \Delta)$, which is impossible.

By (RED-BIND4) we get $lookup(m(\overline{val}), \overline{T} \to T, fid, mtd\ mtds) = process$. By assumption $\Delta \vdash_R mtd$ ok, $\Delta \vdash_R \mathsf{fut}(T)\ destiny\ fid$ ok since $\Delta(destiny) = \Delta(fid) = \mathsf{fut}(T)$, and $\Delta \vdash_R T\ return\ default(T)$ ok. Consequently, we get $\Delta \vdash_R process$ ok. It follows that $\Delta \vdash_R (bind, oid, process)$ ok.

For (RED-RETURN), we have a well-typed process $\Delta \vdash_R (fds; \mathsf{return}\ val)$ ok such that $fds(destiny) = fid$. It follows that $\Delta \vdash_R val : T$ and $\Delta(fid) = \mathsf{fut}(T)$. Consequently, $\Delta \vdash_R (fid, m(\overline{val}), \mathsf{c}, val)$ ok. ■

Remark that when *config* is reduced to normal form, we know that the fields of all idle objects are the same as those of their class definitions. This is necessary to ensure type correctness for Rule (RED-BOUND) when the class definition changes, as new processes could otherwise not be well-typed in the objects. In fact this is an implicit assumption of Theorem 4, which would otherwise not be valid.

10.A.1 Dynamic Class Operations

In order for a dynamic class operation to be effectuated, i.e., applied to a class, the statically type inferred class constraints *dep* must be satisfied by the runtime configuration. This means that runtime classes have evolved to at least the statically required versions. If the dynamic class operation is a simplification upgrade, we must ensure that no process will make references to removed fields, superclasses or invoke removed methods. Technically this means that all processes in objects must be from a stage which is at least as recent as the statically required stage, checked by the object constraint set *odep*. The predicates *resolve* and *oresolve* verifie the class and object constraints based on the current runtime typing environment.

$$\begin{array}{c}\text{(New-Class)}\\ \dfrac{\begin{array}{c} resolve(\Delta, dep) \qquad \Delta' = \Delta + (C\#1, 1, impl, inh, ob, fds, mtds) \\ \Delta' \vdash_R (C\#1, 1, impl, inh, ob, fds, mtds)\ upconfig\ \mathsf{ok}\end{array}}{\Delta \vdash_R (new, C, impl, inh, fds, mtds, dep)\ upconfig\ \mathsf{ok}}\end{array}$$

$$\begin{array}{c}\text{(Class-Extend)}\\ \dfrac{\begin{array}{c} resolve(\Delta, dep) \qquad \Delta_C(C) = (C\#n, vs, impl, inh, ob, fds, mtds) \\ \Delta' = \Delta + [C \mapsto_C (C\#n+1, vs+1, (impl; impl'), (inh; inh'), ob, (fds; fds'), (mtds \oplus mtds'))] \\ \Delta' \vdash_R (C\#n+1, vs+1, (impl; impl'), (inh; inh'), ob, (fds; fds'), (mtds \oplus mtds'))\ upconfig\ \mathsf{ok}\end{array}}{\Delta \vdash_R (ext, C, impl', inh', fds', mtds', dep)\ upconfig\ \mathsf{ok}}\end{array}$$

$$\begin{array}{c}\text{(Class-Simplify)}\\ \dfrac{\begin{array}{c} resolve(\Delta, dep) \qquad \Delta_C(C) = (C\#n, vs, impl, inh, ob, fds, mtds) \qquad upconfig = upg\ config \\ \Delta' = \Delta + [C \mapsto_C (C\#n+1, vs+1, impl, (inh \setminus inh'), ob, (fds \setminus fds'), (mtds \setminus mtds'))] \\ \Delta' \vdash_R (C\#n+1, vs+1, impl, (inh \setminus inh'), ob, (fds \setminus fds'), (mtds \setminus mtds'))\ classes(\Delta')\ upg\ \mathsf{ok}\end{array}}{\Delta \vdash_R (simp, C, inh', fds', mtds', dep, odep)\ upconfig\ \mathsf{ok}}\end{array}$$

Figure 10.9: The typing rules for runtime dynamic class upgrades.

Definition 5 Let Δ be the runtime environment, *dep* the type inferred class constraints and *odep* the runtime object constraints. Define $resolve(\Delta,\ dep)$ and $oresolve(\Delta,\ odep)$ as follows:

$$\begin{array}{lcl} resolve(\Delta, \emptyset) & = & true \\ resolve(\Delta, \{(C, n)\}) & = & version(\Delta(C)) \geq n \\ resolve(\Delta, dep \cup dep') & = & resolve(\Delta, dep) \cup\ resolve(\Delta, dep') \\ & & \\ oresolve(\Delta, \emptyset) & = & true \\ oresolve(\Delta, \{(o, C\#n)\}) & = & \Delta_{old}(o)) \geq n \\ oresolve(\Delta, odep \cup odep') & = & oresolve(\Delta, odep) \cup\ oresolve(\Delta, odep') \end{array}$$

Define the union operation $mtds \oplus mtds'$ with right priority, which retains methods in $mtds'$ in case of name conflicts.

Definition 6 Let $T, T', \overline{T}$ and $\overline{T}'$ be types, m a method name, and *mtds* and *mtds*' sets of methods. Define $mtds \oplus mtds'$ as follows:

$$\begin{array}{l} (\varepsilon \oplus mtds) = (mtds \oplus \varepsilon) = mtds \\ (T\ m(\overline{T\ x})\{process\}\ mtds \oplus T'\ m(\overline{T'\ x'})\{process'\}\ mtds') = \\ \qquad\qquad T'\ m(\overline{T'\ x'})\{process'\}\ (mtds \oplus mtds') \\ (mtds \oplus T\ m(\overline{T\ x})\{process\}\ mtds') = \\ \qquad\qquad T\ m(\overline{T\ x})\{process\}\ (mtds \oplus mtds')\ otherwise \end{array}$$

We define a configuration *upconfig* based on the runtime syntax in Section 10.4.2:

$$upconfig \quad ::= \quad config \mid upg \mid upconfig\ upconfig$$

For the properties of dynamic class operations in the runtime system, we assume that upgrades are injected into the runtime configuration in the same order as they were type checked.

Definition 7 Let P be a well-typed program with the corresponding initial configuration *init*, and let $U_1, \cdots, U_n$ be a sequence of well-typed class upgrade operations such that $\Gamma^i \vdash U_i$. Let upg_i be the corresponding runtime representation of U_i. A run of such a system *order-respecting* for the sequence of upgrades if message upg_{i+1} is injected into the system after message upg_i for all $1 \leq i \leq n$.

Lemma 5 *Let P be a well-typed program such that $\Gamma^0 \vdash P$ and let a run init $\rightarrow$ upconfig order-respecting for $U_1, \cdots, U_i$. Then we have that if $\Delta^i \vdash_R$ upconfig* ok*, $\Gamma^{i+1} \vdash U_{i+1}$ and upg_{i+1} corresponds to U_{i+1} then there is a Δ^{i+1} such that $\Delta^{i+1} \vdash_R$ upconfig upg_{i+1}* ok.

Proof. We consider a class C in the program and the update operation
$\mathsf{update}\ C\ \mathsf{extends}\ \overline{C}\ \mathsf{implements}\ \overline{I}\ \{\overline{T\ f};\overline{M}\}$ and $\mathsf{simplify}\ C\ \mathsf{retract}\ \overline{C}\ \{\overline{T\ f};\overline{M}_s\}$.

We first consider an class extension upgrade. By assumption

$$\Gamma^{i+1} + [(C,n) \mapsto S] \vdash \mathsf{update}\ C\ \mathsf{extends}\ \overline{C}\ \mathsf{implements}\ \overline{I}\ \{\overline{T\ f};\overline{M}\},$$

where

$$\Gamma^{i+1} = \Gamma^i + [C \mapsto ((impl';impl),(inh';inh),(fds';fds),(mtds' \oplus mtds))]$$

Assume first that this is the initial upgrade of the program, so $i = 0$. By Lemma 3, $\Delta \vdash_R init$ ok and by Theorem 4, $\Delta^0 \vdash_R config$ ok. Specifically, $\Delta^0 \vdash_R (C\#0, 0, impl', inh', ob, fds', mtds')$ ok. Note that $\Delta^0(C) = \Gamma^0(C)$ for all classes C. By assumption

$$\Gamma^1 + [(C,1) \mapsto S] \vdash \mathsf{update}\ C\ \mathsf{extends}\ \overline{C}\ \mathsf{implements}\ \overline{I}\ \{\overline{T\ f};\overline{M}\}.$$

We must show that $\Delta^0 \vdash_R config\ (ext, C, impl, inh, fds, mtds, S)$ ok. Since all classes have initial versions in Γ^0, $\forall (D,n) \cdot (D,n) \in S \Rightarrow n = 0$. It follows that $resolve(\Delta^0, S)$. Let $\Delta^1 = \Delta^0 + [C \mapsto_C (C\#1, 1, (impl';impl), (inh';inh), ob, (fds';fds), (mtds' \oplus mtds))]$. Note that $\Delta^1(C) = \Gamma^1(C)$.

Next, we show that the new class definition is well-typed in Δ^1. We need to show that $\forall I \in impl';impl \cdot implements(C, I, \Delta^1)$. Consider first the old interfaces $impl'$ of C. By assumption, $\forall I \in impl' \cdot implements(C, I, \Delta)$ and from the static typing, we have $refines(mtds, mtds')$, so we get $\forall I \in impl' \cdot implements(C, I, \Delta^1)$. Now consider the new interfaces $impl$. It follows from the static typing, that $\forall I \in impl \cdot implements(C, I, \Delta^1)$. By assumption, $\Delta^0 + [\mathsf{this} \mapsto_v C] + [attr(C, \Delta^0)] \vdash_R mtd$ ok for all old method definitions $mtd \in mtds'$. Since the class extension only extends the previous version of the class with new fields, new superclasses and new interfaces, and refines old methods, $\Delta^1 + [\mathsf{this} \mapsto_v C] + [attr(C, \Delta^1)] \vdash_R mtd$ ok for all $mtd \in mtds'$. By the static typing rule (CLASS-EXTEND), $\Gamma^1 + [\mathsf{this} \mapsto_v C] + [attr(C, \Gamma^1)] \vdash mtd$ for all new method definitions $mtd \in mtds$. Since $\Delta^1(C) = \Gamma^1(C)$, we get $\Delta^1 + [\mathsf{this} \mapsto_v C] + [attr(C, \Delta^1)] \vdash_R mtd$ ok for all $mtd \in mtds$. Consequently the new class definition is well-typed in Δ^1; i.e., $\Delta^1 \vdash_R (C\#1, 1, (impl';impl), (inh';inh), ob, (fds';fds), (mtds' \oplus mtds))$ ok.

It remains to show that $\Delta^1 \vdash_R config$ ok. We first consider a process in an object $((o, C'\#n), pv, pq, fds, (l, S; \mathsf{return}\ e))$ of class C'. By assumption, this process is well-typed in Δ^0. We show that the process remains well-typed in Δ^1. The proof is by induction over runtime terms.

For *expressions* x, the process includes the local variables so x is not affected by any method redefinition. Similar for f as the class extension does not redefine fields. For $\mathsf{new}\ C()$ such that $\Delta^0 \vdash_R \mathsf{new}\ C() : T$. However, a well-typed class extension may only add new interfaces to C, so all old interfaces remain implemented. Thus $\exists T' \in \text{interfaces}(\Delta^1(C)) \cdot T' \preceq T$ and $\Delta^1 \vdash_R \mathsf{new}\ C() : T$. Now consider an external call $e'!m(\overline{e})$ and assume $\Delta^0 \vdash_R e'!m(\overline{e}) : \mathsf{fut}(T)$ so $\Delta^0 \vdash_R e' : I$ and $\Delta^0 \vdash_R \overline{e} : \overline{T}$. By the induction hypothesis we have $\Delta^1 \vdash_R e' : I$ and $\Delta^1 \vdash_R \overline{e} : \overline{T}$. For external calls, $match(I, m, \overline{T} \rightarrow T, \Delta^1)$ holds because $\Delta^0(I) = \Delta^1(I)$. For an internal call we have $e' = \mathsf{this}$ and $match(C', m, \overline{T} \rightarrow T, \Delta^0)$ Say that the matching definition of m is in a class C'', where C'' may be C or a superclass of C. If m is *redefined* in C, the new definition in Δ^1 will be matching because it must refine the old definition from Δ^0. Otherwise, the old definition is still available in C'' and matching is guaranteed to succeed even if a new definition of m has been introduced in C. Consequently, $match(\Delta^1(\mathsf{this}), m, \overline{T} \rightarrow T, \Delta^1)$ holds. For $fid.\mathsf{get}$, we may assume that $\Delta^0 \vdash_R fid : \mathsf{fut}(T)$. Since Δ^1 does not modify the typing of futures, $\Delta^1 \vdash_R fid : \mathsf{fut}(T)$. Similarly for null and $fid?$.

For *statements*, first consider the assignment $v := e$, we assume $\Delta^0 \vdash_R e : T$ and $T \preceq \Delta^0(v)$, since the types of old variables remain unchanged, we have $\Delta^1 \vdash_R e : T$ and $T \preceq \Delta^1(v)$, so $\Delta^1 \vdash_R v := e\ \mathsf{ok}$. For release points $\Delta^1 \vdash_R \mathsf{await}\ g\ \mathsf{ok}$ follows from the induction hypothesis. Similar for $\mathsf{release}$, skip, $\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ s'\ \mathsf{fi}$ and $s; s'$.

For *processes* $(fds, s; \mathsf{return}\ e)$, by assumption $\Delta^0 \vdash_R fds\ \mathsf{ok}$, $\Delta^0 \vdash_R s\ \mathsf{ok}$, and $\Delta^0 \vdash_R e : T$ By the induction hypothesis, $\Delta^1 \vdash_R fds\ \mathsf{ok}$, $\Delta^1 \vdash_R s\ \mathsf{ok}$, and $\Delta^1 \vdash_R e : T$. Consequently, $\Delta^1 \vdash_R (fds, s; \mathsf{return}\ e)\ \mathsf{ok}$. *Objects*, *futures*, *bind* and *bound* messages, and other *classes* follow directly from the induction hypothesis.

We now consider the simplification upgrade. By assumption

$$\Gamma^{i+1} + [(C, n) \mapsto S] \vdash \mathsf{simplify}\ C\ \mathsf{retract}\ \overline{C}\ \{\overline{T\ f}; \overline{M_s}\},$$

where

$$\Gamma^{i+1} = \Gamma^i + [C \mapsto (impl', (inh' \setminus inh), (fds' \setminus fds), (mtds' \setminus mtds))]$$

For the initial upgrade of the program, $i = 0$. By Lemma 3, $\Delta \vdash_R init\ \mathsf{ok}$ and by Theorem 4, $\Delta^0 \vdash_R config\ \mathsf{ok}$. Specifically,

$$\Delta^0 \vdash_R (C\#0, 0, impl', inh', ob, fds', mtds')\ \mathsf{ok}.$$

Note that $\Delta^0(C) = \Gamma^0(C)$ for all classes C. By assumption

$$\Gamma^1 + [(C, 1) \mapsto S] \vdash \mathsf{update}\ C\ \mathsf{simplify}\ C\ \mathsf{retract}\ \overline{C}\ \{\overline{T\ f}; \overline{M_s}\}.$$

We must show that $\Delta^0 \vdash_R config\ (simp, C, inh, fds, mtds, S, O)\ \mathsf{ok}$. Since all classes have initial versions in Γ^0, $\forall (D, n) \cdot (D, n) \in S \Rightarrow n = 0$. It follows that $resolve(\Delta^0, S)$.

Let $\Delta^1 = \Delta^0 + [C \mapsto_C (C\#1, 1, impl', (inh' \setminus inh), ob, (fds' \setminus fds), (mtds' \setminus mtds))]$. Note that $\Delta^1(C) = \Gamma^1(C)$. Then

$$\Delta^1 \vdash_R (C\#1, 1, impl', (inh' \setminus inh), ob, (fds' \setminus fds), (mtds' \setminus mtds))\ \mathsf{ok}$$

follows from (CLASS-SIMPLIFY). Since the class simplification is statically well-typed, all interfaces supported by C in Δ^i are still supported in Δ^{i+1}. Consequently, $\Delta^{i+1} \vdash_R classes(\Delta^{i+1})$ ok.

We now consider the induction step. First consider

$$\Gamma^{i+1} + [(C,n) \mapsto S] \vdash \mathsf{update}\ C\ \mathsf{extends}\ \overline{C}'\ \mathsf{implements}\ \overline{I}'\ \{\overline{T\ f};\overline{M}'\}$$

Let $\Delta^i \vdash_R upconfig$ ok and show that $\Delta^i \vdash_R upconfig\ (ext, C, impl, inh, fds, mtds, S)$ ok.

Since $\Delta^i \vdash_R upconfig$ ok, we have that $\forall\ upg \in upconfig\ \cdot\ resolve(\Delta^i, S')$, where $upg \in \{(ext, D, impl, inh, fds, mtds, S'), (simp, D, inh, fds, mtds, S', O), (new, D, impl, inh, fds, mtds, S')\}$. Since the run is order-respecting, we have that $\Delta^i(D) = \Gamma^i(D)$ for all classes D and Δ^i contains all versions of classes that were needed for the static typechecking. Consequently $resolve(\Delta^i, S)$.

Let $\Delta^{i+1} = \Delta^i + [C \mapsto_C (C\#n' + 1, n, (impl'; impl), (inh'; inh), ob, (fds'; fds), (mtds' \oplus mtds))]$. Remark that $\Gamma^{i+1} = \Gamma^i + [C \mapsto_C ((impl'; impl), (inh'; inh), (fds'; fds), (mtds' \oplus mtds))]$, so $\Delta^{i+1}(C) = \Gamma^{i+1}(C)$. By (CLASS-EXTEND) we must show that

$$\Delta^{i+1} \vdash_R (C\#n' + 1, n, (impl'; impl), (inh'; inh), ob, (fds'; fds), (mtds' \oplus mtds))\ upconfig_i\ \mathsf{ok}$$

We first consider the new class definition and show

$$\Delta^{i+1} \vdash_R (C\#n' + 1, n, (impl'; impl), (inh'; inh), ob, (fds'; fds), (mtds' \oplus mtds))\ \mathsf{ok}$$

We first consider the interfaces of C. Since $\Delta^i \vdash_R upconfig_i$ ok, by assumption, $\forall I \in impl' \cdot implements(C, I, \Delta^i)$ and from the static typing, we have $refines(mtds, mtds')$, so we get $\forall I \in impl' \cdot implements(C, I, \Delta^{i+1})$. From the static typing, we know that
$\forall I \in impl \cdot implements(C, I, \Gamma^{i+1})$. Consequently, we have that $\forall I \in impl\ \cdot\ implements(C, I, \Delta^{i+1})$.

Next, we consider the methods of C. By assumption, $\Delta^i + [\mathsf{this} \mapsto_v C] + [attr(C, \Delta^i)] \vdash_R mtd$ ok for all $mtd \in mtds'$. Since the class extension only extends the previous version of the class with new fields, new superclasses and new interfaces, and refines old methods, $\Delta^{i+1} + [\mathsf{this} \mapsto_v C] + [attr(C, \Delta^{i+1})] \vdash_R mtd$ ok for all $mtd \in mtds'$. For new methods, by the type system $\Gamma^{i+1} + [\mathsf{this} \mapsto_v C] + [attr(C, \Gamma^{i+1})] \vdash mtd$ for all $mtd \in mtds$ and $resolve(\Delta^{i+1}, S)$ for a dependency mapping S. Since $\Delta^{i+1}(C) = \Gamma^{1+1}(C)$, we get $\Delta^{i+1} + [\mathsf{this} \mapsto_v C] + [attr(C, \Delta^{i+1})] \vdash_R mtd$ ok for all $mtd \in mtds$.

Finally, we need to show that $\Delta^{i+1} \vdash_R upconfig$ ok. As before, we consider a process inside an object. The proof is similar to the base case above.

Consider the induction step

$$\Gamma^{i+1} + [(C,n) \mapsto S] \vdash \mathsf{simplify}\ C\ \mathsf{retract}\ \overline{C}\ \{\overline{T\ f};\overline{M}_S\}$$

Let $\Delta^i \vdash_R upconfig$ ok. We show $\Delta^i \vdash_R upconfig\ (simp, C, inh, fds, mtds, S, O)$ ok.

The proof is similar to the case for class extension. Since $\Delta^i \vdash_R upconfig$ ok, we have that $\forall\ upg \in upconfig\ \cdot\ resolve(\Delta^i, S')$, where $upg \in \{(ext, D, impl, inh, fds, mtds, S'),$
$(simp, D, inh, fds, mtds, S', O), (new, D, impl, inh, fds, mtds, S')\}$. Since the run is order-respecting, $\Delta^i(D) = \Gamma^i(D)$ for all classes D and Δ^i contains all versions of classes that were needed for the static typechecking. Consequently $resolve(\Delta^i, S)$ holds.

Let $\Delta^{i+1} = \Delta^i + [C \mapsto_C (C\#n'+1, n, impl', (inh' \setminus inh), ob, (fds' \setminus fds), (mtds' \setminus mtds))]$. Remark that $\Gamma^{i+1} = \Gamma^i + [C \mapsto_C (impl', (inh' \setminus inh), (fds' \setminus fds), (mtds' \setminus mtds))]$, so $\Delta^{i+1}(C) = \Gamma^{i+1}(C)$. By (CLASS-SIMPLIFY) we must show that

$$\Delta^{i+1} \vdash_R (C\#n'+1, n, impl', (inh' \setminus inh), ob, (fds' \setminus fds), (mtds' \setminus mtds))\ classes(\Delta^{i+1})\ \mathsf{ok}$$

Let D be C or a subclass of C and let I be an interface of D in Γ^i. Then, from the static typechecking, we have that $implements(D, I, \Gamma^{i+1})$ Consequently, $implements(D, I, \Delta^{i+1})$. Similarly, from the static typechecking we have $\Gamma^{i+1} + [\mathsf{this} \mapsto_v D] + [attr(D, \Gamma^{i+1})] \vdash \Gamma^{i+1}(D).mtds$. It follows that $\Delta^{i+1} + [\mathsf{this} \mapsto_v D] + [attr(D, \Delta^{i+1})] \vdash_R \Delta^{i+1}(D).mtds\ \mathsf{ok}$. All other classes are well-typed by the induction hypothesis.

■

Definition 8 Let $procstage(p)$ be the stage of the class when a process p is bound.

It is obvious that the process stage of a process does not change during process execution. We show that the process version set of an object identifies all nonterminating processes in the object. Moreover, for all process, there is a correspondence between its stored process version and its procstage.

Lemma 6 *Let P be a well-typed program with initial configuration init and let init $\longrightarrow_{up}$upconfig. Let $(o, pv, pq, fds, active)$ be an object in upconfig and let $p \in \{active\} \cup pq$ be a process with $procstage(p) = n$, $p = (l, s; \mathsf{return}\ e)$, $l(destiny) = fid$. Then there is $(fid, n') \in pv$ such that $n' \leq n$.*

Proof. The stage number of an object is always less than or equal to the stage of its class. The proof proceeds by induction over the reduction steps, since n' reflects the stage number of the object before the method call is bound.

■

Note that in (CLASS-SIMPLIFY) the updated configuration Δ' does not give well-typed objects in *upconfig*. We show that only when object constraints are satisfied, *upconfig* is well-typed.

Lemma 7 *Assume that*

$$\Delta \vdash_R (simpl, C, inh, fds, mtds, S, O)(C\#n, vs, impl, inh', ob, fds', mtds')\ upconfig\ \mathsf{ok}$$

and $\mathsf{resolve}(\Delta, S)$. *Define the typing context*

$$\Delta' = \Delta + [C \mapsto (C\#n+1, vs+1, impl, (inh' \setminus inh), ob, (fds' \setminus fds), (mtds' \setminus mtds))]$$

and let

$$\begin{aligned} odep = \{(o, D\#n') \mid\ & o \in ob_D \wedge (D, n) \in S \\ & \wedge (D\#n', vs'_D, impl_D, inh_D, ob_D, fds_D, mtds_D) \in upconfig\} \end{aligned}$$

If $\mathsf{oresolve}(\Delta, O \cup odep)$ *then*

1. $\Delta' \vdash_R upconfig\ \mathsf{ok}$.

2. *If* $\Delta \vdash_R (o, pv, pq, fds, active)$ ok *then* $\Delta' \vdash_R (o, pv, pq, (fds' \setminus fds), active)$ ok.

Proof. Let $upconfig = upg\ config$. By (CLASS-SIMPLIFY), $\Delta' \vdash_R classes(\Delta')\ upg$ ok. We need to show that $\Delta' \vdash_R config$, assuming that $oresolve(\Delta, O \cup odep)$. Consider an object $(o, pv, pq, fds, active)$ in *config* of some class D. Observe that if $(D, n) \in S$ (for some n) and $oresolve(\Delta, O \cup odep)$, then $oldest(pv) \geq n'$. It follows that all processes in objects of the class D are from a stage of D which is at least as recent as statically required. Consequently $\Delta' \vdash_R (o, pv, pq, (fds' \setminus fds), active)$ ok. Observe that for any message $(bind, \overline{C}, fid, mc)$ to o, $(fid, n'') \in pv$. It follows that $n'' \geq oldest(pv) \geq n'$ and consequently $\Delta' \vdash_R (bind, \overline{C}, fid, mc)$ ok. Similarly, $\Delta' \vdash_R (bound, oid, process)$ ok. All other objects of some class D' not in S and messages to these objects and classes are well-typed by the assumption, since $\Delta(D') = \Delta'(D')$. Thus $\Delta' \vdash_R upconfig$ ok.

■

We show that resolving class constraints for a class extension message in a well-typed configuration gives a well-typed configuration.

Lemma 8 *If* $\Delta \vdash_R (ext, C, impl, inh, fds, mtds, dep \cup \{(D, n)\})\ upconfig$ ok *and* $(ext, C, impl, inh, fds, mtds, dep \cup \{(D, n)\})\ upconfig \longrightarrow_{up} (ext, C, impl, inh, fds, mtds, dep)\ upconfig$, *then* $\Delta \vdash_R (ext, C, impl, inh, fds, mtds, dep)\ upconfig$ ok.

Proof. By assumption $\Delta \vdash_R upconfig$ ok and since we apply the rule (DEP) we know that class D is of version n in *config*. It follows that $version(\Delta(D)) \geq n$ and consequently $resolve(\Delta, dep) = resolve(\Delta, dep \cup \{(D, n)\})$. Consequently, the dependency on version n of class D is already satisfied at runtime and the constraint may be removed.

■

We show that resolving class constraints for a class simplification message in a well-typed configuration gives a well-typed configuration.

Lemma 9 *If* $\Delta \vdash_R (simp, C, inh, fds, mtds, dep \cup \{(D, n)\}, odep)\ upconfig$ ok *and* $upconfig = (D\#n', vs, impl, inh, ob, fds, mtds)\ upconfig'$ *and* $(simp, C, inh, fds, mtds, dep \cup \{(D, n)\}, odep)\ upconfig \longrightarrow_{up} (simp, C, inh, fds, mtds, dep, odep \cup odep')\ upconfig$ *where* $odep' = \{(o, D'\#n') | o \in ob\}$ *then* $\Delta \vdash_R (simp, C, inh, fds, mtds, dep, odep \cup odep')\ upconfig$ ok.

Proof. Similar to 8. By assumption $\Delta \vdash_R upconfig$ ok and since we apply the rule (DEP-SIMPLIFY) we know that class D is of version n $(vs = n)$ in *config*. It follows that $version(\Delta(D)) \geq n$ and consequently $resolve(\Delta, dep) = resolve(\Delta, dep \cup \{(D, n)\})$. Consequently, the dependency on version n of class D is already satisfied at runtime and the constraint may be removed. Consequently, $\Delta \vdash_R (simp, C, inh, fds, mtds, dep, odep \cup odep')\ upconfig$ ok.

■

We show that resolving object constraints for a class simplification message in a well-typed configuration gives a well-typed configuration.

Lemma 10 *If* $\Delta \vdash_R (simp, C, inh, fds, mtds, dep, odep \cup \{(o, D\#n)\})\ upconfig$ ok *and* $(simp, C, inh, fds, mtds, dep, odep \cup \{(o, D\#n)\})\ upconfig \longrightarrow_{up} (simp, C, inh, fds, mtds, dep, odep)\ upconfig$ *then* $\Delta \vdash_R (simp, C, inh, fds, mtds, dep, odep)\ upconfig$ ok.

Proof. It follows from the induction hypothesis

■

Finally, Lemma 11 and Lemma 12 show that when all constraints are satisfied, we can safely apply the upgrades and maintain the well-typedness of runtime configurations.

Lemma 11 *If* $\Delta \vdash_R (ext, C, impl', inh', fds', mtds', \emptyset)\ (C\#n, vs, impl, inh, ob, fds, mtds)\ upconfig\ \mathsf{ok}$ *and*

$$\begin{aligned}&(ext, C, impl', inh', fds', mtds', \emptyset)\ (C\#n, vs, impl, inh, ob, fds, mtds)\ upconfig\\ &\longrightarrow_{up}(C\#n+1, vs+1, (impl; impl'), (inh; inh'), ob, (fds; fds'), (mtds \oplus mtds'))\\ &\quad upconfig\end{aligned}$$

then there exists a typing context Δ' *such that* $\Delta' \vdash_R upconfig\ \mathsf{ok}$

Proof. By assumption, $\Delta \vdash_R (ext, C, impl', inh', fds', mtds', \emptyset)$ $(C\#n, vs, impl, inh, ob, fds, mtds)\ upconfig\ \mathsf{ok}$. It follows that

$$\Delta' \vdash_R (C\#n+1, vs+1, (impl; impl'), (inh; inh'), ob, (fds; fds'), (mtds \oplus mtds'))\ upconfig\ \mathsf{ok},$$

where

$$\Delta' = \Delta + [C \mapsto_C (C\#n+1, vs+1, (impl; impl'), (inh; inh'), ob, (fds; fds'), (mtds \oplus mtds'))].$$

■

Lemma 12 *If* $\Delta \vdash_R (simp, C, inh', fds', mtds', \emptyset, \emptyset)\ (C\#n, vs, impl, inh, ob, fds,$ $mtds)\ upconfig\ \mathsf{ok}$ *and*

$$\begin{aligned}&(simp, C, inh', fds', mtds', \emptyset, \emptyset)\ (C\#n, vs, impl, inh, ob, fds, mtds)\ upconfig\\ &\longrightarrow_{up}(C\#n+1, vs+1, impl, (inh \setminus inh'), ob, (fds \setminus fds'), (mtds \setminus mtds'))\ upconfig\end{aligned}$$

then there exists a typing context Δ' *such that* $\Delta' \vdash_R upconfig\ \mathsf{ok}$

Proof. By assumption,

$$\Delta \vdash_R (simp, C, inh', fds', mtds', \emptyset, \emptyset)\ (C\#n, vs, impl, inh, ob, fds, mtds)\ config\ \mathsf{ok}$$

It follows from (CLASS-SIMPLIFY) that

$$\Delta' \vdash_R (C\#n+1, vs+1, impl, (inh \setminus inh'), ob, (fds \setminus fds'), (mtds \setminus mtds'))\ \mathsf{ok}$$

where

$$\Delta' = \Delta + [C \mapsto_C (C\#n+1, vs+1, impl, (inh \setminus inh'), ob, (fds \setminus fds'), (mtds \setminus mtds'))]$$

and since by Lemma 7, $\Delta' \vdash_R upconfig\ \mathsf{ok}$.

■

We can now show the subject reduction property for dynamic classes.

Theorem 13 (Subject reduction for dynamic classes) *Let P be a well-typed program and let $init \rightarrow upconfig$ order-respecting upgrades $upg_1, \cdots, upg_i$. Then there is a typing context Δ such that $\Delta \vdash_R upconfig$ ok.*

Proof. The proof is by induction on the length of the reduction from *init* to *upconfig*.

Base case. It follows from Lemma 3 that there is a typing context Δ such that $\Delta \vdash_R init$ ok.

Induction step. We assume that $\Delta \vdash_R upconfig$ ok for some Δ and show that if

$$upconfig \longrightarrow_{up} upconfig'$$

in a single reduction step, then there is a typing context Δ' such that $\Delta' \vdash_R upconfig'$ ok. The proof proceeds by cases for the different reductions.

Case 1. Let $upconfig = config\ upconfig0$ and $upconfig' = config'\ upconfig0$ and consider the standard reduction rules of Figure 10.4. Then by Theorem 4, $\Delta' \vdash_R config'$ ok.

Case 2. The rule (CLASS-INH) only changes the stage number of a class, which preserves well-typedness.

Case 3. Rule (DEP) follows from Lemma 8 for extend messages and similar for messages for new classes.

Case 4. Rule (DEP-SIMPLIFY) follows from Lemma 9.

Case 5. Rule (DEP-OBJECT) follows from Lemma 10.

Case 6. Rule (NEW-CLASS) is trivial.

Case 7. Rule (EXTEND-CLASS) follows from Lemma 11.

Case 8. Rule (SIMPLIFY-CLASS) follows from Lemma 12.

Case 9. Rule (OBJ-STATE) and consider
$upconfig = ((oid, C\#n), pv, pq, fds, \mathsf{idle})\ (C\#n', vs, impl, inh, ob, fds, mtds)\ upconfig''$ and
$upconfig' = ((oid, C\#n'), pv, pq, fds', \mathsf{idle})\ (C\#n', vs, impl, inh, ob, fds, mtds)\ upconfig''$.

We must show that $\Delta' \vdash_R ((oid, C\#n'), pv, pq, fds', \mathsf{idle})$ ok. We assume that if $\Delta \vdash_R fds$ ok and $\Delta \vdash_R (C\#n', vs, impl, inh, ob, fds, mtds)$ ok, then $\Delta \vdash_R transf(fds, attr(C))$ ok. Consequently, $\Delta \vdash_R fds'$ ok. We must show that the processes in *pq* are well-typed with the new fields of the object. There are two possibilities why rule (OBJ-STATE) has become applicable: either an *ext* or a *simp* message has been consumed by rules (EXTEND-CLASS) or (SIMPLIFY-CLASS).

Consider (EXTEND-CLASS) first. By assumption, this extend-message was well-typed. In this case, fds' is an extension of *fds* and $\Delta \vdash_R config''$ ok follows from Lemma 5.

Consider now (SIMPLIFY-CLASS) for some class C. By assumption, this simplify-message was well-typed and has been applied. Let Δ_o be the typing context at the application time. It follows from Lemma 12 that $oresolve(\Delta_0, odep)$, where $(oid, m) \in odep$ for some $m \leq n'$. By Lemma 7, the typing context Δ_1 which updates Δ_0 with the simplified definition of C gives us $\Delta_1 \vdash_R ((oid, C\#n'), pv, pq, fds', \mathsf{idle})$ ok

Finally, consider rule (UPDATE). By assumption, the update messages are injected into the runtime in the order they are statically analyzed. We consider the last injected message, upg_i in which case well-typedness is covered by Lemma 5. ■