An Object-Oriented Event Calculus

Jeremiah S. Patterson

TR #02-08
July 2002

Keywords: Implicit invocation, event handling, control primitives, control abstractions, $\rho\varsigma$-calculus, operational semantics, Eventua language, Abadi and Cardelli's object calculus.

2001 CR Categories: D.1.m [*Software Engineering*] Miscellaneous — event-based programming; D.3.1 [*Programming Languages*] Formal Definitions and Theory — semantics; D.3.2 [*Programming Languages*] Languages Classifications — object-oriented languages, very high level languages; D.3.3 [*Programming Languages*] Languages Constructs and Features — abstract data types, control structures, procedures, functions, and subroutines; D.3.m [*Programming Languages*] Miscellaneous — event-based programming, implicit invocation; F.3.2 [*Logics and Meanings of Programs*] Semantics of Programming Languages — denotational semantics, operational semantics; F.3.3 [*Logics and Meanings of Programs*] Studies of Program Constructs — control primitives, object-oriented constructs, type structure. F.4.1 [*Mathematical Logic and Formal Languages*] Mathematical Logic — Lambda calculus and related systems;

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Contents

List of Figures

List of Tables

Chapter 1

Abstract

Despite the rising popularity and usefulness of events, or implicit invocation, in software design, the availability of general-purpose event mechanisms are rare. Further, most event mechanisms available for software design are implemented as libraries or sets of macros that are constrained by the language in which they are used; making such mechanisms inconvenient to use as well as error-prone. Event mechanisms that are a part of a programming language can do away with such constraints; thus making events easier to use. However, there are few languages that offer built-in events and even fewer languages have a built-in general-purpose event mechanism. In order to promote the study of implicit invocation in programming languages, this thesis presents a formal programming language foundation for events.

This thesis expands the, object-based, **imp**ς-calculus to create a calculus for objects and events, the $\varrho\varpi\varsigma$-calculus. The $\varrho\varpi\varsigma$-calculus has a formal syntax, semantics, and a sound type system that is useful for defining practical programming languages that include built-in events. This, along with the ability of the calculus to easily simulate many different event mechanisms make it a good start toward a formal understanding of implicit invocation.

Chapter 2

INTRODUCTION

The notion of "handling events" is pervasive in software development. An event is a change of state in a system, e.g., a hardware interrupt, an expired timer, the completion of a thread, etc. Examples of some event handling mechanisms in use today include interrupt handlers, callbacks, "windows messages" or more formal mechanisms such as Java's EventListener class, COM+ Event Services, or CORBA events. Mechanisms such as these are becoming the standard way to handle unpredictable changes in system environments, e.g., a mouse movement, and the standard way to handle software integration, e.g., in C#. Furthermore, Gamma, et. al. [GHJV95], have coined a design pattern based on this idea of event handling called the Observer pattern, where models are observed by controllers that take action based on a change in the model. Aspect-oriented programming [KLM+97] also relies on something similar to an event mechanism to achieve what has been dubbed "cross-cutting". Thus, it is becoming more important for software developers to have tools that make events easier to use, understand, and maintain.

The typical event system in today's typical programming environments is largely ad hoc, add-on implementations of event systems that are tailored for a specific purpose, C#'s system being the only known exception to date. While the add-on approach provides some benefit to the developer, it is often lacking flexibility (not as general purpose as desirable) and cumbersome to use (poorly designed interface or syntax). While some well-done research exists on some of the software engineering aspects and consequences of using events, there is little research to date on the programming language or abstraction fundamentals that apply to events. *On Objects and Events* Eugster, et. al. [EGD01] is one example that makes a good start toward such analysis. However, the research in Eugster, et. al., focuses on providing a taxonomy for syntax, semantics, and typing choices for expressing events, similar to Garlan and Notkin, [GN91] and Garlan and Scott, [GS93], in an object-oriented context. In contrast, the research presented here provides a theoretical foundation for events in order to encourage the sound development of such syntax, semantics, and typing for events.

2.1 Background

Despite the fact that use of event handling mechanisms are becoming more popular, there have been only a few attempts to formalize events and event mechanisms. Surveying the current status of research in this area, one finds that it is lacking for how event mechanisms fundamentally relate to programming languages. Instead, the research focuses on extant implementations of event mechanisms, developing new, general-purpose event mechanisms, and formal reasoning about events and event mechanisms.

Dingel, et. al. [DGJN98a, DGJN98b] develop frameworks for formally reasoning about the results of event announcement. As such, they define a syntax and semantics to use for defining their formal reasoning framework. However, targeting the syntax and semantics for this specific purpose, constrains the syntax and semantics from being general enough to nicely model other event mechanisms, especially models with dynamic features such as dynamic event handler registration, etc. Further, their syntax and semantics was not created to aid inclusion of or provide a basis for adding event support to programming languages.

Barrett, et. al. [BCTW96], and Garlan and Notkin [GN91] define frameworks that generalize event mechanisms but do not attempt to define a formal syntax and semantics for a language that includes events. Instead, these works are focused on the taxonomy and categorization of extant event mechanisms for the purpose of developing a generalized event mechanism framework.

Garlan and Scott [GS93], and Notkin, et. al. [NGGS93], develop implementations of general-purpose event mechanisms using different programming languages and discuss the guidelines and various tradeoffs of developing

such implementations. While the guidelines are useful and the implementations instructive, mechanisms that are not native to a language tend to be much more inconvenient and error-prone to use. It is the same argument that is used for adding native support for object-oriented concepts to programming languages.

Eugster, et. al. [EGD01] suggest some native language constructs in their work and explore the ramifications of including a native event mechanism in a programming language. Also, like Garlan and Scott, and Notkin, et. al., they develop a taxonomy of syntax and semantic choices for event mechanisms. However, they all stop short of providing a formal semantic basis for an event mechanism.

As demonstrated to some extent by [EGD01], an approach that is similar to that of Garlan and Scott, and Notkin, et. al., but with an end-product that is easier and less error-prone to use, can be accomplished by defining a programming language that has built-in, or rather, native support for events. First, software developers would both avoid the inconveniences of having to deal with potentially many different, incompatible event implementations. Second, event features could easily be provided that would otherwise be difficult to implement and use, e.g., implicit event announcement, automatic declaration of events, etc. Finally, as with any language construct, the semantics for various event operations would likely be clearer. At the very least, the event semantics would be much more static than an add-on implementation; thus, providing software developers a more solid foundation for the software they write.

2.1.1 *Implicit Event Announcement*

Some of the recent popularity of events comes from the fact that they can help ease software integration [GN91]. This is typically done by explicitly declaring events for potentially interesting changes that occur within a program. However, it is often difficult to predict how software will be used for integration; thus making it difficult to define the events for every need that might arise in the process of integration. Implicit event announcement, event announcement that occurs without explicit announce statements, can be used to overcome this difficulty by announcing events for each change or transition in state, such as variable assignment, method invocation, etc. This form of announcement will be an integral part of the work presented here.

2.2 Goals and Approach

The goal of this work is to create a calculus, the $\varrho\varpi\varsigma$-calculus, that can be used as a formal semantic foundation for programming languages that have native support for events, making events more convenient to use, less error-prone, and provide useful features that may otherwise be difficult to provide. To accomplish this, the calculus must include an event mechanism general enough to simulate other kinds of event mechanisms in a reasonable manner.

To come up with such a design, the $\varrho\varpi\varsigma$-calculus syntax and semantics design was guided by the implementation issues outlined in Garlan and Scott [GS93] and Notkin, et. al. [NGGS93]: Event definition, event parameters, event registration, event announcement, delivery policy, and concurrency. For each of these issues, we have identified the most general approach and, based on that, determined how the calculus should be designed to ensure that the most general approach was handled directly or could easily be simulated by the calculus (except that concurrency was avoided to simplify our work). The details for these issues and the design of the calculus are given in Chapter 3.

We show that the $\varrho\varpi\varsigma$-calculus is useful as a formal foundation for programming languages with built-in events through its use and comparison to existing event mechanisms. We also use the calculus to specify the semantics for a new language, Eventua, via translations from Eventua declarations and expressions to $\varrho\varpi\varsigma$-calculus expressions. This demonstrates that the calculus can be used as a foundation for a language that could serve as the core of a practical language. Following that, a comparison of the $\varrho\varpi\varsigma$-calculus to the event-based integration framework developed by Barrett, et. al. [BCTW96] shows that the calculus can reasonably model many different event mechanism implementations. We also discuss the possibility of using the $\varrho\varpi\varsigma$-calculus and Eventua as a guide for building events into existing languages.

The definition of the $\varrho\varpi\varsigma$-calculus is a conservative extension of Abadi and Cardelli's **imp**ς-calculus [AC96], an imperative, object-oriented calculus. That is, the definition of the $\varrho\varpi\varsigma$-calculus does not change the original syntax and semantics of the **imp**ς-calculus, only extends it. Furthermore, the formal treatment of the $\varrho\varpi\varsigma$-calculus will closely follow Abadi and Cardelli's formal treatment of the **imp**ς-calculus (and other ς-calculus variants) to provide consistency between the related works. The $\varrho\varpi\varsigma$-calculus is defined in three parts: syntax,

semantics, and type-system. Then, based on those definitions, we prove that the type system for the $\varrho\varpi\varsigma$-calculus is sound.

Eventua is designed as the core for a practical language with features that one might expect in a language that supports events, such as parameter passing on event announcement, explicit event declaration and announcement, etc. The semantics for Eventua are given via translations into the $\varrho\varpi\varsigma$-calculus. A type system is also proposed for Eventua. Using Eventua, we give an example that demonstrates that use of implicit event announcement can make software integration and evolution easier to accomplish than with even explicit event announcement.

2.3 Overview

The rest of this work is presented as follows. Chapter 3 discusses the background for events and the problems with events that this paper attempts to solve. Chapter 4 provides the formal definition of the $\varrho\varpi\varsigma$-calculus. Chapter 5 gives the syntax and type system of Eventua and a translation from Eventua to the $\varrho\varpi\varsigma$-calculus. Chapter 6 gives the analysis of how the $\varrho\varpi\varsigma$-calculus and Eventua contribute to the solution of events in programming languages. Chapter 7 outlines the related work in events and implicit invocation. And chapter 8 provides conclusions of the research and summarizes areas of the research that have been left for future work.

Chapter 3

BACKGROUND AND MOTIVATION FOR THE PROBLEM

The motivation for using events is given by comparing it to its counterpart, procedural abstraction. In the same way that programming languages benefit from natively supporting procedural abstraction, programming languages can also benefit from direct support for events. We will contrast direct support for events against event mechanisms that are implemented within the target programming language, as a library, for example.

In order to discuss the benefits of having such a tool as a calculus that is intended to characterize an entire style of programming, it is necessary to define a basis that encompasses the most useful event mechanisms that currently exist as well as new event mechanisms that may come as a result of this and other works. The basis presented here is defended as a reasonably general solution by comparing it against features that are not directly represented in that basis and by arguing that our basis can simulate other potentially desirable features of an event mechanism in a reasonably simple fashion. This basis and the discussion that follows guides the direction of both the $\varrho\varpi\varsigma$-calculus and Eventua.

3.1 Events: Motivation and Problems

To make programs manageable, programmers break their programs into procedures. In higher-level programming languages, such as Pascal or C++, the use of procedural abstraction appears as functions and methods. The functions and methods are given names which are used in other parts of the program to call or execute them. That is, functions and methods are explicitly invoked to perform some task.

Aside from its typical use of breaking down a program into manageable and reusable parts, procedural abstraction is often used for defining procedures that handle some kind of change in the state of the program or system, e.g., a system interrupt, a mouse click, etc, where the procedure needs to be executed without an explicit invocation. *Explicit invocation* refers to a function or method invocation where the actual label, name, or message is used by the client to cause the invocation. For example, a function call, `function()`, or method invocation, `object.message()`.

Explicit invocation does not handle the case where a component[1] needs to announce that something happens, i.e., a particular change in the state of the component has occurred, and allow unknown clients to handle such a change in a way that suits them. The problem is that procedural abstraction requires names to be statically bound to procedures. Since the components do not know about their clients in advance, they do not know the name of the procedure the client will use to handle their announcements. Thus, they will not be able to directly call the client's procedures to allow them to handle such interrupts and announcements.

The usual way to handle this kind of situation is to allow the client to register a callback procedure that the announcer will call when the change occurs. Usually, to register a callback, a client will pass the component a reference[2] to the procedure, e.g., the procedure's entry point address. Then, the component saves the reference so that it can treat it like any other procedure it would call, except that it must make the call indirectly through the saved reference. However, as discussed above, callbacks are traditionally error-prone for many of the same reasons that make references and pointers error-prone in general.

Take, for example, a situation that can often be encountered in C where we are passing a function pointer to such a component as a callback. While the function pointer will have a type that specifies number and type of its

[1]The word, *component*, is used loosely here to mean any unit of code, e.g., a program, module, COM component, API, etc.
[2]This is the common way to register a callback

arguments, return value, etc., C's type checking is often times defeated by programmers allowing the potential for registering a wrongly typed callback function. While no one would purposefully do this, function pointers, like many other kinds of pointers, are sometimes stored generically (e.g., with `void*`), removing its type information. Then the unwary programmer could cast the pointer to the type required for the callback registration, registering a wrongly typed function.

Perhaps a better example of this error-prone property is the fact that use of callback functions tends to hide or obfuscate the semantics of the execution of the callback. While it is true that event mechanisms in general tend to obfuscate the semantics of a program in general [DGJN98a][DGJN98b], callback mechanisms exacerbate the problem by also hiding the semantic behind the callback function activation. For example, it would be possible to define a function that would in some cases place the callback's context on different thread, but in other cases would execute the callback synchronously.

One additional drawback of the callback mechanism is that it does not easily handle the multiple clients case. Suppose that, for example, instead of the usual case where only one client is interested in a "mouse moved" event, there are two clients. While it is not impossible to handle such a situation, the implementation would be, in general, be much more involved, e.g., maintaining a list of callbacks versus a simple variable. Furthermore, any attempt at conveying the semantics of such a mechanism would be difficult at best, handling issues such as mentioned above around threading, as well as the order of callback execution and the manner of announcement, e.g., only one callback per event or all of them for every event. This may not be too burdensome in a system with very limited use of such a mechanism; however, imagine a system with just ten different but similar callback mechanisms. It seems that use of such a system would quickly become more difficult as the number of callback mechanisms increase.

Aside from the problem of dealing with interaction between parts of a program unknown to a particular component, it is often useful to create general routines or objects that observe the state of a component and report their observations to any component that might want to do something with these observations. For example, imagine a component that simulates a physical system such as an astronomical system. Also, suppose that when we use the component, we usually want the data pertaining to the behavior of the whole system, but on other occasions we would rather have the data for each individual part of the system. That is, we would want one implementation of the simulation that will give the user one of two different sets of data.

There are several different approaches the user could take to solve this kind of problem. We could implement the simulator so that it always generates both sets of data. Or, we could implement the simulator so that it follows different branches, dispersed throughout, that would give the desired output. The first approach would generate perhaps more than we would need, consuming more resources (time and space) than necessary. The second approach would be potentially difficult to maintain.

A third approach would be to have two different simulators with a common base implementation and special code that would generate the desired output of the simulation for a given run. This would be, potentially, easier to maintain and would give only the results desired, however, it doesn't handle the case where both sets of data would be required from the same run. Thus, in the general case where there would be many different kinds of results (not just two), there would have to be a different implementation for every kind of result and every desirable combination of results. Also, this approach is not very modular, it would probably still require redundant implementation to handle each case, for example, redundant code for interacting with the base implementation.

Also, the problem with these and similar approaches is that they do not evolve well. That is, they do not nicely handle the case where we want to do new things with the simulation, such as, adding to or changing the simulation itself. It would also be potentially difficult to add extra interaction between the simulation and the respective data collection implementations.

With this in mind, another approach would be to define the simulator without regard to a specific type of output, similar to the idea of a common base implementation, except that when something "interesting" happens in the process of the simulation, it announces a related event. Then, depending on the data we want from the simulation, we define different handlers for these events. In this way, when we want a different set of data from our simulation, we define different handlers, leaving the rest of the program untouched. Further, if our announcement mechanism is general enough, we may be able to execute our program and get more than one set of data out by using more than one set of handlers at the same time.

So, with this approach, our simulation would observe itself and report its observations to any component that is interested; that is, the simulator *announces events* that other components are welcome to handle, when

desired. Not only would this make it easier to produce different sets of data for the same simulation, it would also make it much easier to mix those sets of data if desired. For example, recalling the original scenario of two desired data sets, we could get all the data that involves the entire system interspersed with just some of the data from each of the individual parts of that system. Further, the simulator could easily, for example, take on more functionality and simply announce more events without requiring changes to the existing data collection code. Of course, such things could all be accomplished without the use of events, but, in general, it would require significant time and energy above and beyond what would be necessary if events were used.

As demonstrated in the simulator example, events can be useful for modular software evolution. Sullivan and Notkin, [SN92], further demonstrate that use of events is beneficial for such things as software evolution and composition. Popular component models, such as COM+, CORBA, JavaBeans, etc., give further credence to this claim by providing event mechanisms for the purpose of integrating components. Aside from component models, event, or event-like, mechanisms also are widely used to handle user interaction (e.g., Windows), tool integration (e.g., Field [Rei90]), and various forms of system interrupt handling, (e.g., OS traps). Such systems have requirements similar to that of our simulator example: different responses (or implementations) for the same or similar processes, structures, environments, given differing circumstances; the ability to add to processes, structures, or environments without effecting current implementations; and the ability to handle new responses instead of or even along with existing ones.

The problem is that programming languages, in general, do not provide direct support for announcing events and defining handlers for events. Despite the benefits that events provide, such as increased reuse, ease of software evolution, etc., there are few general-purpose event mechanisms used in practice [GS93]. However, with ideas such as component-based software development, for example, events and event mechanisms are becoming more commonplace as indicated by the inclusion of events in C#.

Direct programming language support is not necessary to use events, as shown by Garlan and Scott, [GS93], and Notkin, et. al., [NGGS93], but it would, in general, make them less tedious to use and perhaps make the programs that use them easier to read, understand, and maintain. Further, direct support may provide other useful capabilities that would be difficult to provide otherwise.

To see this in practice, consider an analogous example: the difference between writing an object-oriented program using C versus C++. While it is possible to write C programs in an object-oriented style, it is arguably easier to write object-oriented programs using C++. This is because C++ provides some direct support for object-oriented ideas such as inheritance, polymorphism, and private object members, that would be difficult, or at the very least tedious, to achieve in C. The same is true for programming using an event style. It is possible to define and use events in most programming languages by using callbacks, for example, iterating through lists of callbacks, etc. However, it is much more convenient to use events when they are directly supported in a programming language.

Typically, event mechanisms built on top of existing languages require superfluous information, e.g., extra identifiers, declarations, type information, etc., in order to make the mechanism work within the confines of the implementation language and operating environment. Also, applications using such implementations are typically difficult to debug because special adaptations are often made that can "confuse" the compilation and debugging environments, such as, pre-compilation translations, multiple threads, special libraries, etc.

A good case study of the unnecessary complexity caused by non-native event implementations is a comparison between Microsoft Visual Basic, Microsoft Visual C++, and Microsoft .NET C#. Microsoft Visual Basic and Microsoft .NET C# both provide native support for handling certain kinds of events. Consequently, handling events from, for example, COM components, is relatively simple. Visual Basic simply requires a selection from a drop-down list of events or a function or subroutine with a special name designating the event it will handle. If selection from a drop-down list is used, the editor takes the user to a stubbed-in procedure that is called when the event is announced. Figure 3.1 shows one such stub. C# uses an EventHandler class and event declarations to allow event handling. Figure 3.2 shows an example of this syntax.

Microsoft Visual C++, on the other hand, requires programmer-defined event sinks. The event sink contains the definition of the mapping between the events and the handler methods which includes information that could be handled internally for native events such as the event's identifier and the types of the parameters an event will pass. This is illustrated in Figure 3.3 with a series of classes and their partial implementations.

The class, `Backup`, gives us an environment similar to the environment implicit within Visual Basic. The `BackupSink` class is the class that handles the events from our `BackupComponent` COM component, requiring the class to inherit from the Microsoft Foundation Class, `CCmdTarget`. The `DECLARE_DISPATCH_MAP()` macro provides

```
Private Sub BackupComponent_BeforeBackingUpFile(
                     ByVal sFilename As String, ByVal lSize As Long,
                     ByVal xAttributes As BackupLibCtl.bkpFileAttributes,
                     ByVal dtLastModified As Date, ByVal dtLastAccessed As Date,
                     ByVal dtCreated As Date, ByVal lDiskNumber As Long)

'Do something useful

End Sub

Sub DoExample()

'Do something that could trigger an event

End Sub
```

Figure 3.1: Microsoft Visual Basic Event Handling Example

```
class BackupComponent {
    public event EventHandler BeforeBackingUpFile;
}

class App {

    void OnBeforeBackingUpFile( String aFilename, Int32 lSize,
                                bkpFileAttributes xAttributes,
                                Date dtLastModified,
                                Date dtLastAccessed,
                                Date dtCreated, Int32 lDiskNumber ) {
        //Do something useful.
    }

    void DoExample() {
        //The component that will generate the event.
        BackupComponent bkup = new BackupComponent();

        //Register for the event.
        bkup.BeforeBackingUpFile +=
            new EventHandler(this.OnBeforeBackingUpFile);

        //Do something that could trigger an event

    }
}
```

Figure 3.2: Microsoft .NET C# Event Handling Example

an event sink declaration in the `BackupSink` class. The `BEGIN_DISPATCH_MAP()` block provides all the necessary information to set up the message path from the Windows message queue to the appropriate event handlers (the standard Windows message queue is used here to actually deliver the events). Finally, the `AfxConnectionAdvise` and `AfxConnectionUnadvise` functions register and unregister the event sink for events from the component. It should be clear from the examples that the built-in support in Visual Basic makes for a shorter, cleaner program than that in Visual C++.

Notkin, et. al. [NGGS93] give an implementation of a general-purpose event mechanism using macros and classes in C++. While the implementation of the event mechanism is relatively elegant, an extra macro, `STATIC`, is required for the definition of event handlers. Specifically, event handlers are defined as member methods of a class. Then, the `STATIC` macro must be used to "declare" the method as a handler by making a static member of the class that invokes the related handler as in Figure 3.4. In order for the `STATIC` macro to work correctly five arguments must be passed: the name of the class that the handler resides in, the return value of the handler, the name of the handler method, the list of formal parameters for the generated static method, and the list of actual parameters for the invocation of the real handler. While it is not necessarily difficult to understand how to provide this extra information, it is both inconvenient, as the information does not directly pertain to using events, and error-prone, since most of the declaration information for the handler method must be repeated for the `STATIC` macro. If C++ had a native event mechanism, we could avoid passing around this extra information. In Figure 3.4, we see the event instance `inf.TestEvent` getting a registered handler, `Listener::OnTestEvent`. Then, `inf.TestEvent` is announced, which will cause the code (not listed here) in `OnTestEvent` to be executed.

In the same way that native language support for object-oriented concepts has advantages over other ad-hoc or added object support, providing native support for events has an advantage over event mechanisms widely in use today: callbacks, message queues, and other ad-hoc or added event mechanism implementations. Native event constructs allow for a clearer semantics than other approaches by virtue of being a part of the language. The possibility of developer error is decreased due to less cumbersome notation and higher capability of enforcement of requirements, such as typing. Software becomes more portable due to having a standardized event mechanism. Development time is reduced by avoiding one reimplementation after another of various event mechanisms that serve a specific purpose. And finally, a native event mechanism has a better chance of being more flexible than other implementations since the implementation of the mechanism is not restricted by limitations and nuances of a target language. C#'s native support for events provides some support for this position.

3.2 Minimal Definition of Event Mechanisms

Every event mechanism has at least three distinct operations: publication, handler registration, and announcement. *Publication* defines the availability of an event, i.e., when an event is published, it can be announced. It is often the case that events are published upon declaration, e.g., provisions for callbacks in an API, or assumed to be published at execution time, such as windows messaging. *Handler registration* connects handlers to events, i.e., when a handler h is registered for an event E, the program segment defined for h is executed every time E is announced. *Handler unregistration* removes a binding between an event and one of its handlers. The *announcement* operation is a notification of an event occurring, e.g., announcing E lets interested handlers know that a change in the state of the program occurred which is represented by E. These three roles compliment the three roles of procedural abstraction: declaration, implementation, and invocation, where the difference is that procedural abstraction uses the declaration, i.e., a procedure name, to "find" an implementation upon invocation and event abstraction uses registration information to "find" implementations upon announcement.

3.3 Variations and Generalization of Event Mechanisms

In this research, we define a calculus, the $\varrho\varpi\varsigma$-calculus, that is intended to be a foundation for programming languages that directly support events. Since our calculus is intended to be foundational, our goal is to define a general abstraction of event mechanisms to accommodate as many event mechanism implementations as possible. In order to show that the $\varrho\varpi\varsigma$-calculus is useful, we develop a core language, Eventua, in Chapter 5. For Eventua, our goal will be to define an event mechanism that is easy to use and practical, showing that the $\varrho\varpi\varsigma$-calculus can be useful as a foundation for a practical event mechanism implementation.

Garlan and Scott, [GS93], and Notkin, et. al., [NGGS93] have listed several design issues related to implementing event mechanisms using various programming languages. While these were developed for a purpose

```
class Backup
{
  public:
    void DoExample();
};

class BackupSink : public CCmdTarget
{
  DECLARE_DISPATCH_MAP()
  DECLARE_INTERFACE_MAP()  //for GetIDispatch to work correctly
  public:
    void BeforeBackingUpFile( BSTR sFilename, long lSize,
                              enum bkpFileAttributes xAttributes,
                              DATE dtLastModified, DATE dtLastAccessed,
                              DATE dtCreated, long lDiskNumber );
};

BEGIN_DISPATCH_MAP(BackupSink, CCmdTarget)
  DISP_FUNCTION_ID(BackupSink, "BeforeBackingUpFile",DISPID_BEFOREBACKINGUPFILE,
                   BeforeBackingUpFile, VT_EMPTY,
                   VTS_WBSTR VTS_I4 VTS_I4 VTS_DATE VTS_DATE VTS_DATE VTS_I4)
END_DISPATCH_MAP()

BEGIN_INTERFACE_MAP(BackupSink, CCmdTarget)
  INTERFACE_PART(BackupSink, DIID__IBackupEvents, Dispatch)
END_INTERFACE_MAP()

void BackupSink::BeforeBackingUpFile( BSTR sFilename, long lSize,
                                      enum bkpFileAttributes xAttributes,
                                      DATE dtLastModified, DATE dtLastAccessed,
                                      DATE dtCreated, long lDiskNumber )
{  /* do something useful */ }

void Backup::DoExample()
{
   //the component that will genereate the event
   IBackupComponentPtr bkup(CLSID_BACKUPCOMPONENT);

   BackupSink sink; //an instance of the event sink
   DWORD cookie; //the registration ID holder

   LPUNKONWN iSink = sink->GetIDispatch(FALSE); //get interface for advise

   AfxConnectionAdvise(bkup, DIID__IBackupEvents, iSink, FALSE, &cookie);

   //do something that could trigger an event

   AfxConnectionUnadvise(bkup, DIID__IBackupEvents, iSink, FALSE, cookie);
}
```

Figure 3.3: Microsoft Visual C++ Event Handling Example

```
class Object
{
};

class Informer
{
  public:
    VoidEventObject TestEvent;
};

class Listener
{
  private:
    void* OnTestEvent(Object& o);
    STATIC(Informer, void*, OnTestEvent, (void* target, Object& o), (o));
};

void main()
{
  Informer inf;
  inf.TestEvent.REGISTER(Informer, OnTestEvent);
  Object o;
  inf.TestEvent.Announce(o);
}
```

Figure 3.4: Event Handling Example for the [NGGS93] Mechanism

different from ours, most of the issues generally apply to the design of any event mechanism and thus are relevant for discussion here. There are six different categories of event mechanism design issues listed by that research: event definition, event parameters, event registration, event announcement, delivery policy, and concurrency. *Event definition* pertains to the means given by the event mechanism for defining events. *Event parameters* deals with the level of support provided for passing information with event announcement. *Event registration* differentiates between static (compile-time) and dynamic (run-time) handler registration. It also deals with the way parameters are passed between the event announcement and its respective handlers. *Event announcement* covers the different ways event announcement could be allowed to occur for a particular mechanism. *Delivery policy* addresses the various ways that a mechanism can go about notifying handlers of an event announcement. Finally, *concurrency* pertains to the way in which an event mechanism uses (or does not use) concurrency to deal with event announcement.

Since our goal for this research will be to create a generalized event mechanism, we are interested in the most general approach for each category. To determine the most general approach, we consider the expressiveness of each approach with respect to the other approaches. That is, approaches that can directly simulate the other approaches are considered more general than those that cannot. Using this analysis, we will discuss the design rationale for our event mechanism.

It is often the case that the issues surrounding the choices for each category involve static versus dynamic binding, for example, static handler registration versus dynamic handler registration. For every category with such a choice, dynamic binding is more general than static binding, since it is always easy to simulate static binding by dynamically binding everything at the beginning of a program. However, simulating dynamic binding when static binding is the only available option is difficult since there would be no direct way to express the dynamic bindings. Since dynamic binding is more general and since many existing event mechanisms use various kinds of dynamic binding, it is a better choice for a generalized mechanism.

3.3.1 *Event Definition*

For event definition, the primary concern is the amount of latitude given to the user to define events. Essentially the choices are a fixed set of events, where the user cannot define new events; explicit definition of events, where all the events used must be predefined; and automatic definition of events, where no declaration of an event is necessary for an event to be used, e.g., announcing an event e or registering for an event e where e is not explicitly defined anywhere as an event.

The most general approach is to allow automatic definition of events since it could be used to simulate other kinds of event definition. That is, declarations could be allowed for events that would use the automatic events in a specific way in order to simulate the event declaration; however, automatic definition of events would be difficult to simulate by a system that only allows events that are statically declared. Specifically, to simulate such a system, general events would need to be declared (or predefined by the environment) that would be fired for any possible automatically defined event. Then, the handlers for the automatically defined events would need to take a parameter or access some state variable that specifies the event that was fired and dispatch the announcement to the appropriate handlers.

Some existing event mechanisms allow automatic event definition. Examples, of such systems include system in Windows, and tool integration frameworks such as Field [Rei90] and, SoftBench [Ger89] do this.

Allowing for automatic events is necessary when using events to integrate components that do not support the desired event mechanism. In such cases, the output from these components can be used as events within the integration environment. Allowing for automatic events is also useful for making implicit announcement easier to use. In this case, if automatic events were not allowed, events would have to be explicitly declared and then explicitly bound to an implicit announcer. On the other hand, if automatic events were allowed, the implicit announcers could announce their own events without further ado.

Even though automatic event declaration is the most general approach, the $\varrho\varpi\varsigma$-calculus does not directly support it. The $\varrho\varpi\varsigma$-calculus requires every event that is announced to be declared. However, the $\varrho\varpi\varsigma$-calculus has no way to directly denote an event, thus there are no explicit declarations of events. Instead, all the methods of every object automatically create events. So, in a sense, since every variable can be considered an event, the $\varrho\varpi\varsigma$-calculus has something that parallels automatic event declaration.

It is straightforward to simulate a common use of automatic event declaration: adapting programs or modules that do not explicitly announce events for event-based integration. This can be simulated by defining an object as the output space for the module which announces an event each time there is new output. Then, handlers would be registered for the "new output" event and could filter on the contents of the actual output. For theoretical purposes, the $\varrho\varpi\varsigma$-calculus style of automatic event declaration makes the calculus easier to use because it requires less syntax (no event declarations or announcements).

Eventua allows both automatic event declaration on fields and methods, similar to the $\varrho\varpi\varsigma$-calculus, and explicit declaration, since it is often convenient to be able to define events for a specific purpose.

Since automatic event declaration can create more events than necessary, it may be useful to narrow down the number of events that are actually active to provide some control over the set of events that can be announced. Both the $\varrho\varpi\varsigma$-calculus and Eventua accomplish this with event publication. So even though events are automatically declared with every new field and method, they will not be announced unless they are also published.

3.3.2 *Event Parameters*

The event parameters category describes a range of parameter passing flexibility from no parameters; to an event-wide, fixed set of parameters, i.e., all events pass the same parameters; and up to arbitrary parameters for each announcement, i.e., the type and number of parameters that can be passed for any given event are not fixed. If the goal of the event mechanism is to provide a generalized abstraction, it would be best to allow for the most flexible form of parameter passing.

For the $\varrho\varpi\varsigma$-calculus we want a generalized abstraction, but we also want to avoid complexity. Thus, we leave out direct support for parameter passing, showing how it can be simulated in chapters 5 and 6. Leaving out parameter passing also affords languages that are built on the calculus the flexibility to define their own method of parameter passing.

Eventua will employ a common parameter passing style for its explicit events, allowing each one to have their own fixed number of parameters with fixed types. Eventua will also allow the parameters of method invocations

to be passed when the invocation is announced as an event.

3.3.3 *Event Registration*

For event registration, dynamic registration is more general than static registration since static registration can be simulated using dynamic registration by dynamically registering every handler for their respective events at the start of a program, disallowing any unregistrations. Further, dynamic registration cannot be easily simulated with static registration since such a simulation would require events to be interpreted and delivered to appropriate handlers manually, i.e., without direct support from the event mechanism. Also, while registering a handler, it is possible to determine how parameters will be passed between an announced event and the handler. Parameter bindings can be defined to give the handler only the parameters it desires, a push model. Or, parameter binding could also be defined such that the handler will get the parameters it needs when it is triggered, a pull model. It would also be possible, of course, to have a combination of both push and pull.

The $\varrho\varpi\varsigma$-calculus provides dynamic registration since it is more general. Further, its event mechanism has no support for parameters, thus it does not support any form of parameter binding; however, simulating parameter binding is straightforward as it is a variation on the simulation of parameter passing. Support for dynamic registration is justified since many event mechanisms support this feature, e.g., Windows message passing and the event mechanisms in our C++ examples, Figure 3.3 and Figure 3.4. Further, dynamic registration solves various problems that cannot be easily solved with static registration. For example, handlers may become invalid, e.g., an object is dynamically destroyed that has an event handler as a member or a web-browser that is handling events for a particular page is commanded to load a different page. Also, a program may need to handle events differently at different times, e.g., events from an object that processes sensor input, say weather radar input, that are announced for the "arrival" of various kinds of data. The program would need to handle such events differently for different modes of operation, such as analyzing wind speed versus analyzing precipitation.

Eventua also supports dynamic registration. Since Eventua also supports parameter passing, it supports a form of push parameter binding for registration based on the number of parameters being passed and expected. If there are more parameters passed than expected, only the parameters that are expected are given to the handler. If there are fewer passed than expected, the extra parameters for the handler will be set to undefined values.

3.3.4 *Event Announcement*

In general, the relevant issue surrounding event announcement is whether to have explicit or implicit announcement. That is, announcement that is accomplished via an `announce` instruction of some sort as opposed to announcement that occurs as a side-effect of a change in the program state, such as an assignment or a procedure call.

It is possible to simulate explicit announcement with implicit announcement by forcing a particular state change to occur that would cause an implicit announcement of a desired event. On the other hand, it would not be easy to simulate implicit announcement with explicit announcement since there would be no way to announce an event due to a particular change in state unless every statement that causes a state change is immediately followed by the appropriate explicit event announcement. Thus, implicit announcement is more general than explicit announcement. It is valid to consider implicit announcement for a general event mechanism since event mechanisms exist that use implicit announcement for database updates [DHL90] and, according to Garlan and Scott [GS93], changes that occur in programming environments [HGN91].

As discussed later in the next section as well as in Chapter 5, implicit announcement is useful when a component, for example, does not explicitly announce an event that a client needs to effectively integrate the component. In this case, implicit announcement would allow the client (via some means, depending on the implementation of the implicit announcement support) to still get the needed event from the component. Both the $\varrho\varpi\varsigma$-calculus and Eventua support implicit announcement since it is both useful and more general than explicit announcement. For convenience, Eventua also supports explicit announcement of explicitly defined events.

3.3.5 *Delivery Policy*

For delivery policy, the issue of whether events are directly delivered or are filtered in some way prior to delivery is of primary concern. Examples of event filters are, conditional announcement, such as "announce $\hat{e}$ when e is announced and the condition *cond* is true"; event translation, such as "announce $\hat{e}$ when e is announced"; event sequence detection, such as "announce $\hat{e}$ when e_1 is announced followed by the announcement of e_2"; etc.

It would be possible to support filtering by providing syntax within the language to specify any kind of possible event filtering. Then the implementation of the language would keep track of the entire history of event announcements and announce filtered events accordingly; however, this seems to be too complex for a calculus. Fortunately, we can simulate various forms of event filtering with direct delivery by inserting intermediate event handlers and announcers in between the "real" event and the "real" handler without too much difficulty.

Since it is possible to easily simulate, at least, some forms of event filtering with direct delivery, the $\varrho\varpi\varsigma$-calculus does not provide direct support for it, with the exception that delivery can be only be made only on the success of a test that is defined at the time of registration. This conditional announcement filter has been included to provide some control over when handlers are executed. This is arguably beneficial since, as previously mentioned, our mechanism uses implicit announcement, which may cause events to be announced more often than it is desirable to handle the announcements, e.g., we only want our handler for e to be executed when $x > 5$ rather than each time e is announced.

Eventua, other than a conditional announcement filter, also does not have any support for event filtering. Since the syntax for a general filtering construct could be complex, it was left out of Eventua to keep the language simple.

3.3.6 *Concurrency*

The concurrency category focuses on whether handlers are executed concurrently or not. Here, the most general approach would be to give the programmer a choice between concurrent and serial handling and at the time of registration. In the case where concurrent or serial handling is offered exclusively, one is not more general than the other. But, it is often the case that concurrent handling of events is more desirable than serial handling because concurrent handling would eliminate issues involving order of handler execution, since all the handlers would be executed at the same time. However, we do not deal with concurrency in the $\varrho\varpi\varsigma$-calculus and Eventua, in order to avoid other problems that concurrency can cause, e.g., race conditions, uncontrolled access to shared parts of the store, etc.

3.3.7 *Summary of Properties of the $\varrho\varpi\varsigma$-calculus Event Mechanism*

In summary, the event mechanism in the $\varrho\varpi\varsigma$-calculus will be defined to allow automatically defined events. These events will be governed by event publication and announcement conditions, dynamic handler registration, implicit event announcement, and direct delivery of event announcement. The event mechanism will not include parameter passing and concurrency. As argued above and as we will see in chapter 6, the event mechanism for the $\varrho\varpi\varsigma$-calculus is general enough to be used as a basis for many of the existing event mechanisms as well as future ones.

3.3.8 *Summary of Properties of the Eventua Event Mechanism*

The Eventua event mechanism will allow for automatically defined events for fields, methods, and even method invocations. For user convenience, explicitly declared events are allowed. Parameters are allowed to be passed for explicitly declared events and method invocation events. The mechanism also supports dynamic handler registration, both implicit and explicit event announcement, and direct delivery of event announcement. Eventua, however, does not support concurrency.

The goal of Eventua was to make it a viable candidate as a core language that includes support for events that a programmer would either expect to find or find useful in a language. This is partly accomplished with the addition of declarations and a more human readable syntax, however, the addition of parameter passing and explicit event declaration and announcement, also makes Eventua more appealing than something more like the $\varrho\varpi\varsigma$-calculus mechanism for use as a core language.

3.4 Automatic Definition and Implicit Announcement: Rationale and Trade-offs

As previously mentioned, event mechanisms can make modular software evolution easier to achieve. This is typically accomplished by publishing events that could be useful for future integration and evolution. The problem is that developers creating non-trivial components usually do not have enough knowledge about which events other components will need for integration with that component. One solution to this problem would be to define events for everything that happens in a component that could potentially be useful. However, this would typically require a large amount of events, making the component difficult to maintain. Modifications to the component would often require significant changes to when events should be announced, introducing a high potential for errors.

Take, for example, a simple text editor. Suppose we announce events for every letter inserted into the document and for every line, paragraph, and page created. Now, suppose that we announce the "letter" events by detecting that a key is pressed on the keyboard. This will work fine until we decide to change our editor to allow the user paste text into the editor. To ensure that our letter insert event announcement was still being done at the proper times, we would need announce events for each character introduced by a paste as well. As this simple text editor evolves into a complex document editor, it will become more difficult to ensure that events are always announced at the proper times.

The underlying problem is not the skill of the developers, but the event mechanism in use. Specifically, the problem is that explicit announcement forces developers to explicitly define the events they plan to announce and then explicitly announce such events. An alternative approach would be to implicitly or explicitly announce an automatically defined event every time the state of the system changes. One way to accomplish this is to automatically announce different events every time the state of the component changes in a unique way. Usually, the state of a component will change when an assignment, that is, a change in the component's store occurs. To automatically announce such changes would require an event mechanism that has this capability built in, such as the mechanism we will define for the $\varrho\varpi\varsigma$-calculus.

Without an event mechanism that supports automatic event announcement, announcing events for every change in a component's state is not feasible. It would require either a translation of each program that would make every assignment announce an event, or one would have to define all state changes manually as events.

The translation approach could be used, but output of such a translation would be the source used to debug the application in question, which would either be difficult to use, or would require enhancements to existing debuggers or extra help for the debugger in the translation. Using a translation can also be problematic if variables are allowed to be aliased (especially implicitly aliased, e.g., via pointer arithmetic), since it would need to fire an event for both the original variable and its alias, to ensure that handlers for both are notified. In this case it may be better for the translation to be based on memory locations rather than on name. However, announcing events based solely on changes in certain memory locations would make it difficult for developers to understand how to handle such events. For example, a developer could register for events caused by changes in a dynamically allocated array. However, as the array grows, it is reallocated with a larger size, changing the memory location of the array. Thus, upon reallocation, changes to the array would no longer generate the events that the developer had registered to handle. In such cases where there is a disconnect between variables and memory locations, clients of components that announce such events would be at a loss for keeping up with what memory to keep track of and what particular memory locations mean for the component at any given time.

Alternatively, leaving the task of announcing assignments, i.e., every change in state, up to the developer would be even worse. The inconvenience to the developer, requiring them to follow every assignment by an event announcement, and the additional potential for error, announcing the wrong event, would make this solution untenable, or at the very least unacceptable for both the developers of components and the client developers.

Therefore, it is more practical and less error prone for developers if implicit event announcement was handled directly by the language itself. Most developers would not likely adopt implicit event announcement any other way than this, since it would likely be considered too cumbersome to use.

While built-in automatic declaration and implicit announcement of events is more desirable than the solutions listed above, these are not without problems of their own. The most obvious problem is the cost of announcing an event every time an assignment is done. Also, it could be overwhelming for developers to attempt to wade through the set of all possible events that could be announced, e.g., all the variables in the component, and figure out which ones they need to use (even in a small component), or even what an announcement of one of those events would mean. For example, imagine a developer that is trying to decide which variables to register

handlers for in a program that enlists even a hundred variables to complete its task. Further, the developer who is making changes to the program does not know what handlers the change will affect or how it will affect those handlers. One solution to help minimize this problem would be to control which variables are allowed to announce events. The $\varrho\varpi\varsigma$-calculus does this via event publication, allowing only published events to be announced. Event publication may also decrease the cost for having automatic announcement, since only published events would be the ones that would need to be announced.

Providing limits or conditions for when handlers will actually handle events can also help clients manage event notifications. In the $\varrho\varpi\varsigma$-calculus, we can specify the conditions under which handlers will be allowed to handle announced events. This is accomplished by allowing announcement conditions to be defined at the time of registration, allowing the handler to execute only if the conditions succeed. This can further reduce the cost of an event announcement by reducing the number of event handlers that will be allowed to execute for a given event. This will also afford clients the ability to apply extra context around an event announcement before deciding to actually handle the given event. One example of this would be for a component that implements a state machine. The component announces changes to a variable, x. A client is interested in knowing when x changes, but only when the component is in a particular state. Using conditions, the client can register for the "change in x" event and handle it only when the component is in the desired state.

The implementation change problem, where the developer doesn't know how events will be effected by implementation changes, demonstrates the biggest problem with automatic event announcement as presented thus far. This problem is directly related to one of the traditional problems in software engineering: coupling. Independent parts of one component are dependent on a specific implementation of another component. A common approach that avoids coupling with a traditional programming language construct, functional abstraction, is to define abstract data types (ADTs), i.e., interfaces, that hide the details of the underlying implementation. Done correctly, the underlying implementation can be completely changed without affecting any parts of the program depending on the interface.

In a similar way, it would be possible to build event interfaces that represent an abstract view of the ways that the state of the component changes. This would allow developers to change the implementation of a component without affecting the clients that rely on the events from the given interface. This, however, seems to bring us back to the earlier problem of requiring explicit event declaration and announcement. Developers forced to explicitly define events, which was already stated to be, in general, insufficient for clients.

The difference with the event interface approach over the general technique of simply suppling a set of events is twofold. First, in the same way that it is possible to have a complete interface, in the typical sense, it is also possible to specify a complete event interface. However, as stated, this would pose the problem of maintenance, bringing us to the second difference: use of implicit announcement internally to determine when to announce external events.

Developers could use implicit announcement of changes to the implementation's state to announce events on the interface. The example of the word processor illustrates this point. The "letter inserted" event in the word processor from our earlier discussion was explicitly announced when keys were pressed, and then later when text was pasted in. The onus of announcing those events was on the developer. Every time the developer added a new way to add letters to a document, they had to announce an event as well. For the sake of discussion, let us assume that there was a buffer that kept track of the letters in a document. In this case, with implicit event announcement, the developer could use the internal buffer change events to announce the external "letter inserted" events. So, with this approach, it doesn't matter how many different ways there are to add letters, because they all have to change the buffer, which causes buffer changed events to occur automatically. Thus, using event interfaces with implicit announcement within an implementation to announce such events can be an effective engineering practice for event based programming, and it is fundamentally supported in the $\varrho\varpi\varsigma$-calculus.

An alternative approach to keep clients informed of state changes would be to implicitly announce events when a function is called or a method is invoked. Doing this would allow interested clients to keep track of, for example, all the methods of an object that are invoked and, using that invocation history, reconstruct the abstract value of the Abstract Data Type (ADT) [GW98]. The advantage to this approach over other approaches is that the state of the ADT could be monitored despite the fact that the ADT provides no direct support for such monitoring. Of course, use of implicit announcement for a function call or method invocation assumes that there is a sufficiently complete [GH78] specification that says how the functions or methods change the state of its component; otherwise, the client implementations would have to attempt to model the changes that each

function or method makes via observation and trial-and-error. The Eventua language, chapter 5, will provide an announce-on-invocation mechanism.

3.5 Summary

To work toward the resolution of the absence of general-purpose event mechanisms, we will formally define the syntax, structural operational semantics and type system for the $\varrho\varpi\varsigma$-calculus which models events based on the discussion above. We claim that this formalism will be useful as a theoretical foundation for developing some programming languages that have direct support for events. To show that the claim is valid, we construct a core event language, Eventua, that is defined in terms of the $\varrho\varpi\varsigma$-calculus in Chapter 5. We further support this claim by measuring how well the $\varrho\varpi\varsigma$-calculus fits the generalized event framework developed by Barrett, et. al. [BCTW96] in Chapter 6.

Chapter 4

THE $\varrho\varpi\varsigma$-CALCULUS

This chapter details the formal definition of the $\varrho\varpi\varsigma$-calculus (pronounced *rho-pi-sigma*). The calculus is a conservative extension of the **imp**ς-calculus [AC96, Ch. 10, 11] that includes direct support for events. A *conservative extension* to a language is an upwards compatible extension, leaving the syntax and semantics of the original language intact. A syntax and semantics for the calculus is given followed by a type system that includes subsumption. Then, to show that our type system is sound with respect to the semantics, a proof of subject reduction is given, which implies type soundness [AC96].

The **imp**ς-calculus was chosen as the basis for the $\varrho\varpi\varsigma$-calculus for two reasons. First, the **imp**ς-calculus models an imperative programming style. This is desirable for modeling events because, in practice, event handlers often rely on side-effects to accomplish their purposes. The imperative style is also useful since our event mechanism supports several dynamic features: register, unregister, publish, and unpublish. All of these dynamic features use side-effects to carry out their dynamic semantics. Second, the **imp**ς-calculus was selected because it models an object-oriented programming style, which is useful for several reasons. First, the object-oriented style promotes reuse and integration, which compliments event mechanisms which also promote reuse and integration [SN92]. Second, it provides references, via object member labels, that describe what "events" handlers can handle. This is better than a λ-calculus where such labels would either be buried in closures or defined globally, lacking the necessary structure to easily determine what the possible set of events are and what those events relate to. Finally, since the **imp**ς-calculus provides good event references for handlers, it also, provides a store that is well suited for detecting implicit event announcement for handlers, i.e., events that are announced without the explicit use of any announcement construct. In short, the **imp**ς-calculus provides a good, expressive basis for the $\varrho\varpi\varsigma$-calculus formalism because its purpose coincides, in part, with the purpose of event mechanisms, and because it provides a model that is well suited for supporting event mechanisms.

4.1 Syntax and Informal Semantics

In this section we define and explain each part of the syntax for the $\varrho\varpi\varsigma$-calculus given in Figure 4.1. There are four distinct parts of the syntax: object syntax, value abstraction, event syntax, and type syntax. The object syntax is the same as the **imp**ς-calculus [AC96]. These constructs provide a way to create objects, use object methods, update object methods, and clone, i.e., make copies of, objects. The let construct in Figure 4.1 allows for value abstraction by associating the result of an expression (a) with a variable (x). The event syntax allows for events to be published and unpublished, and for handlers to be registered and unregistered. The type syntax provides a way to describe the type for objects and other values and expressions in the calculus.

4.1.1 *Object Syntax*

The computations expressed in the $\varrho\varpi\varsigma$-calculus are primarily based on the creation and manipulation of objects. The syntax for objects has labels that denote the name for each method followed by the definitions of the methods and enclosed in brackets, for example,

$$[label_1 = \varsigma(x)x.label_2, label_2 = \varsigma(y)[another = \varsigma(z)[\ ]\ ]\ ].$$

The $\varsigma(x)$ part of the method declaration specifies the self parameter that can be referenced in the body of the method. For example, the $label_1$ method in the above example has x specified as its self parameter; while the $label_2$ method has y specified as its self parameter. The self parameter provides methods with access to their own object, i.e., the object it resides in, similar to the `this` keyword in C++ or Java.

a, b, c, d	::=			
		x	variable	Object Syntax
	|	$[l_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$	object (l_i distinct)	
	|	$a.l$	method invocation	
	|	$a.l \Leftarrow \varsigma(x)b$	method update	
	|	$clone(a)$	cloning	
	|	$let\ x = a\ in\ b$	let	Value Abstraction
	|	$\varrho(a.l, b \equiv c)d$	register	Event Syntax
	|	$\not\varrho(a)$	unregister	
	|	$\varpi(a.l)$	publish	
	|	$\not\varpi(a.l)$	unpublish	
A, B, C	::=			
		K	constant type	Type Syntax
	|	Top	topmost type	
	|	$[l_i : B_i \ ^{i\in 1..n}]$	object type	
	|	$\mathcal{R}$	registry type	

Figure 4.1: $\varrho\varpi\varsigma$-calculus Syntax

$$
\begin{array}{l}
let\ cpy = [lc = \varsigma(x)[\ ]\ ]\ in \\
\quad let\ a = [l = \varsigma(x)[\ ]\ ]\ in \\
\quad\quad let\ reg = \varrho(a.l, [\ ] \equiv [\ ])let\ val = a.l\ in\ cpy.lc \Leftarrow \varsigma(x)val\ in \\
\quad\quad\quad let\ zz = \varpi(a.l)\ in \\
\quad\quad\quad\quad a.l \Leftarrow \varsigma(y)[l_2 = \varsigma(z)[\ ]\ ]
\end{array}
$$

Figure 4.2: Example of $\varrho\varpi\varsigma$-calculus registration

There are three different object constructs in the syntax that can be exercised to use and manipulate objects: method selection, method update, and object clone. Method selection, for example $x.label_2$, executes the $label_2$ method in x and returns its results. Method update, for example, $x.label_2 \Leftarrow \varsigma(y)[\ ]$, gives $label_2\ \varsigma(y)[\ ]$ as its new method, replacing the previous method. This is an in-place (imperative) update. Finally, object clone, e.g., $clone(a)$, makes a shallow copy of the object passed into it. So, suppose that a is $[l = \varsigma(x)[\ ]\ ]$. Then $clone(a)$ will return a copy of a such that, for example, $clone(a).l \Leftarrow \varsigma(y)[l_2 = \varsigma(z)[\ ]\ ]$, will only change a copy of a, leaving a's original l method intact.

4.1.2 *Value Abstraction*

Values in the calculus may be saved for later use with the let construct. For example, we could have the expression, $let\ a = [l = \varsigma(x)[\ ]]\ in\ clone(a).l \Leftarrow \varsigma(y)[l_2 = \varsigma(z)[\ ]\ ]$. This demonstrates how the previous example could be constructed, defining a value for a. It is also possible to use let expressions to serially compose two expressions. For example, in $let\ x = a.l \Leftarrow \varsigma(y)[l_2 = \varsigma(z)[\ ]\ ]\ in\ a.l.l_2$, the update will be completed before the select expression, $a.l.l_2$, is evaluated.

4.1.3 *Event Syntax and Informal Semantics*

Use of the $\varrho\varpi\varsigma$-calculus event mechanism is provided through publication and registration constructs and method update. Any method of an object in the $\varrho\varpi\varsigma$-calculus can be used as an event, so there is no need for syntax to explicitly declare events. The publication constructs essentially provide a way to allow or disallow methods to be used as events. For example, $\varpi(a.l)$ would cause $a.l$ to be announced every time it is updated; alternatively, $\not\varpi(a.l)$ would ensure that $a.l$ is not announced on update.

Registration of handlers is done using the ϱ construct. Figure 4.2 shows an example of a registration of a handler for $a.l$. In the example, every time $a.l$ is updated, the handler updates $cpy.lc$ with the new $a.l$ value.

The $b \equiv c$ part of the registration construct provides a way to limit the announcements that the handler

will actually handle. Specifically, if b and c evaluate to the same value, the handler, d, is allowed to handle the announcement. Otherwise, the handler will not be executed. In the example in Figure 4.2, we used $[\,] \equiv [\,]$ so that the handler is always allowed to handle announcements of $a.l$.

The idea behind the use of the $\equiv$ check is primarily to allow handlers to handle events only if a certain condition holds. This may seem superfluous, as such a restriction could be maintained with an `if-then` expression. However, the separation of the condition from the body of the handler allows the conditions to be executed at a different time from the handler bodies. As discussed later in this chapter, page 28, this flexibility allows us to define the operational semantics in a way that is better than if we were left with only the use of an `if-then`.

To undo a registration, the syntax provides the $\not\varrho$ construct. In order to successfully unregister a previously registered handler, we need to have the result of its earlier registration. For example, in Figure 4.2, we could use *reg* to unregister our handler like this: $\not\varrho(reg)$. If we did this in the example and then updated $a.l$ again, we would see that the handler that updates *cpy* would not have been executed as before, due to the unregister. While it might be more convenient if the unregistration syntax was more like the registration syntax, for example, $\not\varrho(a.l, d)$, where $a.l$ and d would be the same expressions used in the registration, the a and d expressions would not be guaranteed to uniquely identify what should be unregistered. That is, it would be possible for the same d expression to be used to register for $a.l$ in two different parts of a program. Likewise, the syntactic expression of a, i.e., the "actual text" for a, is not unique because the evaluation of a could differ based on the context of the evaluation (let expressions and self parameters define the context). Further, the result of the expression, a could not be used because the value of a could have changed between the time the handler registered for $a.l$ and the unregistration.

It is necessary here to have a unique key to use for unregistration to avoid ambiguity of the unregistration's results. Specifically, if a non-unique key is used for unregistration, we must undo every registration that matches that key to ensure that the "intended" unregistration occurs. For example, suppose that we use an expression text, a, as the key for unregistration. Suppose further that this a was used in two different contexts to register different handlers for different events. Now, in the process of doing an unregistration, using an expression text, such as a, as a key, we find both of the event-handler bindings that were created earlier. At this point, we have no way of knowing which binding is the intended target for unregistration. To ensure that the intended target is unregistered, we would need to unregister both handlers, which is unacceptable since the unregistration of the second handler would usually be unintentional.

4.1.4 *Type Syntax*

The type syntax for objects is similar to the expression syntax for objects, complete with brackets ('[' ']') and labels. The difference is that the labels are associated with the *return types* of the methods, rather than actual methods. For example, the type of $[l = \varsigma(x)[\,]\,]$ is $[l : [\,]\,]$. So, the type of an object has all the same labels as the relevant object, where each label has the result type of the label's associated method in the object. The registry type, $\mathcal{R}$, is the type of the result of a registration (ϱ). Constant types, denoted by K, can be used for types that are built-in, i.e., native, to the language, for example, integers, characters, etc. Constant types are only included for the sake of completeness since there are no expressions that yield such built-in types in the $\varrho\varpi\varsigma$-calculus. Since our type system will allow for subtyping, a top-most type *Top* is included such that every type is a subtype of *Top*. Unlike registration, since publication and unpublication expressions do not have special result values, they do not have their own special types.

4.2 Operational Semantics

We now formalize the meaning, or semantics, of the syntax using structural operational semantics [Plo81]. To specify the semantics, we define a reduction rule for each construct in the syntax. Each rule has three basic parts: hypotheses, judgments, and side-conditions. The notation for each rule is shown schematically as follows.

$$[\textit{rule name}] \quad (\text{where } \textit{side-conditions})$$
$$\frac{h_1 \quad h_2 \quad \ldots \quad h_k}{\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow v \cdot \sigma' \cdot R' \cdot P'}$$

The conclusion of a rule (below the line) holds when all the hypotheses for that rule (above the line) hold. Some rules have '$\forall$' hypotheses, i.e., a formula followed by "$\forall i \in 1..n$". This notation is an abbreviation for n distinct hypotheses — one for each i from 1 to n. So, for example, the notation

ι			store location
I			integer id
v	$::=$	$[l_i = \iota_i \ ^{i\in 1..n}]$	object result
	$\mid$	$(\iota,\ I)$	register result
σ	$::=$	$\iota_i \mapsto < \varsigma(x_i)b_i, S > \ ^{i\in 1..n}$	store
S	$::=$	$x_i \mapsto v_i \ ^{i\in 1..n}$	stack
P	$::=$	$\iota_i \ ^{i\in 1..m}$	publications
R	$::=$	$(\iota_i,\ I_j) \mapsto < b, c, d, S > \ ^{i\in 1..l,\ j\in 1..p_i}$	registered event handlers
	$\mid$	$(\iota_i,\ I_j) \mapsto <>$	unregistered handler
EVT	$::=$	$\{< b, c, d, S >_i^{i\in 1..n}\}$	set of registration contexts
	$\mid$	$\{< b, c, d, S >_i^{i\in 1..n}\} \cup \{<>\}$	set of registration and empty contexts
D	$::=$	$\{< b, c, d, S >_i^{i\in 1..n}\}$	set of filtered registration contexts
T	$::=$	$\{< b, c, d, S >\}$	accepted registration context
	$\mid$	$\emptyset$	rejected registration context

Figure 4.3: $\varrho\varpi\varsigma$-calculus Context and Value Syntax

$$\sigma \cdot R \cdot P \cdot S_i \vdash \diamond \quad \forall i \in 1..n$$

can be expanded to the following hypotheses,

$$\sigma \cdot R \cdot P \cdot S_1 \vdash \diamond \quad \sigma \cdot R \cdot P \cdot S_2 \vdash \diamond \quad \cdots \quad \sigma \cdot R \cdot P \cdot S_n \vdash \diamond.$$

Given a valid rule, the $a \rightsquigarrow v$ part of the judgment means that the construct represented by a will compute to a value v. The $\sigma \cdot R \cdot P \cdot S$ part of the judgment prior to the turnstile ($\vdash$) defines the *context* used for determining the value of v. Specifically, σ is the *store*, i.e., memory, where object values are kept and changed via method updates; R is the *registry*, containing the information for all event handler registrations; P is the *publication set* which is the set of events that are published (i.e., the set of methods that can implicitly announce events); and S is the *stack* which provides the associations for value abstraction from the use of *let* and self-parameters. The $\sigma' \cdot R' \cdot P'$ part of the judgment to the right of the $\rightsquigarrow$ denotes the updated versions of the store, registry, and publication set resulting from the reduction of a to v. By convention, any part of the context that is empty is written as ϕ.

Aside from the reduction rules for each construct in the syntax, we also need a way to ensure that each part of the context, σ, R, P, and S, is well-formed, that is, that each part of the context is valid with respect to the definition of reduction rules and the other parts of the context. For example, if the reductions for let and method selection do not allow a variable name to be used that is already on the stack, S, (and these are the only rules that modify the stack), then it would be impossible to have a stack that contains a particular variable twice. Thus, a requirement for a well-formed stack would be that all the variables contained in any given stack must be unique. We use '$\diamond$' and '$\circ$' to denote a well-formedness judgment of the context and the stack with respect to the rest of the context respectively. We use two different notations for well-formed contexts because stacks must be checked against both the store and the registration context. Further, the store and the registration context contain closures, each have stacks, that must be checked as well for well-formedness. Thus, we need judgments that are just for stacks and judgments that check the rest of the context.

We specify the well-formedness rules for the context as follows. First, the syntax for each part of the context is given in Figure 4.3. Next, the list of well-formed judgments is given that shows how judgments can be correctly expressed in Figure 4.4. That is, judgments that do not have one of the forms in Figure 4.4 are not valid judgments. Finally, the rules for checking the various contexts for well-formedness are listed in Figure 4.5. Note that in these rules, additions to iterated syntactic units are expressed with commas. For example, in the [Publish ι] rule, $(P,\ \iota)$ is part of the grammar for P in Figure 4.4.

The context and value syntax in Figure 4.3 defines what the contents of the different parts of the environment and what the results of an expression reduction (a computation) can be. The store locations are integers denoted by variations of ι (and occasionally ε) throughout the definition and use of the formalism. Object results (or values) are similar to object expressions except that the method is replaced with a reference to a location in the store. The results of registrations are represented as tuples of store references and integers denoted by variations of I that are registration identifiers. These tuples can be thought of as registration locations since they are used

$\sigma \vdash \diamond$	well-formed store judgment
$\sigma \cdot R \vdash \diamond$	well-formed registry judgment
$\sigma \cdot R \vdash \circ$	well-formed stacks of closures judgment
$\sigma \cdot R \cdot P \vdash \diamond$	well-formed published judgment
$\sigma \cdot R \cdot P \cdot S \vdash \circ$	well-formed stack judgment
$\sigma \cdot R \cdot P \cdot S \vdash \diamond$	well-formed context judgment
$\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow v \cdot \sigma' \cdot R' \cdot P'$	term reduction judgment
$\sigma \cdot R \cdot P \cdot S \vdash EVT \rightharpoonup D \cdot \sigma' \cdot R' \cdot P'$	subsystem closure judgment
$\vdash (< b, c, d, S >, v, w) \rightharpoonup T$	subsystem $\equiv$ judgment
$\sigma \cdot R \cdot P \cdot S \vdash D \rightharpoonup \sigma' \cdot R' \cdot P'$	subsystem handler judgment

Figure 4.4: $\varrho\varpi\varsigma$-calculus Well-Formed Judgments

[Store ϕ]

$$\frac{}{\phi \vdash \diamond}$$

[Store ι] (where $\iota \notin dom(\sigma)$)

$$\frac{\sigma \vdash \diamond}{(\sigma,\ \iota \mapsto < \varsigma(x)b, S >) \vdash \diamond}$$

[Registry ϕ]

$$\frac{\sigma \vdash \diamond}{\sigma \cdot \phi \vdash \diamond}$$

[Registry $(\iota,\ I)$] (where $\iota \in dom(\sigma)$ and $(\iota,\ I) \notin dom(R)$)

$$\frac{\sigma \cdot R \vdash \diamond}{\sigma \cdot (R,\ (\iota,\ I) \mapsto < b, c, d, S >) \vdash \diamond}$$

[Closure Stacks Valid]

$$\left(\begin{array}{ll} \text{where} & \\ \sigma = & \iota_i \mapsto < \varsigma(x_i)b_i, S_i >^{i \in 1..n} \text{ and} \\ R = & (\iota_j,\ I_{k_j}) \mapsto < b_{(j,k_j)}, c_{(j,k_j)}, d_{(j,k_j)}, S_{(j,k_j)} >^{j \in 1..p,\ k_j \in 1..q_j}, \\ & (\iota_{j'},\ I_{k_{j'}}) \mapsto <>^{j' \in p+1..p',k_{j'} \in 1..q_{j'}} \end{array} \right)$$

$$\frac{\sigma \cdot R \cdot \phi \cdot S_i \vdash \circ \quad \forall i \in 1..n \quad \sigma \cdot R \cdot \phi \cdot S_{(j,k_j)} \vdash \circ \quad \forall j \in 1..p \quad \forall k_j \in 1..q_j}{\sigma \cdot R \vdash \circ}$$

[Publish ϕ]

$$\frac{\sigma \cdot R \vdash \diamond \quad \sigma \cdot R \vdash \circ}{\sigma \cdot R \cdot \phi \vdash \diamond}$$

[Publish ι] (where $\iota \notin P$ and $\iota \in dom(\sigma)$)

$$\frac{\sigma \cdot R \cdot P \vdash \diamond}{\sigma \cdot R \cdot (P,\ \iota) \vdash \diamond}$$

[Stack ϕ]

$$\frac{}{\sigma \cdot R \cdot P \cdot \phi \vdash \circ}$$

[Stack x Regval] (where $x \notin dom(S)$ and $(\iota,\ I) \in dom(R)$)

$$\frac{\sigma \cdot R \cdot P \cdot S \vdash \circ}{\sigma \cdot R \cdot P \cdot (S,\ x \mapsto (\iota,\ I)) \vdash \circ}$$

[Stack x Object] (where $x \notin dom(S)$, $\iota_i \in dom(\sigma)$, and l_i, ι_i distinct, $\forall i \in 1..n$)

$$\frac{\sigma \cdot R \cdot P \cdot S \vdash \circ}{\sigma \cdot R \cdot P \cdot (S,\ x \mapsto [l_i = \iota_i\ ^{i \in 1..n}]) \vdash \circ}$$

[Context $\diamond$]

$$\frac{\sigma \cdot R \cdot P \vdash \diamond \quad \sigma \cdot R \cdot P \cdot S \vdash \circ}{\sigma \cdot R \cdot P \cdot S \vdash \diamond}$$

Figure 4.5: $\varrho\varpi\varsigma$-calculus Context Judgments

to reference the handlers associated with a particular method.

The various parts of the environment maintain information related to different store and registration values. The '$\mapsto$' symbol is used to denote a mapping between a value and some information. The store, σ, is a mapping between integers, ι, and closures, $< \varsigma(x_i)b_i, S >$. A closure can be thought of as a "frozen" computation. That is, a closure encapsulates both the "code" that should be executed as well as the environment in which it should be executed. The closures saved in the store are from the method definitions of evaluated object expressions. The environment saved with the code is the stack, S, from the computation that the closure came from. Stacks, S, are mappings from variable names to values. For example, an entry in a stack, $x \mapsto [l = \iota]$, means that the expression x will evaluate to the object value $[l = \iota]$. The publication set, P, simply keeps track of which store locations will be allowed to announce events. That is, for any ι in P, when the value of ι is updated in the store, an event for that update will be announced; if ι is not in P, no event will be announced for updates of ι. The registry, R, is a mapping from registration values to handler closures, $< b, c, d, S >$, and empty closures, $<>$. Event closures are similar to the method closures except they store three "frozen" computations, the announcement conditions, b and c, and the handler, d.

The empty closures prevent registration identifier aliasing. If the registration mappings were removed upon unregistration, there would be nothing to prevent the identifier to be used for a subsequent registration, thus allowing identifiers to be aliased. Aliasing registration identifiers in this way would be problematic since it would allow the client of the prior registration to unregister the latter one. For example, suppose client A registers to handle an event and receives a registration identifier, r. Client A then unregisters its registration. An unrelated client, B, then registers for an event and also receives registration identifier r. Client A, for whatever reason, then attempts to perform another unregistration using r, unregistering B's handler. This is clearly a violation of the expected behavior of both clients A and B.

To understand the context judgment rules, Figure 4.5, it is best to start with the last rule and work backwards. The hypotheses for the [Context $\diamond$] rule ensure that the stack, S, is well formed ($\circ$) and that the rest of the context, σ, R, and P, is also well formed ($\diamond$). The [Stack x Object] and [Stack x Regval] rules ensure that the stack is valid when the entries in the stack map to object values and registration values respectively. The side-conditions for the [Stack x Object] rule ensure that entries that map to object values in the stack have unique variable names ($x \notin dom(S)$), that the object values reference valid locations in the store ($\iota_i \in dom(\sigma)$), and that each label and related store location is unique for the object value. The side-conditions for the [Stack x Regval] rule ensure that entries that map to registration values in the stack have unique variable names and that the registration values are in the registry. The [Stack ϕ] rule is an axiom stating that an empty stack is always well-formed. The [Publish ι] rule ensures that entries, ι, in the publication set are not repeated and that ι is a valid store location. For empty publication sets, the [Publish ϕ] rule ensures that the remaining parts of the context, σ and R, are well-formed ($\diamond$) and that the stacks in both σ and R are well-formed ($\circ$). The hypotheses of the [Closure Stacks Valid] rule ensure that the stacks of all the method and event closures are well-formed. The [Registry $(\iota,\ I)$] rule ensures that the store location, ι, in the registry value, $(\iota,\ I)$, exists in the store, σ, and that the registry value is unique in the registry. For empty registries, the [Registry ϕ] rule ensures that the rest of the context, in this case, the store, σ, is well-formed. The [Store ι] rule ensures that all the store locations in the store, σ are unique. Finally, the [Store ϕ] rule is an axiom that states that an empty store is well-formed.

Now that we have a system set up for ensuring that expressions and contexts are well-formed, we can proceed to the operational semantics reduction rules for the various constructs in the syntax. The rules for variables, objects, object manipulation, and let, in Figure 4.6, are all taken directly from Abadi and Cardelli, [AC96, Chapter 10], modulo the addition of the R and P contexts and the extra side-condition in the [Red Update] rule, $\iota_j \notin P$. This side condition is necessary since both the [Red Update] and [Red Update and Announce] rules handle updates. The rules for events, in Figure 4.7, handle both method update, for the case when the method is published, and the registration and publication constructs. To make the method update rule easier to understand as well as easier to change, we have defined the announcement and handler execution semantics as a subsystem of rules in Figure 4.8. The notation, $\sigma, \iota \leftarrow < \varsigma(x)b, S >$ in the [Red Update] and [Red Update and Announce] rules means ι in the store, σ, is updated with the given closure, $< \varsigma(x)b, S >$.

4.2.1 *Object Semantics*

Since all of the computations in the $\varrho\varpi\varsigma$-calculus are based on objects, the original **imp**ς-calculus semantics, defined in Figure 4.6, composes the core computational semantics for the $\varrho\varpi\varsigma$-calculus. Variables are resolved

[Red x]

$$\frac{\sigma \cdot R \cdot P \cdot (S',\ x \mapsto v,\ S'') \vdash \diamond}{\sigma \cdot R \cdot P \cdot (S',\ x \mapsto v,\ S'') \vdash x \rightsquigarrow v \cdot \sigma \cdot R \cdot P}$$

[Red Object] (where $\forall i \in 1..n,\ \iota_i \notin dom(\sigma)$, and ι_i distinct)

$$\frac{\sigma \cdot R \cdot P \cdot S \vdash \diamond}{\sigma \cdot R \cdot P \cdot S \vdash [l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}] \rightsquigarrow [l_i = \iota_i\ ^{i\in 1..n}] \cdot (\sigma,\ \iota_i \mapsto < \varsigma(x_i)b_i, S >^{i\in 1..n}) \cdot R \cdot P}$$

[Red Select] (where $\sigma''(\iota_j) = < \varsigma(x_j)b_j, S'' >$, $x_j \notin dom(S'')$, and $j \in 1..n$)

$$\frac{\begin{array}{c}\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow [l_i = \iota_i\ ^{i\in 1..n}] \cdot \sigma'' \cdot R'' \cdot P'' \\ \sigma'' \cdot R'' \cdot P'' \cdot (S'',\ x_j \mapsto [l_i = \iota_i\ ^{i\in 1..n}]) \vdash b_j \rightsquigarrow v \cdot \sigma' \cdot R' \cdot P'\end{array}}{\sigma \cdot R \cdot P \cdot S \vdash a.l_j \rightsquigarrow v \cdot \sigma' \cdot R' \cdot P'}$$

[Red Update] (where $j \in 1..n$, $\iota_j \in dom(\sigma')$, and $\iota_j \notin P'$)

$$\frac{\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow [l_i = \iota_i\ ^{i\in 1..n}] \cdot \sigma' \cdot R' \cdot P'}{\sigma \cdot R \cdot P \cdot S \vdash a.l_j \Leftarrow \varsigma(x)b \rightsquigarrow [l_i = \iota_i\ ^{i\in 1..n}] \cdot (\sigma',\ \iota_j \leftarrow < \varsigma(x)b, S >) \cdot R' \cdot P'}$$

[Red Clone] (where $\forall i \in 1..n$, $\iota_i \in dom(\sigma')$, $\iota'_i \notin dom(\sigma')$, and ι'_i distinct)

$$\frac{\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow [l_i = \iota_i\ ^{i\in 1..n}] \cdot \sigma' \cdot R' \cdot P'}{\sigma \cdot R \cdot P \cdot S \vdash clone(a) \rightsquigarrow [l_i = \iota'_i\ ^{i\in 1..n}] \cdot (\sigma',\ \iota'_i \mapsto \sigma(\iota_i)^{i\in 1..n}) \cdot R' \cdot P'}$$

[Red Let] (where $x \notin dom(S)$)

$$\frac{\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow v'' \cdot \sigma'' \cdot R'' \cdot P'' \quad \sigma'' \cdot R'' \cdot P'' \cdot (S,\ x \mapsto v'') \vdash b \rightsquigarrow v' \cdot \sigma' \cdot R' \cdot P'}{\sigma \cdot R \cdot P \cdot S \vdash let\ x = a\ in\ b \rightsquigarrow v' \cdot \sigma' \cdot R' \cdot P'}$$

Figure 4.6: $\varrho\varpi\varsigma$-calculus Object Semantics [AC96, Ch. 10]

[Red Update and Announce] $\left(\begin{array}{l}\text{where} \\ \quad j \in 1..n,\ \iota_j \in dom(\sigma'),\ \iota_j \in P', \\ \quad D \subseteq R'(\iota_j), \text{ and} \\ \quad R'(\iota_j) = \{x \mid \exists I.R'((\iota,\ I)) = x\}\end{array}\right)$

$$\frac{\begin{array}{c}\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow [l_i = \iota_i\ ^{i\in 1..n}] \cdot \sigma' \cdot R' \cdot P' \\ R' \cdot (\sigma', \iota_j \leftarrow < \varsigma(x)b, S >) \cdot R' \cdot P' \vdash R'(\iota_j) \rightharpoonup D \cdot \sigma'' \cdot R'' \cdot P'' \\ R' \cdot \sigma'' \cdot R'' \cdot P'' \vdash D \rightharpoondown \sigma''' \cdot R''' \cdot P'''\end{array}}{\sigma \cdot R \cdot P \cdot S \vdash a.l_j \Leftarrow \varsigma(x)b \rightsquigarrow [l_i = \iota_i\ ^{i\in 1..n}] \cdot \sigma''' \cdot R''' \cdot P'''}$$

[Red Register] (where $j \in 1..n$, $(\iota_j,\ I) \notin dom(R')$, and $R'' = R',\ (\iota_j,\ I) \mapsto < b, c, d, S >$)

$$\frac{\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow [l_i = \iota_i\ ^{i\in 1..n}] \cdot \sigma' \cdot R' \cdot P'}{\sigma \cdot R \cdot P \cdot S \vdash \varrho(a.l_j, b \equiv c)d \rightsquigarrow (\iota_j,\ I) \cdot \sigma' \cdot R'' \cdot P}$$

[Red Unregister] (where $R'' = (R',\ (\iota,\ I) \leftarrow <>)$)

$$\frac{\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow (\iota,\ I) \cdot \sigma' \cdot R' \cdot P'}{\sigma \cdot R \cdot P \cdot S \vdash \not{\varrho}(a) \rightsquigarrow [\,] \cdot \sigma' \cdot R'' \cdot P'}$$

[Red Publish] (where $j \in 1..n$)

$$\frac{\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow [l_i = \iota_i\ ^{i\in 1..n}] \cdot \sigma' \cdot R' \cdot P'}{\sigma \cdot R \cdot P \cdot S \vdash \varpi(a.l_j) \rightsquigarrow [\,] \cdot \sigma' \cdot R' \cdot P' \cup \{\iota_j\}}$$

[Red Unpublish] (where $j \in 1..n$ and $P'' = P' \setminus \{\iota_j\}$)

$$\frac{\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow [l_i = \iota_i\ ^{i\in 1..n}] \cdot \sigma' \cdot R' \cdot P'}{\sigma \cdot R \cdot P \cdot S \vdash \not{\varpi}(a.l_j) \rightsquigarrow [\,] \cdot \sigma' \cdot R' \cdot P''}$$

Figure 4.7: $\varrho\varpi\varsigma$-calculus Event Semantics

[Event Red Filter] (where EVT is a set of event and empty closures)

$$\frac{\sigma \cdot R \cdot P \cdot S' \vdash b \rightsquigarrow u \cdot \sigma' \cdot R' \cdot P' \quad \sigma' \cdot R' \cdot P' \cdot S' \vdash c \rightsquigarrow w \cdot \sigma'' \cdot R'' \cdot P'' \quad \vdash (< b, c, d, S' >, u, w) \rightharpoonup \hat{D} \quad R_0 \cdot \sigma'' \cdot R'' \cdot P'' \vdash EVT \rightharpoonup D \cdot \sigma''' \cdot R''' \cdot P'''}{R_0 \cdot \sigma \cdot R \cdot P \vdash EVT \uplus \{< b, c, d, S' >\} \rightharpoonup D \uplus \hat{D} \cdot \sigma''' \cdot R''' \cdot P'''}$$

[Event Red Empty] (where EVT is a set of event and empty closures)

$$\frac{R_0 \cdot \sigma \cdot R \cdot P \vdash EVT \rightharpoonup D \cdot \sigma' \cdot R' \cdot P'}{R_0 \cdot \sigma \cdot R \cdot P \vdash EVT \uplus \{<>\} \rightharpoonup D \cdot \sigma' \cdot R' \cdot P'}$$

[Event Red Filter $\emptyset$]

$$\frac{}{R_0 \cdot \sigma \cdot R \cdot P \vdash \emptyset \rightharpoonup \emptyset \cdot \sigma \cdot R \cdot P}$$

[Event Red $\equiv$] (where $u \equiv w$)

$$\frac{}{\vdash (< b, c, d, S >, u, w) \rightharpoonup \{< b, c, d, S >\}}$$

[Event Red $\not\equiv$] (where $u \not\equiv w$)

$$\frac{}{\vdash (< b, c, d, S >, u, w) \rightharpoonup \emptyset}$$

[Event Red Handler] (where D is a set of tuples of handlers and stacks)

$$\frac{\sigma \cdot R \cdot P \cdot S' \vdash d \rightsquigarrow v \cdot \sigma' \cdot R' \cdot P' \quad R_0 \cdot \sigma' \cdot R' \cdot P' \vdash D \multimap \sigma'' \cdot R'' \cdot P''}{R_0 \cdot \sigma \cdot R \cdot P \vdash D \uplus \{< b, c, d, S' >\} \multimap \sigma'' \cdot R'' \cdot P''}$$

[Event Red Handler $\emptyset$]

$$\frac{}{R_0 \cdot \sigma \cdot R \cdot P \vdash \emptyset \multimap \sigma \cdot R \cdot P}$$

Figure 4.8: $\varrho\varpi\varsigma$-calculus Event Announcement Subsystem

$$\dfrac{\dfrac{\vdots\ [\text{Assumption}]}{\sigma\cdot\phi\cdot\phi\vdash\diamond}\quad\dfrac{\dfrac{}{\sigma\cdot\phi\cdot\phi\cdot\phi\vdash\diamond}\,[\text{Stack }\phi]}{\sigma\cdot\phi\cdot\phi\cdot x\mapsto[l=\iota]\vdash\diamond}\,[\text{Stack x Object}]}{\dfrac{\sigma\cdot\phi\cdot\phi\cdot x\mapsto[l=\iota]\vdash\diamond}{\sigma\cdot\phi\cdot\phi\cdot x\mapsto[l=\iota]\vdash x\rightsquigarrow[l=\iota]\cdot\sigma\cdot\phi\cdot\phi}\,[\text{Red x}]}\,[\text{Context }\diamond]$$

Figure 4.9: [Red x] Rule Application Example

For the figure, $\sigma_0 = (\iota \mapsto < \varsigma(x)x, \phi >)$

$$\dfrac{\left[\dfrac{\dfrac{\vdots}{\phi\cdot\phi\cdot\phi\cdot\phi\vdash\diamond}}{\phi\cdot\phi\cdot\phi\cdot\phi\vdash[l=\varsigma(x)x]\rightsquigarrow[l=\iota]\cdot\sigma_0\cdot\phi\cdot\phi}\,[\text{Red Object}]\right]\quad\left[\dfrac{\vdots}{\sigma_0\cdot\phi\cdot\phi\cdot(x\mapsto[l=\iota])\vdash x\rightsquigarrow[l=\iota]\cdot\sigma_0\cdot\phi\cdot\phi}\,[\text{Red x}]\right]}{\phi\cdot\phi\cdot\phi\cdot\phi\vdash[l=\varsigma(x)x].l\rightsquigarrow[l=\iota]\cdot\sigma_0\cdot\phi\cdot\phi}\,[\text{Red Select}]$$

Figure 4.10: [Red Select] Rule Computation Example

with the [Red x] rule by matching the variable expression with a variable in the domain of the stack and returning the value associated with that variable. For example, assume we have the context and expression, $\sigma\cdot\phi\cdot\phi\cdot x\mapsto[l=\iota]\ \vdash\ x$. Figure 4.9 demonstrates the application of the [Red x] rule, assuming that σ is well-formed and $\iota \in dom(\sigma)$.

Object expressions are handled by the [Red Object] rule. This rule works by expanding the store to accommodate the new methods defined by the object, associating the new store locations with their respective method closures, and then returning the object value that contains the new store locations. For example, the syntactic form, $[l = \varsigma(x)x]$, would compute to the object value $[l = \iota]$. The resulting store from the computation would map the location ι to the closure for l, $(\sigma,\ \iota \mapsto < \varsigma(x)x, S >)$, where S would be the original stack and σ the original store of the computation.

The [Red Select] rule handles method selection. The rule works by, first, evaluating the selection target (a in the figure) to an object value. Next, the rule evaluates the body of method, which is stored in the closure associated with the store location for the selected label, using the stack stored in the closure and expanded with the self parameter (x in the figure) associated with the object value — the result is then the result of evaluating the body. Figure 4.10 is an example of the application of the [Red Select] rule.

The [Red Update] rule also evaluates the target, a, to an object value. However, instead of evaluating the body of the method, the method is replaced with a new one. It replaces the old method by making a closure with the body of the new method and the current stack and updating the store location for the method with the new closure. So, for example, assume that a evaluates to, $[l = \iota]$. Also, let the current store map ι to $< \varsigma(x)[\,], S >$. Then, the update expression, $a.l \Leftarrow \varsigma(x)x$, would change the store to map ι to $< \varsigma(x)x, S' >$, where S' is the stack that the update is evaluated under. The result of evaluating an update expression is the target's object value, in this case, $[l = \iota]$.

The [Red Clone] rule also evaluates the selection target, a, to an object value, then expands the store and copies the closures of the object value to the new store locations. This rule returns an object value that references the new store locations. So, again assume that a is $[l = \iota]$ and the store maps ι to $< \varsigma(x)[\,] >$. Then, the result of evaluating $clone(a)$ would be $[l = \iota']$ and the new store would map ι to $< \varsigma(x)[\,] >, \iota' \mapsto < \varsigma(x)[\,] >$.

Finally, the [Red Let] rule works by first evaluating the initialization expression, a; then it evaluates the body of the let, b, using the stack, S, extended to give an association from the variable, x, to the result of evaluating a. For example, the let expression, $let\ a = [l = \varsigma(x)[\ ]]\ in\ a.l$, would be evaluated by, first, evaluating $[l = \varsigma(x)[\ ]]$. The result of evaluating this object would be that the store is updated, mapping ι to $< \varsigma(x)[\ ], S >$, where S is the original stack for the let expression, yielding $[l = \iota]$. Then $a.l$ is evaluated with a modified stack, namely, $(S, a \mapsto [l = \iota])$, yielding the object value $[\ ]$.

4.2.2 *Event Semantics*

The reduction rules for events, Figure 4.7, are relatively straightforward, except for, perhaps, the [Red Update and Announce] rule. This rule uses an operational semantics subsystem, Figure 4.8, to determine, based on the update, the event handlers that should be invoked. This is explained below.

The [Red Register] rule first evaluates the event holder, a, to an object value. Then, it updates the registry resulting from that evaluation, R', to associate the event, ι_j (a store location that represents a method of an object, in this case, $a.l_j$) to the announcement and handler conditions, b, c, and d, respectively. The rule also associates an integer, I, that is unique with respect to the given ι_j, with the handler registration to provide a unique way to identify this particular registration. Having such an unique identifier will prevent ambiguous unregistrations, e.g., an unregistration for a store location with two handlers. Finally, the rule returns the unique registration id, which is a tuple that has both the event, i.e., store location, ι_j and the unique integer, I.

To illustrate how the [Red Register] rule works, assume that there is a stack, $S_0 = a \mapsto [l = \iota]$, and a store, $\sigma_0 = \iota \mapsto < \varsigma(x)[\ ], \phi >$. Now, suppose that $\varrho(a.l, [\ ] \equiv [\ ])a.l \Leftarrow \varsigma(y)y$, is evaluated. The a will be evacuated to be $[l = \iota]$, thus the registration environment, R, is updated to be, $R, (\iota, I) \mapsto < [\ ], [\ ], a.l \Leftarrow \varsigma(y)y, S_0 >$ where (ι, I) is a unique integer tuple in R. The tuple (ι, I) is the value that results from the registration evaluation.

Unregistration is handled by the [Red Unregister] rule. The unregistration expression takes an expression that evaluates to a registration value. The [Red Unregister] rule first reduces the expression, a in the figure, down to the registration value. It then replaces the closure that the registration value maps to with an empty closure, $<>$. Thus, $\not\varrho(rval)$ unmaps the registration represented by $rval$. So, where $S_1 = rval \mapsto (\iota, I)$ and $R' = R, (\iota, I) \mapsto < [\ ], [\ ], a.l \Leftarrow \varsigma(y)y, S_2 >$, the new R' will be $R', (\iota, I) \mapsto <>$.

The publication rules, [Red Publish] and [Red Unpublish], maintain the publication set, P. The publication set determines which events are allowed to be announced (i.e., whether the [Red Update] or the [Red Update and Announce] rule is applicable for a method update). The [Red Publish] rule inserts the store location related to the publication target event, $a.l_j$ in the figure. That is, the rule first ensures that a is an object and that l_j is a member of that object. Then, the store location, ι_j, related to the l_j in the object value is added to the set of events that can be announced, P. The [Red Unpublish] rule also ensures for the unpublication target event, $a.l_j$ in the figure, that a evaluates to an object value and that l_j is a valid member of a. Then, the store location related to l_j in the object value is removed from the set of events that can be announced, P. Since P is treated as a set, the publish and unpublish operations are idempotent, i.e., multiple publications or unpublications of an event has the same effect as if the event was only published or unpublished once.

Event announcement is handled by the [Red Update and Announce] rule. The rule first handles the method update just as in the [Red Update] rule. Then, to find and evaluate all the appropriate handlers of the update event announcement, the rule enlists the help of the subsystem in Figure 4.8 via the '$\rightharpoonup$' and '$\rightharpoondown$' reduction hypotheses. The first subsystem hypothesis, the '$\rightharpoonup$' reduction, determines the handlers that should receive the update, i.e., it filters the handlers. The second hypothesis, the '$\rightharpoondown$' reduction, then executes that set of handlers.

To filter the handlers, the [Event Red Filter] and [Event Red Empty] rules recurse through the set of event closures that are related to the event via the registry environment, represented in these rules as EVT ([Red Update and Announce], Figure 4.7, demonstrates how the EVT set is constructed). The [Event Red Filter] rule handles the active registrations by evaluating the announcement conditions of the each non-empty event closure in EVT, b and c in the figure. Then, the results of b and c, which are u and w respectively, are tested with the $\equiv$ relation, defined in Definition 4.2.1, below, via the [Event Red $\equiv$] and [Event Red $\not\equiv$] rules; these auxiliary rules used to simplify the [Event Red Filter] rule. The [Event Red Empty] rule handles the entries in EVT that have been unregistered, i.e., the entries that map to empty event closures. This rule simply discards the empty closure and continues. The basis of this recursion through EVT, i.e., $EVT = \emptyset$, is handled by the [Event Red Filter $\emptyset$] rule, an axiom that just seeds the result set of this filtering process, D in the figure, with the empty set. Once all the valid handlers are found and placed in D, the [Event Red Handler] rule recurses through this

set of handlers and evaluates each one. The basis of this recursion, when D is empty, is handled by the [Event Red Handler Ø] rule, an axiom ending the recursion.

Definition 4.2.1 (≡, identical values). Let $\iota_1, \ldots, \iota_n$, ι_{i_1}, and ι_{j_1} be store locations, I_{i_2} and I_{j_2} be registry indices, $l_1, \ldots, l_n$, be method labels, A and B be types, and v and w be values. Then, the relation, $\equiv$, is defined as follows:

i) $[l_i = \iota_i \ ^{i \in 1..n}] \equiv [l_j = \iota_j \ ^{j \in 1..n}]$ where $\forall i . \exists j$ such that $\iota_i = \iota_j$, and $l_i = l_j$
ii) $(\iota_{i_1},\ I_{i_2}) \equiv (\iota_{j_1},\ I_{j_2})$ when $\iota_{i_1} = \iota_{j_1}$ and $I_{i_2} = I_{j_2}$

The definition of $\equiv$ may seem overly restrictive, since it requires objects to be *exactly* the same value. An alternative would be to relax that constraint such that the object comparison would hold if the values in the store, the ι's, for the corresponding methods in the two objects were equal. For example, define objects a and b as follows, $a \triangleq [\, l = \varsigma(x)[\ ]\,]$ and $b \triangleq [\, l = \varsigma(x)[\ ]\,]$. The comparison of a and b using the relaxed definition would succeed. However, since the evaluation of an object always produces a new, unique object value, the store locations for the methods in the objects will be unique, i.e. $[\, l = \iota_i \,]$ and $[\, l = \iota_j \,]$ where $\iota_i \neq \iota_j$.

The definition of $\equiv$ would be similar to, for example, saying two strings are equivalent in C only if their base addresses are the same, which would obviously be inconvenient. However, there is no native support for string comparison built into C for string comparison, leaving it up to a library or the user to accomplish such a task. Doing this simplifies C, making C easier to implement and more adaptable, e.g., ASCII vs. Unicode. This is similar to the reason for the strictness of $\equiv$ for the $\varrho\varpi\varsigma$-calculus: it makes the semantics for the calculus reasonably straightforward and it provides another degree of freedom for any work based on the $\varrho\varpi\varsigma$-calculus in the future, that may, for example, provide event filtering. One simple way that $\equiv$ can be easily satisfied is to define some constant objects up front, for example, objects that represent Boolean values: true and false. Then, one side of the $\equiv$ part of the registration could be an expression that evaluates to either true or false, and the other side could always be true, allowing for a simple Boolean event filter.

It would be possible to leave event filtering out of the event mechanism altogether, leaving it up to event handlers to implement any kind of desired filtering. However, this approach can introduce problems and limit the power of filtering. In systems that provide concurrency, race conditions on filtering could be a problem. For example, two handlers are invoked as a result of an event, at the time the event is announced, handler one would pass its filtering, but handler two would not. However, suppose that handler one is allowed to execute to a point where it changes the results of the filter for handler two before handler two has the opportunity to evaluate it. The intended semantic of the filter for handler two would be violated, potentially causing serious problems.

This problem points to how this approach of allowing event filtering to be interleaved with event handling limits the power of filtering. That is, it may be desirable to ensure that all filters are evaluated for a given event announcement *before* any handlers are allowed to proceed. While it may be possible to ensure this without direct support for event filtering in a language, perhaps with the use of some form of synchronization, it would be difficult, in general, to achieve. Plus, synchronization and other non-language native approaches would have the distinct disadvantage of forcing every handler for a given event to have, at least, some common knowledge amongst them, e.g., a common lock, breaking one of the primary goals for using events: decreasing unnecessary coupling. The $\varrho\varpi\varsigma$-calculus, therefore, provides a way to do event filtering.

The semantics for event announcement is that the filters are all evaluated before the handlers are executed, i.e., filter evaluation is not interleaved with handler execution, minimizing the amount of change that can occur between the time an event is announced and the last event filter has been evaluated. It could be possible to further restrict the calculus such that the filtering could not perform updates or cause other side-effects, eliminating the possibility that the state of a system could change between the time of event announcement and event filter evaluation occurs, unless, of course, concurrency is allowed. In that case we could still have the situation where one thread is working on evaluation of event filters while another thread, unrelated to any event announcement, is doing updating some values that a filter might rely on.

The $\varrho\varpi\varsigma$-calculus semantics shows how the publication, P, registry, R, and store, σ, environments work together to provide support for events. For example, we can see how the [Red Update], Figure 4.6, and [Red Update and Announce] rules, Figure 4.7, use P to decide whether to announce an event, and how the [Red Update and Announce] rule uses R to determine what conditions and subsequent handlers to execute.

$$\text{[Env } \phi\text{]} \quad \frac{}{\phi \vdash \diamond} \qquad \text{[Env x]} \quad \text{(where } x \notin dom(E)\text{)} \quad \frac{E \vdash A}{E, x : A \vdash \diamond} \qquad \text{[Type } \mathit{Top}\text{]} \quad \frac{E \vdash \diamond}{E \vdash \mathit{Top}}$$

$$\text{[Type } K\text{]} \quad \frac{E \vdash \diamond}{E \vdash K} \qquad \text{[Type Object]} \quad \text{(where each } l_i \text{ is distinct)} \quad \frac{E \vdash B_i \quad \forall i \in 1..n}{E \vdash [l_i : B_i \ ^{i \in 1..n}]}$$

Figure 4.11: $\varrho\varpi\varsigma$-calculus Type Environment and Well-Formedness Rules [AC96, Ch. 11]

4.3 Type System

A type system for the $\varrho\varpi\varsigma$-calculus should ensure that any expression that can have a valid type will also have meaning with respect to the semantics rules. That is, the evaluation of a typed expression will not get "stuck". A reduction is *stuck* if there is no value and there is no rule that can satisfy a hypothesis of one of the steps of the reduction. For example, the reduction of $[\, l_1 = \varsigma(x)[\,]\,].l_2$ will get stuck since the object has no method labeled l_2. A type system with the property of ensuring that only reduceable expressions are typed is called a *sound* type system. In this section, a type system for the $\varrho\varpi\varsigma$-calculus syntax will be presented. Then, we will show that the type system is sound with respect to the $\varrho\varpi\varsigma$-calculus semantics.

The type system defined for the $\varrho\varpi\varsigma$-calculus is the same as the type system defined for the **imp**ς-calculus [AC96], extended to handle the new expressions and semantics that the $\varrho\varpi\varsigma$-calculus defines. Thus, like the **imp**ς-calculus type system, our type system provides types for objects, such as for $[\, l = \varsigma(x)[\,]\,]$, which has type, $[\, l : [\,]\,]$. The type syntax $a : b$ means that a will have a result that conforms to b. So, instead of explaining the example using English, i.e., saying, "which has type," a colon is used: $[\, l = \varsigma(x)[\,]\,] : [\, l : [\,]\,]$. Typing for objects is based on the structure of the object. In the example, the object had one method, l, that returns an empty object, the type of the object reflects this. As another example, take the object, $[\, l = \varsigma(x)\,[\, p = \varsigma(y)x.n\,],\ n = \varsigma(z)[\,]\,]$. The type for this object will be, $[\, l : [\, p : [\,]\,],\ n : [\,]\,]$, following the structure of the object itself.

Also like the **imp**ς-calculus, the $\varrho\varpi\varsigma$-calculus includes structural subtyping. *Structural subtyping* is subtyping based on the structure, i.e., method names and types, of the object, as opposed to by-name subtyping, as in Java, C++, C#, etc., where the subtyping relationship is defined using type names is declared. For example, in Java we would write, `class A extends B`, which means that `A` will be a subtype of `B`.

The basic idea of structural subtyping in the calculus for object types is that one object type, A, is a subtype of another, B, when A has, at least, all of the same methods as B and all of those methods have the same result type. For example, an object with type $[\, l_1 : B_1,\ l_2 : B_2\,]$ would be a subtype of both object types $[\, l_1 : B_1\,]$ and $[\, l_2 : B_2\,]$.

As shown in Figure 4.1, the $\varrho\varpi\varsigma$-calculus type system consists of object types, registry types, constant types, and the *Top* type. Most of the typing rules deal with object types, since most of the $\varrho\varpi\varsigma$-calculus expressions deal with objects. Registry types are used for registration results. Constant types allow for built-in types. Finally, the *Top* type is a type that is a supertype of every type.

4.3.1 *Well-formed Type Environment and Types*

The type environment, E must be well-formed in order for any typing judgment to be valid. Further, every type used in the judgment must also be well-formed. The rules for checking the type environment and types for well-formedness are in Figure 4.11. A type environment is checked for well-formedness using the [Env x] rule. The rule judges an entry in the type environment well-formed if the entry's type is valid, as long as the variable name for the entry (x in the figure) is not listed again in the rest of the typing environment. The [Env ϕ] axiom states that an empty type environment is well-formed. The [Type K] rule declares constant types valid if the type environment is valid. The [Type *Top*] rule declares the *Top* type valid if the type environment is valid. The [Type Object] rule ensures that object types are well-formed by ensuring that the type of each method in the object is well formed.

[Val x]

$$\frac{E',\ x : A, E'' \vdash \diamond}{E',\ x : A, E'' \vdash x : A}$$

[Val Object] (where $A = [l_i : B_i \ ^{i \in 1..n}]$ and l_i are distinct)

$$\frac{E,\ x_i : A \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash [l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}] : A}$$

[Val Select] (where $j \in 1..n$)

$$\frac{E \vdash a : [l_i : B_i \ ^{i \in 1..n}]}{E \vdash a.l_j : B_j}$$

[Val Update] (where $A = [l_i : B_i \ ^{i \in 1..n}]$ and $j \in 1..n$)

$$\frac{E \vdash a : A \quad E,\ x : A \vdash b : B_j}{E \vdash a.l_j \Leftarrow \varsigma(x) b : A}$$

[Val Clone]

$$\frac{E \vdash a : A}{E \vdash \mathit{clone}(a) : A}$$

[Val Let]

$$\frac{E \vdash a : A \quad E, x : A \vdash b : B}{E \vdash \mathit{let}\ x = a\ \mathit{in}\ b : B}$$

Figure 4.12: $\varrho\varpi\varsigma$-calculus Object Typing Rules [AC96, Ch. 11]

4.3.2 *Object Typing*

The rules for judging the types of the original **imp**ς-calculus expressions [AC96, Ch. 11] are listed in Figure 4.12. Despite the changes and additions made to the syntax and semantics of the **imp**ς-calculus, the type rules for the original **imp**ς-calculus remain unchanged from their presentation in Abadi and Cardelli; supporting the claim that the extension to the **imp**ς-calculus is a conservative extension.

We now explain each of the rules from Figure 4.12. To give a type to a variable, the [Val x] rule matches a variable, x, with an entry in the type environment. The type for the entry in the environment then becomes the type for the expression, x. The [Val Object] rule judges object expressions as having object types such that the type follows the structure of the object expression. Method selection expressions are handled by the [Val Select] rule. The rule's hypothesis determines that the selection target is an object such that the method (l_j in the figure) is in the object, and that the type for l_j is the result type of the selection. The [Val Update] rule ensures that the target of the update (a in the figure) has an object type that has an l_j member. The rule further ensures that the new method will have the same type as the previous method. The [Val Clone] rule judges clone expressions to have the same type as the object they are cloning. Finally, the [Val Let] rule judges the let expression as having the result type of the body with respect to the judged type of the definition expression for the new variable, x.

4.3.3 *Subtypes*

The subtyping rules, Figure 4.13, specify how types can be valid subtypes of other types. The [Sub Refl] and [Sub Trans] rules make subtyping reflexive and transitive. The [Sub Top] rule simply states that every valid type is a subtype of type *Top*. For object subtyping, the [Sub Object] rule requires that the subtype must have at least the same method names with the same return values as the supertype and all the method types of the subtype must be valid. The subtyping feature is "applied" in the judgments via the [Val Subsumption] rule. The rule allows an expression to be judged as a different type if the new type is a supertype of its current type. For example, we could have the following judgment (assuming that a was given type A and b was given type B):

[Sub Refl]
$$\frac{E \vdash A}{E \vdash A <: A}$$

[Sub Trans]
$$\frac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C}$$

[Sub Top]
$$\frac{E \vdash A}{E \vdash A <: Top}$$

[Sub Object]
$$\frac{E \vdash B_i \quad \forall i \in 1..n+m}{E \vdash [l_i : B_i \ ^{i \in 1..n+m}] <: [l_i : B_i \ ^{i \in 1..n}]}$$

[Val Subsumption]
$$\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$$

Figure 4.13: $\varrho\varpi\varsigma$-calculus Subtyping Rules [AC96, Ch. 11]

$$\frac{a : A,\ b : B \vdash [\, l = \varsigma(x)a,\ l_2 = \varsigma(y)b \,] : [\, l : A,\ l_2 : B \,] \quad [\, l : A,\ l_2 : B \,] <: [\, l : A \,]}{a : A,\ b : B \vdash [\, l = \varsigma(x)a,\ l_2 = \varsigma(y)b \,] : [\, l : A \,]}$$

The purpose of subsumption is to weaken the type system to allow more expressions to be typed while maintaining the soundness property. Take, for example, the object, $obj \triangleq [\, l = \varsigma(x)[\, n_1 = \varsigma(y)a \,]\,]$, where $a : A$. The type judgment for this object will be, $[\, l : [\, n_1 : A \,]\,]$. Further, assume that we want to evaluate the expression, $obj.l.n_1$. In this case, we could evaluate this expression successfully since $obj.l$ returns an object with method n_1 defined. Now, we update l via the following expression: $obj.l \Leftarrow \varsigma(x)[\, n_1 = \varsigma(y)a,\ n_2 = \varsigma(z)b \,]$. Without subsumption, the type system would not allow such an update, since the body of the update has a different type from the l method. However, the subsumption rules would allow the update since the body of the update is a structural subtype of l's type, i.e., $[\, n_1 : A,\ n_2 : B \,] <: [\, n_1 : A \,]$. In fact, we can still evaluate the expression $obj.l.n_1$ without any problems, despite the difference in types.

4.3.4 *Event Typing*

The rules for the expressions that deal with events, registrations and publications, given in Figure 4.14, are straightforward. The [Type Reg] and [Val Register] rules deal with the registry type $\mathcal{R}$. Recall that $\mathcal{R}$ has no internal structure, but simply marks the result of a registration. The [Type Reg] rule states that $\mathcal{R}$ is a valid type if the type environment is well-formed. The results of registrations are typed according to the [Val Register] rule as a registry type, $\mathcal{R}$. The [Val Register] rule also ensures that the registration target (a in the figure) is an object and that the announcement conditions (b and c in the figure) and the handler (d in the figure) are all expressions that can be given a valid type, thus having a type that is a subtype of *Top*. The rule can ignore the return type of the announcement conditions since their results will only be used for value equality comparison, which is always type safe. Further, the result type for the handler can be ignored since the value of its result is ignored in the semantics. Since the type of these results are ignored, the only type information necessary for the registration values is that they are registration values, which is denoted with the $\mathcal{R}$ type.

It could be considered awkward to use a special set of values for the results of registrations since it is possible to use object values. However, having registration values as non-object values, and thus distinct types prevents use of registration values as objects and vice versa. While doing such a thing is not necessarily invalid or harmful, the meaning of doing such a thing could be unclear. Having this distinction makes the semantics and type system more clear.

The rest of the rules, [Val Unregister], [Val Publish], and [Val Unpublish] are computed for their side-effects,

[Type Reg]
$$\frac{E \vdash \diamond}{E \vdash \mathcal{R}}$$

[Val Register] (where $j \in 1..n$)
$$\frac{E \vdash a : [l_i : B_i \ ^{i \in 1..n}] \quad E \vdash b : Top \quad E \vdash c : Top \quad E \vdash d : Top}{E \vdash \varrho(a.l_j, b \equiv c)d : \mathcal{R}}$$

[Val Unregister]
$$\frac{E \vdash a : \mathcal{R}}{E \vdash \not{\varrho}(a) : [\,]}$$

[Val Publish] (where $j \in 1..n$)
$$\frac{E \vdash a : [l_i : B_i \ ^{i \in 1..n}]}{E \vdash \varpi(a.l_j) : [\,]}$$

[Val Unpublish] (where $j \in 1..n$)
$$\frac{E \vdash a : [l_i : B_i \ ^{i \in 1..n}]}{E \vdash \not{\varpi}(a.l_j) : [\,]}$$

Figure 4.14: $\varrho\varpi\varsigma$-calculus Event Typing Rules

therefore their results are empty objects and their types are empty object types. The [Val Unregister] rule ensures that its unregistration target (a in the figure) is a result of a previous registration. The [Val Publish] and [Val Unpublish] rules ensure that their targets (a in the figure) are all objects that include the specified event (l_j in the figure) as part of their type.

4.3.5 *Conclusion*

The $\varrho\varpi\varsigma$-calculus type system provides rules for determining the type of any $\varrho\varpi\varsigma$-calculus. In the next section, we show that these rules are consistent with the operational semantics presented in the previous section by giving a proof of subject reduction for our semantics and the type system. That is, a proof of subject reduction will show that our type system for the $\varrho\varpi\varsigma$-calculus is sound with respect to its semantics [AC96]. A sound type system is useful because it prevents any "bad" expressions from having types, that is, it prevents expressions that, when evaluated, would get "stuck" in the evaluation. For example, $[\,l_1 = \varsigma(x)[\,]\,].l_2$ could not be given a type with our type system since we are attempting to access a method, l_2 in an object where l_2 does not exist.

Next, we prove that the type system is indeed sound by showing that the syntax, semantics, and type system of the $\varrho\varpi\varsigma$-calculus has the subject reduction property, that is, when a type judgment is given for an expression it is at least a super type of the type of the result of evaluating the expression.

4.4 Subject Reduction

4.4.1 Type System

Before the actual proof of subject reduction is given for the $\varrho\varpi\varsigma$-calculus, we must give some preliminary definitions and lemmas. In order to prove that the types given to $\varrho\varpi\varsigma$-calculus expressions by the type rules are correct, there must be a way to type results. According to Abadi and Cardelli [AC96, p. 146],

> "Unfortunately, the typing of results is delicate. We would not be able to determine the type of a result by examining its substructures recursively, including the ones accessed through the store, because stores may contain loops. Store types...allow us to type results independently of particular stores. This is possible because type-sound computations do not store results of different types in the same location."

Thus, in order to define types for results, we also provide type judgments for the store itself. Figure 4.15 lists the syntax for store typing and Figure 4.16 gives the rules for the store typing. The syntax and the rules for

$M ::= [l_i : B_i \ ^{i\in 1..n}] \Rightarrow B_j \ (j \in 1..n)$	method type
$\Sigma ::= \iota_i \mapsto M_i \ ^{i\in 1..n}$	store type (ι_i distinct)
$\Sigma_1(\iota) \triangleq [l_i : B_i \ ^{i\in 1..n}]$	if $\Sigma(\iota) = [l_i : B_i \ ^{i\in 1..n}] \Rightarrow B_j$
$\Sigma_2(\iota) \triangleq B_j$	if $\Sigma(\iota) = [l_i : B_i \ ^{i\in 1..n}] \Rightarrow B_j$
$\models M \in Meth$	well-formed method type judgment
$\Sigma \models \diamond$	well-formed store type judgment
$\Sigma \models v \ : \ A$	result typing judgment
$\Sigma \models S \ : \ E$	stack typing judgment
$\Sigma \models \sigma$	store typing judgment
$\Sigma \models R$	registry typing judgment

Figure 4.15: $\varrho\varpi\varsigma$-calculus Context Typing Syntax

store typing depend on having valid stack types, i.e., given a stack, the corresponding type for the stack would be a matching typing environment. Definition 4.4.1 gives a formal definition of a stack type.

Definition 4.4.1 (Stack Types). A type environment, E, is a type for a stack, S, i.e., $S : E$, with respect to a store type Σ, i.e., $\Sigma \models S : E$ if and only if, either both E and S are empty (ϕ), or S is $(S', x \mapsto v)$, E is $(E', x : A)$, and $\Sigma \models S' : E'$, $\Sigma \models v : A$, $x \notin dom(S')$, and $x \notin dom(E')$.

The environment typing syntax, Figure 4.15, details the notation for the environment typing and the structure of the judgments that the environment typing rules, Figure 4.16, can perform. The method type notation, definition of M, denotes the type of a store location. Each store location has two relevant types: the type of the method's object, i.e., the type of the self parameter, and result type of the method, B_j. Thus, method types are defined to have both an object type and a type for the result of a method from that object, using the arrow ($\Rightarrow$) notation to show the relationship between these two types. In order to use both of these parts of the method type, the Σ_1 and Σ_2 functions are defined that return the object type and the result type of the method respectively. The $\models M \in Meth$ judgment specifies that M is a valid member of the set *Meth*, where *Meth* is the set of all valid method types. The rest of the judgment forms will be explained along with the rule they are used for.

To ensure that the environment typing is valid for a given system, we define a set of rules that validates a store type Σ for a given value, stack, registry, or store. The [Method Type] axiom states that a method type is valid if it is a member of the set of all valid method types, *Meth*. The store type, Σ, is judged to be well-formed by the [Store Type] rule if all of its member method types are members of *Meth*. Object values are given a type based on the store type environment via the [Result Object] rule such that the object type (Σ_1) is valid for each member of the object value. The [Result Reg] rule gives the registry type, $\mathcal{R}$, to registry values. The [Stack ϕ Typing] and [Stack x Typing] rules ensure that the stack type (E in the figure) is a correct typing of its related stack (S in the figure). The [Store Typing] rule ensures that the store type is a correct representation of the store, by checking that each closure in the store has the type that the store type has associated with it. The registry is validated via the [Registry <> Typing], [Registry Closure Typing], and [Registry Typing] rules. The [Registry Typing] rule is used to iterate through the registration context, allowing the appropriate rule for each entry to be applied for its hypothesis. The [Registry <> Typing] rule ensures that entries in the registration context that have been unregistered are well-formed. Finally, the [Registry Closure Typing] rule ensures that registry closures can be properly typed by ensuring that the announcement conditions (b and c in the figure) and the handler (d in the figure) can be judged to type *Top* (presumably via subsumption), in order to ensure that there are no type errors in those expressions. The rule further ensures that the event closures are well-formed by making sure the stack (S in the figure) has a valid stack type.

4.4.2 Subject Reduction Proof

It is often the case in the course of the subject reduction proof that the store will be expanded to accommodate a new object. To denote that the store type associated with the expanded store was only changed from the original store type to handle the store expansion, we will use, $\preceq$. The formal definition for $\preceq$ is given below.

[Method Type] (where $j \in 1..n$)

$$\frac{}{\models [l_i : B_i \ ^{i \in 1..n}] \Rightarrow B_j \in Meth}$$

[Store Type] (where ι_i distinct)

$$\frac{\models M_i \in Meth \quad \forall i \in 1..n}{\iota_i \mapsto M_i \ ^{i \in 1..n} \models \diamond}$$

[Result Object]

$$\frac{\Sigma \models \diamond \quad \Sigma_1(\iota_i) = [l_i : \Sigma_2(\iota_i)^{i \in 1..n}] \quad \forall i \in 1..n}{\Sigma \models [l_i = \iota_i \ ^{i \in 1..n}] : [l_i : \Sigma_2(\iota_i)^{i \in 1..n}]}$$

[Result Reg]

$$\frac{\Sigma \models \diamond}{\Sigma \models (\iota,\ I) : \mathcal{R}}$$

[Stack ϕ Typing]

$$\frac{\Sigma \models \diamond}{\Sigma \models \phi : \phi}$$

[Stack x Typing] (where $x \notin dom(E)$)

$$\frac{\Sigma \models S : E \quad \Sigma \models v : A}{\Sigma \models (S, x \mapsto v) : (E, x : A)}$$

[Store Typing]

$$\frac{\Sigma \models S_i : E_i \quad E_i, x_i : \Sigma_1(\iota_i) \vdash b_i : \Sigma_2(\iota_i) \quad \forall i \in 1..n}{\Sigma \models \iota_i \mapsto < \varsigma(x_i) b_i, S >^{i \in 1..n}}$$

[Registry <> Typing]

$$\frac{\Sigma \models \diamond}{\Sigma \models (\iota, I) \mapsto <>}$$

[Registry Closure Typing]

$$\frac{\Sigma \models S : E \quad E \vdash b : Top \quad E \vdash c : Top \quad E \vdash d : Top}{\Sigma \models (\iota_i, I_j) \mapsto < b, c, d, S >}$$

[Registry Typing]

$$\frac{\Sigma \models (\iota_i,\ I_j) \mapsto t_{(i,j)} \quad \forall i \in 1..n \quad \forall j \in 1..m_i}{\Sigma \models (\iota_i,\ I_j) \mapsto t_{(i,j)}^{i \in 1..n,\ j \in 1..m_i}}$$

Figure 4.16: $\varrho\varpi\varsigma$-calculus Context Typing Rules

Definition 4.4.2 ($\preceq$). The relation, $\preceq$, is defined for all store typing environments, Σ and Σ', store location ι, and registry lookup key $(\iota,\ I)$ such that $\Sigma \preceq \Sigma' \Leftrightarrow dom(\Sigma) \subseteq dom(\Sigma')$, and, $\forall \iota \in dom(\Sigma),\ \Sigma(\iota) = \Sigma'(\iota)$

Before we actually present the proof of subject reduction, we give two lemmas and a corollary that will prove general properties about the $\varrho\varpi\varsigma$-calculus formalism that will be useful for the subject reduction proof. Lemma 4.4.3 shows that when a stack and its type are judged to be well-formed for a given store type, the stack-type pair will be valid for extended versions of that store type. Corollary 4.4.4 states that values under a given store type environment have the same type under a different environment that is an extension ($\preceq$) of the original. Finally, Lemma 4.4.5 shows that when type environments validate a judgment, the types within the type environment may be weakened, i.e., changed to subtypes, and the judgments will still remain valid.

Lemma 4.4.3 (Stack Typing Preserved With Extension [AC96, Ch. 11]). For any store typing environments Σ and Σ', stack S, and type environment E:
If $\Sigma \models S : E$ and $\Sigma \models \diamond$ with $\Sigma \preceq \Sigma'$, then $\Sigma' \models S : E$.

Proof

Assume $\Sigma \models S : E$ where Σ is well formed and $\Sigma \preceq \Sigma'$. By the definition of $S : E$ (Definition 4.4.1), $\Sigma \models v : A, \forall (x \mapsto v) \in S$ such that $(x : A) \in E$.

Now, by the definition of $\preceq$ (Definition 4.4.2), $dom(\Sigma) \subseteq dom(\Sigma')$, and $\forall \iota \in dom(\Sigma), \Sigma(\iota) = \Sigma'(\iota)$. Thus, $\Sigma' \models v : A, \forall (x \mapsto v) \in S$ such that $(x : A) \in E$, which implies, by definition of $S : E$ that $\Sigma' \models S : E$. $\square$

Corollary 4.4.4. For any store typing environments Σ and Σ', value v, and type A:
If $\Sigma \models v : A$ and $\Sigma \models \diamond$ with $\Sigma \preceq \Sigma'$, then $\Sigma' \models v : A$.

Lemma 4.4.5 (Bound Weakening [AC96, Ch. 11]). For any type judgment $\Im$, type environments E and E', and types D and D':
If $E,\ x : D,\ E' \vdash \Im$ and $E \vdash D' <: D$, then $E,\ x : D',\ E' \vdash \Im$.

Proof

Assume the hypothesis. Since $E,\ x : D,\ E' \vdash \Im$, then either one of two possibilities is true, the type judgment, $\Im$, requires $x : D$ in the type environment in order for its judgment to be satisfied, or it does not.

First, assume that $\Im$ does not require $x : D$ in the type environment. Then, since, by assumption, $E \vdash D' <: D$, the type environment, $E, x : D', E'$ will be well-formed, thus, $E, x : D', E' \vdash \Im$.

Now, assume that $\Im$ does require $x : D$ in the type environment. This implies that x is referenced in the expression of $\Im$, which will require the application of the [Val x] rule for the judgment, Figure 4.12 (there is no other rule that makes use of the type environment to validate a type judgment). Specifically, the [Val x] rule application would be (except, the type environment could be larger at this point, but still well-formed since $\Im$ is assumed to be a valid type judgment):

$$\frac{E, x : D, E' \vdash \diamond}{E, x : D, E' \vdash x : D}\ [\text{Val x}]$$

Now, replace D with D' in the type environment. The proof tree for the same application of [Val x] would appear as follows:

$$\frac{\dfrac{E, x : D', E' \vdash \diamond}{E, x : D', E' \vdash x : D'}\ [\text{Val x}] \quad \dfrac{E \vdash D' <: D}{E, x : D', E' \vdash D' <: D}\ [\text{By hypothesis}]}{E, x : D', E' \vdash x : D}\ [\text{Val Subsumption}]$$

Thus, $E, x : D', E' \vdash \Im$. $\square$

The idea for the proof of subject reduction is that the type of a $\varrho\varpi\varsigma$-expression given by the type system should be a supertype of the type of the result of the expression determined by the environment typing rules via a valid store type. Thus, since the subject reduction theorem holds, we have that a valid typing for an expression

(according to the type system) is an accurate typing of the results of computing the expression. For the proof, we perform a simultaneous induction over the semantic judgments of the $\varrho\varpi\varsigma$-calculus and the subsystem judgments for the event announcement.

Theorem 4.4.6 (Subject Reduction [AC96]). For any expression α, subsystem sets δ and δ', types A and A', value v, type environment E, stores σ and σ', registry contexts R, R' and R_0, publication contexts P and P', stack S, and store typing environments Σ and Σ':

If $E \vdash \alpha : A$, $\sigma \cdot R \cdot P \cdot S \vdash \alpha \rightsquigarrow v \cdot \sigma' \cdot R' \cdot P'$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S : E$, then there exists a type A' and a store typing environment, Σ', such that $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $dom(\sigma') = dom(\Sigma')$, $\Sigma' \models v : A'$, and $\vdash A' <: A$.

If $R_0 \cdot \sigma \cdot R \cdot P \vdash \delta \rightharpoonup \delta' \cdot \sigma' \cdot R' \cdot P'$, $\Sigma \models \sigma$, $\Sigma \models R$, $\Sigma_0 \models R_0$, $dom(\sigma) = dom(\Sigma)$, and $\forall e \in \delta.\exists(\iota, I).(\iota, I) \mapsto e \in R_0$, then there exists a store typing environment, Σ', such that $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, and $dom(\sigma') = dom(\Sigma')$.

If $R_0 \cdot \sigma \cdot R \cdot P \vdash \delta \rightharpoondown \sigma' \cdot R' \cdot P'$, $\Sigma \models \sigma$, $\Sigma \models R$, $\Sigma_0 \models R_0$, $dom(\sigma) = dom(\Sigma)$, and $\forall e \in \delta.\exists(\iota, I).(\iota, I) \mapsto e \in R_0$, then there exists a store typing environment, Σ', such that $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, and $dom(\sigma') = dom(\Sigma')$.

Proof

By induction on the derivations of the following judgments:

$\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow v \cdot \sigma' \cdot R' \cdot P'$,

$R_0 \cdot \sigma \cdot R \cdot P \vdash D \rightharpoondown \sigma' \cdot R' \cdot P'$, and

$R_0 \cdot \sigma \cdot R \cdot P \vdash EVT \rightharpoonup D \cdot \sigma' \cdot R' \cdot P'$.

Assume the hypothesis.

Base Cases

Case (Red x [AC96, Ch. 11])

Suppose the expression, α, is x and the last step of the derivation is the [Red x] rule, Figure 4.6.

By hypothesis, $E \vdash x : A$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S : E$.

Now, by the [Val x] typing rule, Figure 4.12, and bound weakening, Lemma 4.4.5, $E = E', x : A', E''$ for $\vdash A' <: A$, thus $\vdash A' <: A$. Then, $\Sigma \models (S', x \mapsto v, S'') : E$ is derived via several applications of the [Stack Typing] rule. One of these applications gives $\Sigma \models v : A'$.

Therefore, $\Sigma \preceq \Sigma$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, $\Sigma \models v : A'$, and $\vdash A' <: A$.

Case (Event Red Filter Ø)

Suppose the subsystem set, δ, is Ø, the reduction in question is a '$\rightharpoonup$' reduction, and the last step of the derivation is the [Event Red Filter Ø] axiom.

By the [Event Red Filter Ø] axiom, we have $R_0 \cdot \sigma \cdot R \cdot P \vdash \emptyset \rightharpoonup \emptyset \cdot \sigma \cdot R \cdot P$

By the hypothesis, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$. Therefore, since σ, R, and P are left unchanged, the conclusion holds, $\Sigma \preceq \Sigma$, $\Sigma \models \sigma$, $\Sigma \models R$, and $dom(\sigma) = dom(\Sigma)$.

Case (Event Red Handler Ø)

Suppose the subsystem set, δ, is Ø, the reduction in question is a '$\rightharpoondown$' reduction, and the last step of the derivation is the [Event Red Handler Ø] axiom.

By the [Event Red Filter Ø] axiom, we have $R_0 \cdot \sigma \cdot R \cdot P \vdash \emptyset \rightharpoondown \sigma \cdot R \cdot P$

By the hypothesis, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$. Therefore, since σ, R, and P are left unchanged, the conclusion holds, $\Sigma \preceq \Sigma$, $\Sigma \models \sigma$, $\Sigma \models R$, and $dom(\sigma) = dom(\Sigma)$.

Induction Hypotheses

Given an expression, α, a store, σ, a registry context, R, a publication context, P, a stack, S, a type environment, E, a store typing environment, Σ, and a type A, and assuming the hypothesis of this theorem for an expression reduction '$\rightsquigarrow$', the reduction hypotheses for the expression reduction are assumed to satisfy the subject reduction theorem.

Given a subsystem set, δ, a store, σ, registry contexts R and R_0, a publication context, P, a store typing environment, Σ, and assuming the hypothesis of this theorem for a '$\rightharpoonup$' subsystem reduction, the reduction hypotheses for the subsystem reduction are assumed to satisfy the subject reduction theorem.

Given a subsystem set, δ, a store, σ, registry contexts R and R_0, a publication context, P, a store typing environment, Σ, and assuming the hypothesis of this theorem for a '$\rightharpoondown$' subsystem reduction, the reduction hypotheses for the subsystem reduction are assumed to satisfy the subject reduction theorem.

Induction Cases

Case (Red Object [AC96, Ch. 11])

Suppose the expression, α, is $[l_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$ and the last step of the derivation is the [Red Object] rule from Figure 4.6.

By hypothesis, $E \vdash [l_i = \varsigma(x_i)b_i \ ^{i\in 1..n}] : A$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S : E$.

Since $E \vdash [l_i = \varsigma(x_i)b_i \ ^{i\in 1..n}] : A$, we must have, by the [Val Object] typing rule, $E \vdash [l_i = \varsigma(x_i)b_i \ ^{i\in 1..n}] : [l_i : B_i \ ^{i\in 1..n}]$, for some $[l_i : B_i \ ^{i\in 1..n}] <: A$, such that $\vdash [l_i : B_i \ ^{i\in 1..n}] <: A$. So, let $A' = [l_i : B_i \ ^{i\in 1..n}]$.

Now, let $\Sigma' = \Sigma,\ \iota_j \mapsto (A' \Rightarrow B_j)^{j\in 1..n}$. Since, by the [Red Object] rule, $\iota_j \notin dom(\sigma)$, $\iota_j \notin dom(\Sigma)$. Also, $\models A' \Rightarrow B_j \in Math$, for $j \in 1..n$. Thus, by the [Store Type] rule, Figure 4.16, $\Sigma' \models \diamond$.

(1) Since $\Sigma \preceq \Sigma'$, by Lemma 4.4.3, $\Sigma' \models S : E$. Also, since, $E \vdash [l_i = \varsigma(x_i)b_i \ ^{i\in 1..n}] : A'$, we have $E,\ x_i : A' \vdash b_i : B_i$, which is equivalent to, $E,\ x_i : \Sigma'_1(\iota_i) \vdash b_i : \Sigma'_2(\iota_i)$.

(2) By definition, we have that σ is of the form $\varepsilon_k \mapsto\!< \varsigma(x_k)b_k, S_k >^{k\in 1..m}$. Now, by the [Store Typing] rule, Figure 4.16, with $\Sigma \models S_k : E_k$ and $E_k,\ x_k : \Sigma_1(\varepsilon_k) \vdash b_k : \Sigma_2(\varepsilon_k)$, we have $\Sigma \models \sigma$. By Lemma 4.4.3, $\Sigma' \models S_k : E_k$; moreover, since $dom(\sigma) = dom(\Sigma) = \{\varepsilon_k \ ^{k\in 1..m}\}$ and $\Sigma \preceq \Sigma'$, $\Sigma'(\varepsilon_k) = \Sigma(\varepsilon_k)$ for $k \in 1..m$. Thus, $E_k,\ x_k : \Sigma'_1(\varepsilon_k) \vdash b_k : \Sigma'_2(\varepsilon_k)$.

Now, by (1) and (2), and the [Store Typing] rule, we have $\Sigma' \models (\sigma,\ \iota_i \mapsto\!< \varsigma(x_i)b_i, S >^{i\in 1..n})$. Since $\Sigma' \models \diamond$ and $\Sigma' = \Sigma,\ \iota_j \mapsto (A' \Rightarrow B_j)^{j\in 1..n}$, we have, by the conclusion of the [Store Typing] rule, $\Sigma' \models [l_i = \iota_i \ ^{i\in 1..n}] : A'$. Therefore, $\Sigma \preceq \Sigma'$, $\Sigma' \models (\sigma,\ \iota_i \mapsto\!< \varsigma(x_i)b_i, S >^{i\in 1..n})$, $\Sigma' \models R$, $dom((\sigma,\ \iota_i \mapsto\!< \varsigma(x_i)b_i, S >^{i\in 1..n})) = dom(\Sigma')$, $\Sigma' \models [l_i = \iota_i \ ^{i\in 1..n}] : A'$, and $\vdash A' <: A$.

Case (Red Select [AC96, Ch. 11])

Suppose the expression, α, is $a.l_j$ and the last step of the derivation is the [Red Select] rule, Figure 4.6.

By hypothesis, $E \vdash a.l_j : A$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, $\Sigma \models S : E$.

Since, $E \vdash a.l_j : A$, we must have, by the [Val Object] rule, $E \vdash a : [l_j : B_j, \ldots]$, such that $\vdash B_j <: A$.

By induction hypothesis, since $E \vdash a : [l_j : B_j, \ldots]$, $\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow [l_i = \iota_i \ ^{i\in 1..n}] \cdot \sigma'' \cdot R'' \cdot P''$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S : E$, there exists a type A'' and a store typing environment Σ'' such that, $\Sigma \preceq \Sigma''$, $\Sigma'' \models \sigma''$, $\Sigma'' \models R''$, $dom(\sigma'') = dom(\Sigma'')$, $\Sigma'' \models [l_i = \iota_i \ ^{i\in 1..n}] : A''$, and $\vdash A'' <: [l_j : B_j, \ldots]$.

Since $\sigma''(\iota_j) =\!< \varsigma(x_j)b_j, S' >$, the judgment, $\Sigma'' \models \sigma''$, must come via the [Store Typing] rule from $\Sigma'' \models S' : E_j$ and $E_j,\ x_j : \Sigma''_1(\iota_j) \vdash b_j : \Sigma''_2(\iota_j)$, for some E_j. Since $\Sigma'' \models [l_i = \iota_i \ ^{i\in 1..n}] : A''$ must come from the [Result Object] rule, Figure 4.16, we have $A'' = [l_i : \Sigma''_2(\iota_i)^{i\in 1..n}] = \Sigma''_1(\iota_j)$. Since $\vdash A'' <: [l_j : B_j, \ldots]$, we have, $\Sigma''_2(\iota_j) = B_j$. Then, from $E_j,\ x_j : \Sigma''_1(\iota_j) \vdash b_j : \Sigma''_2(\iota_j)$, we obtain $E_j, x_j : A'' \vdash b_j : B_j$. Moreover, by the [Stack x Typing] rule, Figure 4.16, we get $\Sigma'' \models S',\ x_j \mapsto [l_i = \iota_i \ ^{i\in 1..n}] : E_j,\ x_j : A''$.

Let $E'' = E_j,\ x_j : A''$. By induction hypothesis, since
$E'' \vdash b_j : B_j,\ \sigma'' \cdot R'' \cdot P'' \cdot (S',\ x_j \mapsto [l_i = \iota_i \ ^{i\in 1..n}]) \vdash b_j \rightsquigarrow v \cdot \sigma' \cdot R' \cdot P'$, $\Sigma'' \models \sigma''$, $\Sigma'' \models R''$, $dom(\sigma'') = dom(\Sigma'')$, and $\Sigma'' \models S', x_j \mapsto [l_i = \iota_i \ ^{i\in 1..n}] : E''$, there exists a type A' and a store typing environment Σ' such that $\Sigma'' \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $dom(\sigma') = dom(\Sigma')$, $\Sigma' \models v : A'$, and $\vdash A' <: B_j$.

Also, by transitivity of $\preceq$, $\Sigma \preceq \Sigma'$ and by transitivity of $<:$, $A' <: A$.

Therefore, $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $dom(\sigma') = dom(\Sigma')$, $\Sigma' \models v : A'$, and $\vdash A' <: A$.

Case (Red Update [AC96, Ch. 11])

Suppose the expression, a, is $a.l_j \Leftarrow \varsigma(x)b$ and $a.l_j \notin P$ and the last step of the derivation is the [Red Update] rule, Figure 4.6.

By hypothesis, $E \vdash a.l_j \Leftarrow \varsigma(x)b : A$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S : E$.

Since, $E \vdash a.l_j \Leftarrow \varsigma(x)b : A$, we must have, by the [Val Update] typing rule, Figure 4.12, $E \vdash a : [l_j : B_j, \ldots]$, and $E,\ x : [l_j : B_j, \ldots] \vdash b : B_j$ for some $[l_j : B_j, \ldots]$ such that $\vdash [l_j : B_j, \ldots] <: A$.

By induction hypothesis, since $E \vdash a : [l_j : B_j, \ldots]$, $\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow [l_i = \iota_i \ ^{i\in 1..n}] \cdot \sigma' \cdot R' \cdot P'$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S : E$, there exists a type A' and a store typing environment Σ' such that,

$\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $dom(\sigma') = dom(\Sigma')$, $\Sigma' \models [l_i = \iota_i \ ^{i\in 1..n}] : A'$, and $\vdash A' <: [l_j : B_j, \ldots]$.

By assumption, $\iota_j \in dom(\sigma')$, hence $\iota_j \in dom(\Sigma')$. But, $\Sigma' \models [l_i = \iota_i \ ^{i\in 1..n}] : A'$ must have been derived via the [Result Object] rule, from $\Sigma'_1(\iota_i) = [l_i : \Sigma'_2(\iota_i)^{i\in 1..n} = A']$ for all $i \in 1..n$. Hence, since $\vdash A' <: [l_j : B_j, \ldots]$, we have, $\Sigma'_2(\iota_j) = B_j$.

(1) By lemma 4.4.3, we have $\Sigma' \models S : E$. Also, by lemma 4.4.5, we have $E,\ x : A' \vdash B_j$, from $E,\ x : [l_j : B_j, \ldots] \vdash b : B_j$ and $A' <: [l_j : B_j, \ldots]$. So, $E,\ x : \Sigma'_1(\iota_j) \vdash b : \Sigma'_2(\iota_j)$.

(2) Since, by the [Store Typing] rule, $\Sigma' \models \sigma'$, σ' is of the form $\varepsilon_k \mapsto < \varsigma(x_k)b_k, S_k >^{k\in 1..m}$, and for all k such that $\varepsilon_k \neq \iota j$ and for some E_k we have $\Sigma' \models S_k : E_k$, and $E_k,\ x_k : \Sigma'_1(\varepsilon_k) \vdash b_k : \Sigma'_2(\varepsilon_k)$.

Now, by (1) and (2) and the [Store Typing] rule, we have $\Sigma' \models (\sigma'.\iota_j \leftarrow < \varsigma(x)b, S >)$. Also, since $dom((\sigma'.\iota_j \leftarrow < \varsigma(x)b, S >)) = dom((\sigma'))$, $dom((\sigma'.\iota_j \leftarrow < \varsigma(x)b, S >)) = dom(\Sigma')$. Further, by transitivity of $<:$, $\vdash A' <: A$.

Therefore, $\Sigma \preceq \Sigma'$, $\Sigma' \models (\sigma'.\iota_j \leftarrow < \varsigma(x)b, S >)$, $\Sigma' \models R'$, $dom((\sigma'.\iota_j \leftarrow < \varsigma(x)b, S >)) = dom(\Sigma')$, $\Sigma' \models [l_i = \iota_i \ ^{i\in 1..n}] : A'$, and $\vdash A' <: A$.

Case (Red Clone [AC96, Ch. 11])

Suppose the expression, α, is $clone(a)$ and the last step of the derivation is the [Red Clone] rule, Figure 4.6.

By hypothesis, $E \vdash clone(a) : A$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, $\Sigma \models S : E$.

Since $E \vdash clone(a) : A$, we must have, by the [Val Clone] rule, Figure 4.12, $E \vdash a : A$. By induction hypothesis, since $E \vdash a : A$, $\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow [l_i = \iota_i \ ^{i\in 1..n}] \cdot \sigma' \cdot R' \cdot P'$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S : E$, there exists a type A' and a store typing environment Σ' such that, $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $dom(\sigma') = dom(\Sigma')$, $\Sigma' \models [l_i = \iota_i \ ^{i\in 1..n}] : A'$, and $\vdash A' <: A$.

Let $\Sigma^\dagger = (\Sigma',\ \iota'_i \mapsto \Sigma'(\iota_i)^{i\in 1..n})$ and $\sigma^\dagger = (\sigma',\ \iota'_i \mapsto \sigma'(\iota_i)^{i\in 1..n})$.

By the [Store Type] rule, we have, $\Sigma^\dagger \models \diamond$, because $\iota'_i \notin dom(\sigma') = dom(\Sigma')$, ι'_i are all distinct for $i \in 1..n$, and $\Sigma' \models \diamond$ is a prerequisite of $\Sigma' \models \sigma'$.

Now, since, by the [Store Type] rule, $\Sigma' \models \sigma'$, σ' has the shape $\varepsilon_k \mapsto < \varsigma(x_k)b_k, S_k >^{k\in 1..m}$, and for all $k \in 1..m$, for some E_k we have $\Sigma' \models S_k : E_k$ and $E_k, x_k : \Sigma'_1(\varepsilon_k) \vdash b_k : \Sigma'_2(\varepsilon_k)$.

Then, we also have $E_k,\ x_k : \Sigma^\dagger_1(\varepsilon_k) \vdash b_k : \Sigma^\dagger_2(\varepsilon_k)$, and by Lemma 4.4.3, $\Sigma^\dagger \models S_k : E_k$. Let $f : 1..n \to 1..m$ be such that $\iota_i = \varepsilon_{f(i)}$, for all $i \in 1..n$. The function f exists because the locations in the store are distinct. We have $E_{f(i)},\ x_{f(i)} : \Sigma'_1(\varepsilon_{f(i)}) \vdash \Sigma'_2(\varepsilon_{f(i)})$ for $i \in 1..n$, so $E_{f(i)},\ x_{f(i)} : \Sigma'_1(\iota_i) \vdash b_{f(i)} : \Sigma'_2(\iota_i)$. Moreover, since $\Sigma'(\iota_i) = \Sigma^\dagger(\iota'_i)$, we have $E_{f(i)},\ x_{f(i)} : \Sigma^\dagger_1(\iota'_i) \vdash b_{f(i)} : \Sigma'_2(\iota'_i)$. Then, by the [Store Typing] rule, since for $k \in 1..m$ and $i \in 1..n$, $\Sigma^\dagger \models S_k : E_k$, $\Sigma^\dagger \models S_{f(i)} : E_{f(i)}$, $E_k,\ x_k : \Sigma^\dagger_1(\varepsilon_k) \vdash b_k : \Sigma^\dagger_2(\varepsilon_k)$, $E_{f(i)},\ x_{f(i)} : \Sigma^\dagger_1(\iota'_i) \vdash b_{f(i)} : \Sigma^\dagger_2(\iota'_i)$, and $\Sigma^\dagger \models \sigma^\dagger$.

Further, $\Sigma' \models [l_i = \iota_i \ ^{i\in 1..n}] : A'$ must come from the [Result Object] rule, Figure 4.16 with $A' = \Sigma'_1(\iota_i) = [l_i : \Sigma'_2(\iota_i)^{i\in 1..n}]$, and $\Sigma' \models \diamond$. But, $\Sigma^\dagger(\iota'_i) = \Sigma'(\iota_i)$ for $i \in 1..n$. So, $\Sigma^\dagger_1(\iota'_i) = [l_i : \Sigma^\dagger_2(\iota'_i)^{i\in 1..n}] = A'$, and by the [Result Object] rule, we obtain $\Sigma^\dagger \models [l_i = \iota'^{i\in 1..n}_i] : [l_i : \Sigma^\dagger_2(\iota'_i)^{i\in 1..n}]$. Thus, $\Sigma^\dagger \models [l_i = \iota'_i] : A'$.

Also, by transitivity of $\preceq$, $\Sigma \preceq \Sigma^\dagger$ and $dom(\sigma^\dagger) = dom(\Sigma^\dagger)$, by construction and because $dom(\sigma') = dom(\Sigma')$.

Therefore, $\Sigma \preceq \Sigma^\dagger$, $\Sigma^\dagger \models \sigma^\dagger$, $\Sigma^\dagger \models R'$, $dom(\sigma^\dagger) = dom(\Sigma^\dagger)$, $\Sigma^\dagger \models [l_i = \iota_i \ ^{i\in 1..n}] : A'$, and $\vdash A' <: A$.

Case (Red Let [AC96, Ch. 11])

Suppose the expression, α, is $let\ x = c\ in\ b$ and the last step of the derivation is the [Red Let] rule, Figure 4.6.

By hypothesis, $E \vdash let\ x = c\ in\ b : A$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S : E$. Since, $E \vdash let\ x = c\ in\ b : A$, we must have, by the [Val Let] typing rule, Figure 4.12, $E \vdash c : C$, and $E,\ c : C \vdash b : A$.

By the induction hypothesis, since $E \vdash c : C$, $\sigma \cdot R \cdot P \cdot S \vdash c \rightsquigarrow v' \cdot \sigma' \cdot R' \cdot P'$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, $\Sigma \models S : E$, there exists a type C' and a store typing environment Σ' such that, $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $dom(\sigma') = dom(\Sigma')$, $\Sigma' \models v' : C'$, and $\vdash C' <: C$.

By, Lemma 4.4.3, $\Sigma' \models S : E$, hence by the [Stack x Typing] rule, Figure 4.16, $\Sigma' \models (S,\ x \mapsto v') : (E,\ x : C')$. From $E,\ x : C \vdash b : A$ and Lemma 4.4.5, we obtain $E,\ x : C' \vdash b : A$.

Then, by the induction hypothesis, we have that, since $E,\ x : C' \vdash b : A$, $\sigma' \cdot R' \cdot P' \cdot (S,\ x \mapsto v') \vdash b \rightsquigarrow v'' \cdot \sigma'' \cdot R'' \cdot P''$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $dom(\sigma') = dom(\Sigma')$, $\Sigma' \models (S,\ x \mapsto v') : (E, x : C')$, there exists a type A'', and a store typing environment Σ'' such that $\Sigma' \preceq \Sigma''$, $\Sigma'' \models \sigma''$, $\Sigma'' \models R''$, $dom(\sigma'') = dom(\Sigma'')$, $\Sigma'' \models v'' : A''$, and $\vdash A'' <: A$.

Further, by transitivity of $\preceq$, $\Sigma \preceq \Sigma''$.

Therefore, $\Sigma \preceq \Sigma''$, $\Sigma'' \models \sigma''$, $\Sigma'' \models R''$, $dom(\sigma'') = dom(\Sigma'')$, $\Sigma'' \models v'' : A''$, and $\vdash A'' <: A$.

Case (Red Register)

Suppose the expression, α, is $\varrho(a.l_j, b \equiv c)d$. Then, the last step of the derivation must be the [Red Register] rule, Figure 4.7.

By hypothesis, $E \vdash \varrho(a.l_j, b \equiv c)d : A$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, $\Sigma \models S : E$.

Since $E \vdash \varrho(a.l_j, b \equiv c)d : A$, , we must have, by the [Val Register] typing rule, Figure 4.14, $A = \mathcal{R}$. Also, by [Val Register], $E \vdash a : [l_i : B_i \ ^{i \in 1..n}]$, l_j is a valid label in $[l_i = \iota_i \ ^{i \in 1..n}]$, and b, c, and d all have type *Top*.

By induction hypothesis, since $E \vdash a : [l_i : B_i \ ^{i \in 1..n}]$, $\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow [l_i = \iota_i \ ^{i \in 1..n}] \cdot \sigma_0 \cdot R_0 \cdot P_0$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, $\Sigma \models S : E$, there exists an A_0 and a Σ_0 such that, $\Sigma \preceq \Sigma_0$, $\Sigma_0 \models \sigma_0$, $\Sigma_0 \models R_0$, $dom(\sigma_0) = dom(\Sigma_0)$, $\Sigma_0 \models [l_i = \iota_i \ ^{i \in 1..n}] : A_0$, and $\vdash A_0 <: [l_i : B_i \ ^{i \in 1..n}]$.

Now, let $\Sigma' = \Sigma_0$, $\sigma' = \sigma_0$, $R' = R_0$.

Then, $\Sigma' \models \sigma'$ by [Store Typing] with σ' of the form $\iota_k \mapsto < \varsigma(x_k)b_k, S_k >^{k \in 1..n}$, since $\Sigma' = \Sigma_0$, $\Sigma' \models S_k : E_k$ (by Lemma 4.4.3), and $E_k, x_k : \Sigma_1(\iota_k) \vdash b_k : \Sigma_2(\iota_k)$ for all $k \in 1..n$, which satisfies the [Store Typing] rule.

Furthermore, $\Sigma' \models R'$, $(\iota_j, I) \mapsto < b, c, d, S >$ by [Registry Typing] where R' has the form $(\iota_i, \ I_k) \mapsto t_{(i,k)} \ ^{i \in 1..n, k \in 1..m_i}$. If for all k, $t_{(i,k)} =< b', \ c', \ d', S' >$ then since $\Sigma' = \Sigma_0$, $\Sigma' \models S' : E'$ (Lemma 4.4.3), by the [Registry Closure Typing] for R'. Otherwise, $t_{(i,k)} =<>$ and [Registry <> Typing] is satisfied trivially. Finally, $\Sigma' \models S : E$ (Lemma 4.4.3) and by [Val Register] $E \vdash b : B, \ E \vdash c : C, \ E \vdash d : D$, satisfying [Registry Closure Typing].

So, $\Sigma' \models (\iota_j, \ I) : \mathcal{R}$ holds trivially via the [Result Reg] rule. Further, b, c, and d require type *Top*, which holds from the [Val Register] rule and subsumption.

Therefore, since, $\Sigma \preceq \Sigma'$ by transitivity of $\preceq$, $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $(\iota_j, \ I) \mapsto < b, c, d, S >$, $dom(\sigma') = dom(\Sigma')$, $\Sigma' \models (\iota_j, \ I) : \mathcal{R}$, and $\vdash \mathcal{R} <: \mathcal{R}$, by [Sub Refl].

Case (Red Unregister)

Suppose the expression, α, is $\not\varrho(a)$ and the last step of the derivation is the [Red Unregister] rule, Figure 4.7.

By hypothesis, $E \vdash \not\varrho(a) : A$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S : E$.

By the [Val Unregister] rule, Figure 4.14, A must be [ ] and $E \vdash a : \hat{A}$.

By induction hypothesis, since $E \vdash a : \hat{A}$, $\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow [l_i = \iota_i \ ^{i \in 1..n}] \cdot \sigma' \cdot R' \cdot P'$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, $\Sigma \models S : E$, there exists an A' and a Σ' such that, $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $dom(\sigma') = dom(\Sigma')$, $\Sigma' \models v : A'$, and $\vdash A' <: A$.

Now by the [Val Unregister] rule, Figure 4.14, v must be some $(\iota, \ I)$ and, A' must be $\mathcal{R}$. Now let $R'' = (R', (\iota, \ I) \leftarrow <>)$. Since $\Sigma' \models R'$ and by the [Registry <> Typing] rule, $\Sigma' \models (\iota, \ I) \mapsto <>$, $\Sigma' \models R''$.

Therefore, $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R''$, $dom(\sigma') = dom(\Sigma')$, $\Sigma \models [\,] : [\,]$, and $\vdash [\,] <: [\,]$.

Case (Red Publish)

Suppose the expression, α, is $\varpi(a.l_j)$ and the last step of the derivation is the [Red Publish] rule, Figure 4.7.

By hypothesis, $E \vdash \varpi(a.l_j) : A$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S : E$.

By the [Val Publish] rule, Figure 4.14, A must be [ ].

By induction hypothesis, since $E \vdash a : [l_i : B_i \ ^{i \in 1..n}]$, $\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow [l_i = \iota_i \ ^{i \in 1..n}] \cdot \sigma' \cdot R' \cdot P'$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S : E$, there exists an A' and a Σ' such that, $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $dom(\sigma') = dom(\Sigma')$, $\Sigma' \models [l_i = \iota_i \ ^{i \in 1..n}] : A'$, and $\vdash A' <: [l_i : B_i \ ^{i \in 1..n}]$.

Therefore, $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $dom(\sigma') = dom(\Sigma')$, $\Sigma' \models [\,] : [\,]$, and $\vdash [\,] <: [\,]$.

Case (Red Unpublish)

Suppose the expression, α, is $\not\varpi(a.l_j)$ and the last step of the derivation is the [Red Unpublish] rule, Figure 4.7.

By hypothesis, $E \vdash \varpi(a.l_j) : A$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S : E$.

By the [Val Unpublish] rule, Figure 4.14, A must be [ ].

By induction hypothesis, since $E \vdash a : [l_i : B_i \ ^{i \in 1..n}]$, $\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow [l_i = \iota_i \ ^{i \in 1..n}] \cdot \sigma' \cdot R' \cdot P'$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S : E$, there exists an A' and a Σ' such that, $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $dom(\sigma') = dom(\Sigma')$, $\Sigma' \models [l_i = \iota_i \ ^{i \in 1..n}] : A'$, and $\vdash A' <: [l_i : B_i \ ^{i \in 1..n}]$.

Therefore, $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $dom(\sigma') = dom(\Sigma')$, $\Sigma' \models [\,] : [\,]$, and $\vdash [\,] <: [\,]$.

Case (Red Update and Announce)

Suppose the expression, α, is $a.l_j \Leftarrow \varsigma(x)b$, and $a.l_j \in P$ and the last step of the derivation is the [Red Update and Announce] rule, Figure 4.7.

By hypothesis, $E \vdash a.l_j \Leftarrow \varsigma(x)b : A$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S : E$.

The update part of the proof for the [Red Update and Announce] rule is the same as for the [Red Update] rule [AC96, Ch. 11] and is given again here for completeness.

Since, $E \vdash a.l_j \Leftarrow \varsigma(x)b : A$, we must have, by the [Val Update] typing rule, Figure 4.12, $E \vdash a : [l_j : B_j, \ldots]$, and $E,\ x : [l_j : B_j, \ldots] \vdash b : B_j$ for some $[l_j : B_j, \ldots]$ such that $\vdash [l_j : B_j, \ldots] <: A$.

By induction hypothesis, since $E \vdash a : [l_j : B_j, \ldots]$, $\sigma \cdot R \cdot P \cdot S \vdash a \rightsquigarrow [l_i = \iota_i\ ^{i \in 1..n}] \cdot \sigma' \cdot R^\dagger \cdot P^\dagger$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S : E$, there exists a type $A^\dagger$, and a store typing environment $\Sigma^\dagger$, such that, $\Sigma \preceq \Sigma^\dagger$, $\Sigma^\dagger \models \sigma'$, $\Sigma^\dagger \models R^\dagger$, $dom(\sigma') = dom(\Sigma^\dagger)$, $\Sigma^\dagger \models [l_i = \iota_i\ ^{i \in 1..n}] : A^\dagger$, and $\vdash A^\dagger <: [l_j : B_j, \ldots]$.

By assumption, $\iota_j \in dom(\sigma')$, hence $\iota_j \in dom(\Sigma^\dagger)$. But, $\Sigma^\dagger \models [l_i = \iota_i\ ^{i \in 1..n}] : A^\dagger$ must have been derived via the [Result Object] rule, from $\Sigma_1^\dagger(\iota_i) = [l_i : \Sigma_2^\dagger(\iota_i)^{i \in 1..n} = A^\dagger]$ for all $i \in 1..n$. Hence, since $\vdash A^\dagger <: [l_j : B_j, \ldots]$, we have, $\Sigma_2^\dagger(\iota_j) = B_j$.

(1) By lemma 4.4.3, we have $\Sigma^\dagger \models S : E$. Also, by lemma 4.4.5, we have $E,\ x : A^\dagger \vdash B_j$, from $E,\ x : [l_j : B_j, \ldots] \vdash b : B_j$ and $\vdash A^\dagger <: [l_j : B_j, \ldots]$. So, $E,\ x : \Sigma_1^\dagger(\iota_j) \vdash b : \Sigma_2^\dagger(\iota_j)$.

(2) Since, by the [Store Typing] rule, $\Sigma^\dagger \models \sigma'$, σ' is of the form $\varepsilon_k \mapsto\!< \varsigma(x_k)b_k, S_k >^{k \in 1..m}$, and for all k such that $\varepsilon_k \neq \iota_j$ and for some E_k we have $\Sigma^\dagger \models S_k : E_k$, and $E_k,\ x_k : \Sigma_1^\dagger(\varepsilon_k) \vdash b_k : \Sigma_2^\dagger(\varepsilon_k)$.

Let $\sigma^\dagger = (\sigma'.\iota_j \leftarrow\!< \varsigma(x)b, S >)$.

Now, by (1) and (2) and the [Store Typing] rule, we have $\Sigma^\dagger \models \sigma^\dagger$. Also, since $dom(\sigma^\dagger) = dom(\sigma')$, $dom(\sigma^\dagger) = dom(\Sigma^\dagger)$. Further, by transitivity of $<:$, $A^\dagger <: A$.

The rest of this case will be for the announce part of the [Red Update and Announce] rule.

The next hypothesis in the rule is $R^\dagger \cdot \sigma^\dagger \cdot R^\dagger \cdot P^\dagger \vdash R^\dagger(\iota_j) \rightharpoonup D \cdot \sigma'' \cdot R'' \cdot P''$.

The hypothesis from the theorem that fits this rule requires that the resulting set from $R^\dagger(\iota_j)$ be as follows: $\forall e \in R^\dagger(\iota_j).\exists(\iota, I).(\iota, I) \mapsto e \in R^\dagger$. The definition of $R^\dagger(\iota_j)$ is $\{x \mid R^\dagger((\iota_j, I)) = x\}$. Thus, every closure of the set resulting from $R^\dagger(\iota_j)$ has a mapping from some (ι_j, I) to that closure in $R^\dagger$ by definition.

So, from, $R^\dagger \cdot \sigma^\dagger \cdot R^\dagger \cdot P^\dagger \vdash R^\dagger(\iota_j) \rightharpoonup D \cdot \sigma'' \cdot R'' \cdot P''$, $\Sigma^\dagger \models \sigma^\dagger$, $\Sigma^\dagger \models R^\dagger$, $dom(\sigma) = dom(\Sigma)$, and $\forall e \in R^\dagger(\iota_j).\exists(\iota, I).(\iota, I) \mapsto e \in R^\dagger$, the induction hypothesis gives a store typing environment, Σ'', such that $\Sigma^\dagger \preceq \Sigma''$, $\Sigma'' \models \sigma''$, $\Sigma'' \models R''$, $dom(\sigma'') = dom(\Sigma'')$.

Now, since $D \subseteq R^\dagger(\iota_j)$, we have $\forall e \in D.\exists(\iota, I).(\iota, I) \mapsto e \in R^\dagger$.

We then have, $R^\dagger \cdot \sigma'' \cdot R'' \cdot P'' \vdash D \rightharpoonup\ \cdot \sigma''' \cdot R''' \cdot P'''$ $\Sigma'' \models \sigma''$, $\Sigma'' \models R''$, $\Sigma^\dagger \models R^\dagger$, $dom(\sigma'') = dom(\Sigma'')$, and $\forall e \in D.\exists(\iota, I).(\iota, I) \mapsto e \in R^\dagger$. Thus, by the induction hypothesis, there is a store typing environment Σ''', such that, $\Sigma'' \preceq \Sigma'''$, $\Sigma''' \models \sigma'''$, $\Sigma''' \models R'''$, $dom(\sigma''') = dom(\Sigma''')$.

By the transitivity of $\preceq$, $\Sigma \preceq \Sigma'''$. Also, since $\Sigma^\dagger \models [l_i = \iota_i\ ^{i \in 1..n}] : A^\dagger$, and $\Sigma^\dagger \preceq \Sigma'''$ by transitivity of $\preceq$, $\Sigma''' \models [l_i = \iota_i\ ^{i \in 1..n}] : A^\dagger$ by Corollary 4.4.4.

Therefore, $\Sigma \preceq \Sigma'''$, $\Sigma''' \models \sigma'''$, $\Sigma''' \models R'''$, $dom(\sigma''') = dom(\Sigma''')$, $\Sigma''' \models [l_i = \iota_i\ ^{i \in 1..n}] : A^\dagger$, and $\vdash A^\dagger <: A$

Case (Event Red Filter)

Suppose the subsystem set, δ, is $EVT \uplus \{< b, c, d, S' >\}$, where EVT and D are subsystem sets, b, c, and d are expressions, S' is a stack, and $< b, c, d, S' >$ is a event registration closure. Further, suppose that the reduction is a '$\rightharpoonup$' reduction and that the last step of the derivation is the [Event Red Filter] rule.

By the hypothesis, $\Sigma \models \sigma$, $\Sigma \models R$, $\Sigma_0 \models R_0$, $dom(\sigma) = dom(\Sigma)$, and $\forall e \in \delta.\exists(\iota, I).(\iota, I) \mapsto e \in R_0$. Namely, there is a (ι_0, I_0) such that, $(\iota_0, I_0) \mapsto\!< b, c, d, S' >\in R_0$ since $< b, c, d, S' >\in \delta$.

Now, since $\Sigma_0 \models R_0$ and $(\iota_0, I_0) \mapsto\!< b, c, d, S' >\in R_0$ we have, by the [Registry Typing] rule that $\Sigma \models (\iota_0, I_0) \mapsto\!< b, c, d, S' >$. Then, by the [Registry Closure Typing] rule, we have $\Sigma_0 \models S' : E$, $E \vdash b : Top$, and $E \vdash c : Top$ (where E satisfies the [Stack x Typing] and [Stack ϕ Typing] rules).

Since $E \vdash b : Top$, $\sigma \cdot R \cdot P \cdot S' \vdash b \rightsquigarrow u \cdot \sigma' \cdot R' \cdot P'$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S' : E$, by the induction hypothesis, there is a type, A', and a store typing environment, Σ', such that $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $dom(\sigma') = dom(\Sigma')$, $E \vdash u : A'$, and $\vdash A' <: Top$.

Now, by Lemma 4.4.3 we have $\Sigma' \models S' : E$.

So, since $E \vdash c : Top$, $\sigma' \cdot R' \cdot P' \cdot S' \vdash c \rightsquigarrow w \cdot \sigma'' \cdot R'' \cdot P''$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $dom(\sigma') = dom(\Sigma')$, and $\Sigma' \models S' : E$, by the induction hypothesis, there is a type, A'', and a store typing environment, Σ'', such that $\Sigma' \preceq \Sigma''$, $\Sigma'' \models \sigma''$, $\Sigma'' \models R''$, $dom(\sigma'') = dom(\Sigma'')$, $E \vdash w : A''$, and $A'' <: Top$.

Since $\forall e \in \delta.\exists(\iota, I).(\iota, I) \mapsto e \in R_0$ and $EVT \subseteq \delta$, we have $\forall e \in EVT.\exists(\iota, I).(\iota, I) \mapsto e \in R_0$.

Thus, since $R_0 \cdot \sigma'' \cdot R'' \cdot P'' \vdash EVT \rightharpoonup D \cdot \sigma''' \cdot R''' \cdot P'''$, $\Sigma'' \models \sigma''$, $\Sigma'' \models R''$, $\Sigma_0 \models R_0$, $dom(\sigma'') = dom(\Sigma'')$, and $\forall e \in EVT.\exists(\iota, I).(\iota, I) \mapsto e \in R_0$, by the induction hypothesis we have a store typing environment, Σ''', such that, $\Sigma'' \preceq \Sigma'''$, $\Sigma''' \models \sigma'''$, $\Sigma''' \models R'''$, and $dom(\sigma''') = dom(\Sigma''')$.

By the transitivity of $\preceq$, $\Sigma \preceq \Sigma'''$.

Therefore, $\Sigma \preceq \Sigma'''$, $\Sigma''' \models \sigma'''$, $\Sigma''' \models R'''$, and $dom(\sigma''') = dom(\Sigma''')$.

Case (Event Red Empty)

Suppose the subsystem set, δ, is $EVT \uplus \{< >\}$ where, $< >$ is an empty registration closure and EVT is a subsystem set. Further, suppose that the reduction is a '$\rightharpoonup$' reduction and that the last step of the derivation is the [Event Red Empty] rule.

By the hypothesis, $\Sigma \models \sigma$, $\Sigma \models R$, $\Sigma_0 \models$, $dom(\sigma) = dom(\Sigma)$, and $\forall e \in \delta.\exists(\iota, I).(\iota, I) \mapsto e \in R_0$.

Now, the condition, $\forall e \in EVT.\exists(\iota, I).(\iota, I) \mapsto e \in R_0$ holds because $EVT \subseteq \delta$.

So, since $R_0 \cdot \sigma \cdot R \cdot P \vdash EVT \rightharpoonup D \cdot \sigma' \cdot R' \cdot P'$, $\Sigma \models \sigma$, $\Sigma \models R$, $\Sigma_0 \models R_0$, $dom(\sigma) = dom(\Sigma)$, and $\forall e \in EVT.\exists(\iota, I).(\iota, I) \mapsto e \in R_0$, by the induction hypothesis we have, $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, and $dom(\sigma') = dom(\Sigma')$.

Therefore, $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, and $dom(\sigma') = dom(\Sigma')$.

Case (Event Red Handler)

Suppose the subsystem set, δ, is $D \uplus \{< b, c, d, S' >\}$, where b, c, and d are expressions, S' is a stack, $< b, c, d, S' >$ is a event registration closure, and D is a subsystem set. Further, suppose that the reduction is a '$\rightharpoonup$' reduction and that the last step of the derivation is the [Event Red Handler] rule.

By the hypothesis, $\Sigma \models \sigma$, $\Sigma \models R$, $\Sigma_0 \models R_0$, $dom(\sigma) = dom(\Sigma)$, and $\forall e \in \delta.\exists(\iota, I).(\iota, I) \mapsto e \in R_0$. Namely, there is a (ι_0, I_0) such that, $(\iota_0, I_0) \mapsto < b, c, d, S' > \in R_0$ since $< b, c, d, S' > \in \delta$.

Now, since $\Sigma_0 \models R_0$ and $(\iota_0, I_0) \mapsto < b, c, d, S' > \in R_0$, we have, by the [Registry Typing] rule that $\Sigma_0 \models (\iota_0, I_0) \mapsto < b, c, d, S' >$. Then, by the [Registry Closure Typing] rule we have $\Sigma_0 \models S' : E$ and $E \vdash d : Top$ (where E satisfies the [Stack x Typing] and [Stack ϕ Typing] rules).

So, since $E \vdash d : Top$, $\sigma \cdot R \cdot P \cdot S' \vdash d \rightsquigarrow v \cdot \sigma' \cdot R' \cdot P'$, $\Sigma \models \sigma$, $\Sigma \models R$, $dom(\sigma) = dom(\Sigma)$, and $\Sigma \models S' : E$, by the induction hypothesis we have a type A' and a store typing environment Σ', such that, $\Sigma \preceq \Sigma'$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $dom(\sigma') = dom(\Sigma')$, $\Sigma' \models v : A'$, and $\vdash A' <: Top$.

Now, the condition, $\forall e \in D.\exists(\iota, I).(\iota, I) \mapsto e \in R_0$ holds since $D \subseteq \delta$.

Thus, since $R_0 \cdot \sigma' \cdot R' \cdot P' \vdash D \rightharpoonup \cdot \sigma'' \cdot R'' \cdot P''$, $\Sigma' \models \sigma'$, $\Sigma' \models R'$, $\Sigma_0 \models R_0$, $dom(\sigma') = dom(\Sigma')$, and $\forall e \in D.\exists(\iota, I).(\iota, I) \mapsto e \in R_0$, by the induction hypothesis we have a store typing environment, Σ'', such that, $\Sigma' \preceq \Sigma''$, $\Sigma'' \models \sigma''$, $\Sigma'' \models R''$, and $dom(\sigma'') = dom(\Sigma'')$.

By the transitivity of $\preceq$, $\Sigma \preceq \Sigma''$.

Therefore, $\Sigma \preceq \Sigma''$, $\Sigma'' \models \sigma''$, $\Sigma'' \models R''$, and $dom(\sigma'') = dom(\Sigma'')$. □

4.5 Conclusion

The syntax and semantics for the $\varrho\varpi\varsigma$-calculus detail a concise formalism that supports events. We have seen that the type system for the $\varrho\varpi\varsigma$-calculus given here is sound, showing that $\varrho\varpi\varsigma$-calculus expressions can be checked to ensure that they have a valid reduction in the calculus — a valuable quality of a foundational formalism. In the next chapter, we will demonstrate that the $\varrho\varpi\varsigma$-calculus is useful by using it as the formal foundation for a core programming language, Eventua.

Chapter 5

EVENTUA: A LANGUAGE WITH EVENTS

In this chapter, a core language, Eventua, is built on the calculus work from the previous chapter. The goal is for Eventua to contain syntax and semantics that is likely to be expected in a programming language that has native support for events. The semantics for Eventua are defined as translations from the Eventua syntax to full or partial $\varrho\varpi\varsigma$-calculus expressions. A type system for Eventua is also defined along with its translation into $\varrho\varpi\varsigma$-calculus types.

The Eventua core language demonstrates two things: the $\varrho\varpi\varsigma$-calculus is general enough to express variants of event semantics, and to illustrate that programming language support for events can be useful. We demonstrate the first by showing that the event trigger mechanism in the $\varrho\varpi\varsigma$-calculus, implicit trigger on assignment, is enough to support Eventua's event triggering semantics, which includes explicit triggering and implicit triggering on assignment and on method invocation. We also demonstrate the flexibility of the calculus, for example, by showing how event handlers can be tied to objects in Eventua; something that is not done in the calculus. Further, Eventua shows that the calculus is general enough to support parameter passing on both events and methods, despite the fact that the calculus does not directly support it.

The examples in this chapter are intended to show how Eventua's event mechanism works, but they also show that the mechanism can be useful for solving real-world problems.

This research does not focus on the question of whether implicit invocation is, in general, useful since other research, such as Dingel, et. al. [DGJN98a], Barrett, et. al. [BCTW96], and Garlan and Notkin [GN91], cover that. Instead, we show that it is useful to have native implicit invocation in a programming language.

5.1 Syntax

The syntax for Eventua in Figure 5.1 is intended to model a practical object-oriented language that includes native support for events by including classes, fields, a self keyword, and parameter passing for both methods and events. The type syntax is included in Figure 5.2 to complete the definition of the syntax. In this section, we explain the syntax for Eventua as well as the syntax for Eventua types.

The Eventua syntax is a compromise between simplicity and functionality. The goal was to come up with a core language that was simple enough to avoid getting bogged down in theoretical details, but functional enough to show that Eventua represents a useful object-oriented language with support for events. As illustrated in the syntax, Eventua programs are broken up into two sections: class declarations and computation.

Class declarations are made up of fields (public and private), methods, events, and event handlers. The fields and methods are just as one would expect of any object-oriented language. The event declarations are there to provide an explicit way to say that an object member will be used solely for the purpose of raising events. This event declaration provides a simple syntax for declaring parameters that can be passed when an event is raised. We also have event handlers which are similar to methods except that they can be attached to various implicit and explicit events. An event handler can specify that it is able to take parameters passed in from the raising of an event. Figure 5.3 is an example of a class declaration with a field and an explicit event. The use of T and U should be viewed as class types, for arbitrary classes T and U, for this and any remaining examples.

The computation, i.e., body of the program, is shown as a single expression. However, with the inclusion of the **let** expression in Eventua, it is possible to string together several expressions to be executed sequentially, simulating a list of sequential instructions. The expression productions can be grouped together by what they do: data and method access, store management, and event management.

The data and method access productions all provide read, write, or member selection capabilities. **self**

$< prgm >$	::=		
		$< cl >_i \ ^{i \in 1..n} \ < exp >$	Program
$< cl >$	::=		
		class l **begin** $< decl >_i \ ^{i \in 1..n}$ **end**	Class declaration
$< decl >$	::=		
		$l =$ **field** $: A$	Field declaration
	$\mid$	$l =$ **field** $: A := < exp >$	Field decl and init
	$\mid$	$l =$ **private field** $: A$	Private field declaration
	$\mid$	$l =$ **private field** $: A := < exp >$	Private field decl and init
	$\mid$	$l =$ **method**$(x_1 : T_1, \ldots, x_n : T_n) : A < exp >$	Method declaration
	$\mid$	$l =$ **event**$(x_1 : T_1, \ldots, x_n : T_n)$	Event declaration
	$\mid$	$l =$ **handler**$(x_1 : T_1, \ldots, x_n : T_n) < exp >$	Event handler declaration
$< exp >$	::=		
		self	Self identifier
	$\mid$	x	Identifier
	$\mid$	$< exp >.l$	Selection
	$\mid$	**private**.l	Private selection
	$\mid$	**private**.$l \ := \ < exp >$	Private update
	$\mid$	$< exp >.l(< exp >_1, \ldots, < exp >_n)$	Invocation
	$\mid$	$< exp >_1.l \ := \ < exp >_2$	Field update
	$\mid$	**let** $x = < exp >_1$ **in** $< exp >_2$	Let expression
	$\mid$	**new** l	Object creation
	$\mid$	**register** $< exp >_1.l_i$ **for** $< reg >< when >$	Registration
	$\mid$	**announce** $< exp >.l(< exp >_1, \ldots, < exp >_n)$	Announce
	$\mid$	**unregister** $< exp >.l$	Unregistration
	$\mid$	**publish** $< rspec >< exp >.l$	Publish
	$\mid$	**unpublish** $< rspec >< exp >.l$	Unpublish
$< reg >$	::=		
		invocation of $< exp >_2.l_j(p_1, \ldots, p_k)$	invocation event registration
	$\mid$	(**event** $\mid$ **update of** $)< exp >_2.l_j$	other event registration
$< when >$	::=		
			unconditional registration
	$\mid$	**when** $< exp >_3 \equiv < exp >_4$	conditional registration
$< rspec >$	::=		
		event $\mid$ **update of** $\mid$ **invocation of**	Event handling specifiers

Figure 5.1: Eventua Syntax

A, B, T	::=		
		Class(A)	Class type
	$\mid$	**Field**(A)	Field type
	$\mid$	**Private**(A)	Private field type
	$\mid$	**Method**$(A; T_1, T_2, \ldots, T_n)$	Method type
	$\mid$	**Event**$(T_1, T_2, \ldots, T_n)$	Event type
	$\mid$	**Handler**$(T_1, T_2, \ldots, T_n)$	Handler type
	$\mid$	**Object**$(l_i : B_i \ ^{i \in 1..n})$	Object type
	$\mid$	**Top**	Topmost type
	$\mid$	K	Constant type

Figure 5.2: Eventua Type Syntax

```
class  myClass  begin
       myField = field : int
       myEvent = event(x:T, y:U)
end
```

Figure 5.3: Eventua Class Declaration Example

```
let  x = a.data := new One  in  a.data
```

Figure 5.4: Eventua Sequencing and Data Access Example

provides access to the current object, like the self parameter in the $\varrho\varpi\varsigma$-calculus. x represents a variable such as one created by a **let** expression. The dot ('.') notation provides selection into an object of a public or private field, or a method. Finally, the assignment notation (':=') allows for assignment to object fields. Figure 5.4 is an example showing an update and a data access expression on an object, a, using a **let** expression as a sequencing operator. The example first updates the *data* member of a with a new *One* object, then it accesses the *data* member of a. The result of the access will be the new *One* object that was just created, demonstrating that the let caused the two expressions to be sequenced one after the other. The let variable, x, is a previously unused variable demonstrating how a **let** expression can be used solely for sequencing.

The **let** and **new** expressions make up the store management productions. The **let** expression provides for a kind of stack storage while **new** provides storage that is similar to heap storage. As mentioned earlier, the **let** expression also provides ordering since the variable being created by the **let** must be initialized before the body of the **let** is computed. The **new** expression provides a way for an Eventua class to be instantiated, creating the actual objects used in the Eventua program expression. The example in Figure 5.5 shows how **let** and **new** expressions can be used. For the example, assume that the class, *myClass*, is defined as in Figure 5.3.

The event management productions are similar to that of their $\varrho\varpi\varsigma$-calculus counterparts, managing publication and registration, however Eventua also provides syntax for explicit event announcement. There are other differences from the $\varrho\varpi\varsigma$-calculus as well. The **register** expression in Eventua connects an event generating member of one object to the event handler of another object, whereas the $\varrho\varpi\varsigma$-calculus registration allows for an event generating member to be connected to an expression. Further, the syntax allows for registration of not only explicit events and implicit events on assignment, but it also allows for handling events due to the invocation of a method, via the **on invocation of** part of the registration expression. Events are explicitly announced via the **announce** expression, which allows parameters to be passed to its respective event handlers. The **publish** and **unpublish** productions are also different from the calculus in that they allow for a distinction between an explicit announcement, **event**, an implicit announcement due to an assignment, **update of**, or an implicit announcement due to a method invocation, **invocation of**. One final difference is in the **unregister** expression. The $\varrho\varpi\varsigma$-calculus ensures that every registration results in a unregistration value, providing registration with an exact inverse which is desirable for a calculus. To show that the unregistration syntax and semantics in the calculus can be used to represent a different unregistration scheme, Eventua allows event handlers to be unregistered in a similar way to how they were registered. The disadvantage of this approach over the calculus is the possibility of unregistering another's registration. Figure 5.6 is an example of an event getting published, an event handler being registered, and an event being announced. The example assumes that *myClass* is already defined as in Figure 5.3.

The unregistration semantics can lead to semantic misunderstanding in Eventua because as previously discussed the syntax does not allow for a way to distinguish which handler-to-event connection should be unregistered. This is a problem because a mapping from handlers to events is not guaranteed to be an injective (at

```
let  obj = new myClass in obj.myField := 5
```

Figure 5.5: Eventua Let and New Example

```
class HandlerClass begin
    aHandler = handler(arg:T)
        self.state := arg
    state = field : T
end
    let obj = new myClass in
        let handler = new HandlerClass in
            let x = publish event obj.myEvent in
            let y = register handler.aHandler for event obj.myEvent in
            announce obj.myEvent(new T, new U)
```

Figure 5.6: Eventua Registration Example

most one event per handler) mapping because a handler can be registered for multiple events. As we will see in the translation from Eventua to the $\varrho\varpi\varsigma$-calculus, this is handled by simply unregistering the handler from *all* of its respective events.

The type syntax provides types for classes, class members, and objects, as well as a topmost type, **Top**. As we will see from the translation of the syntax and the types, the different types for each of the kinds of class members is important for the translation to work out correctly. Also, while the type **Class** appears to be able to contain any kind of type, according to the type grammar it will only hold the type of the objects it will produce. That is, a class type will always be listed as something like, **Class(Object(...))**. Schmidt [Sch94] makes heavy use this kind of typing based on structure.

5.2 Semantics

Eventua's semantics are defined by a set of translations from Eventua syntax to the syntax of the $\varrho\varpi\varsigma$-calculus. Since we have formally defined the semantics for the calculus in chapter 4, the $\varrho\varpi\varsigma$-calculus translation provides a formal semantics for Eventua. Each translation will start with an Eventua syntax rule and its accompanying translation, with recursive translations when necessary. A translation operator, $\ll \gg$, will be used to denote a pending translation and the definition operator, $\stackrel{\triangle}{=}$, is used to show how one step of the translation is defined, with the left side being the before part and the right side, the after part.

As discussed above in the introduction section, Eventua provides for parameter passing but the $\varrho\varpi\varsigma$-calculus does not. The typical way that Abadi and Cardelli [AC96] handle parameter passing in their calculi is to create extra object members for each parameter, which is the same way it has been done here. A problem with using that technique here is that the names for the parameters are arbitrary and not guaranteed to be uniquely named across an entire object, e.g., two different methods could have a parameter named, *x*. A second problem occurs for translations that contain parameter passing, e.g. *obj.f*([ ]) — there is, in general, no context free way to know which member of the object is supposed to get the parameter. Both of these problems are solved by, first, defining a standard naming convention for the formal parameters for the members of objects, and second, by allowing a context to be passed through the translation to allow for conformance to the naming convention. Thus, the translation function is given a context parameter that defines a mapping, called the *translation environment function*, from each original parameter's name to the corresponding translation's naming convention name.

The translation environment function is used to "fix up" references to formal parameters in methods and event handlers forcing the methods and event handlers to use the parameters according to the naming convention. Further, as seen in Figure 5.7, the naming convention for each parameter is a combination of the name of the member, e.g., the f in *x.f*([ ]), and the position of the parameter relative to the others. Thus, the naming convention narrows the context sensitivity of the parameter passing translation down to the expression directly responsible for the parameter passing, making the translation of that expression context free.

Definition 5.2.1 (ρ). The translation environment function, ρ, is a function mapping Eventua variables to $\varrho\varpi\varsigma$ terms as follows:

$\phi(x) \triangleq x$ where ϕ is the empty environment.

$$\rho\{y \leftarrow a\}(x) \triangleq \begin{cases} a & \text{if } x = y \\ \rho(x) & \text{otherwise} \end{cases}$$

The entire Eventua-to-$\varrho\varpi\varsigma$-calculus syntax translation is presented in three parts: declaration, non-registration expressions, and registration expressions. To make the translations easier to read, a ; b is used for $let\ x = a\ in\ b$ where $x \notin FV(b)$. The meaning of $FV(b)$ is the set of the free variables in expression b. Also, parens are used to show the binding scope of a let expression. Further, to avoid variable name capture, we have adopted the convention that underbars (_) cannot be used to start variable names in Eventua, allowing us to use such variables freely for the translations.

The translation of an entire Eventua program is specified by the translation of the $< prog >$ production, the first rule in Figure 5.7. This rule shows that each class declaration is translated individually followed by the translation of the body of the program. Notice that the context parameter, ρ, is not maintained from one class translation to the next. That is, each class translation, and the translation of the body of the program, get their own translation context. As the figure illustrates and the previous discussion of the translation syntax outlines, the context parameter is only useful for handling the translation of parameters and references to the class from within its own definition.

The translation of a class declaration results in a **let** expression where the **let** variable refers to a $\varrho\varpi\varsigma$-calculus object that is a factory for instances of the given class. The body of the **let** expression is the rest of the program, i.e., any other class definitions and the final, main expression. The self parameter for the factory, *_class*, and the respective translation environment function, $l \leftarrow$ *_class* allow for members of the class to refer to the class itself. The members of the class cannot refer to the class name in the translation because the class name *is* the name of the **let** variable. That is, in the translation, the class name will not be in the scope of the definition of the class. So, the factory object's self parameter and the translation environment are used to fix this problem.

Each member of a class relies heavily on the member name to help ensure that the translation provides the desired semantics. Members (or parts of members) that have no up-front initialization provided for in Eventua get self-referential, infinite loops in the translation. As we will see in the discussion of Eventua's type system and translation, section 5.4, this will provide for the appropriate type translation to the $\varrho\varpi\varsigma$-calculus type system for these fields.

The **field** and **private** member translations that have initialization are set up to avoid multiple evaluations of the initialization expression. This is accomplished by saving the result of the expression evaluation in a **let** variable, setting the value of the translated member to the value of the **let** variable, then returning that value. The next time the field is referred to, it simply returns that value (until the field is changed with an assignment). This also means that the initialization expression does not get evaluated until the first time the field is accessed. This will make programs that assign to a member that is used in an initialization pretty hard to read. Here is a simple example:

```
class OddClass begin
    normal = field : Nat := 5
    odd = field : Nat := self.normal
    theExample = method() : Nat
        self.normal := 22;
        self.odd
end
(new OddClass).theExample()
```

Reading sequentially from top to bottom, it would appear that the evaluation of the initialization should take place at the time of the declaration, meaning that the answer would be 5. However, since the initialization expression is not evaluated until the first access of its variable, the answer will end up as 22 since *normal* is assigned a new value before *odd* is accessed.

The private variable semantics are upheld in the translation via object scope and the naming convention. To ensure that private variables are not accessible via standard member selection, they are given a $\_priv_l$ label, with the subscript, l in this case, being the original label for the private variable. Due to the naming conventions we have established for Eventua, there is no way to produce this label with a standard Eventua member selection. Since private variables are not accessible via standard member selection, an alternate selection mechanism for

$$\ll cl_i\ ^{i\in 1..n}\ exp \gg_\phi \stackrel{\triangle}{=} \ll cl_i \gg_\phi\ ^{i\in 1..n}\ \ll exp \gg_\phi$$

$$\begin{array}{l}
\ll \textbf{class}\ l\ \textbf{begin}\ decl_i\ ^{i\in i..n}\ \textbf{end} \gg_\phi \stackrel{\triangle}{=} \\
\quad let\ l = [_new = \varsigma(_class) \\
\qquad\qquad [\ll decl_i \gg_{\{l \leftarrow _class\}}\ ^{i\in 1..n}]\ ] \\
\quad in
\end{array}$$

$$\ll l = \textbf{field}\ : A \gg_\rho \stackrel{\triangle}{=} l = \varsigma(self) self.l$$

$$\begin{array}{l}
\ll l = \textbf{field}\ : A := exp \gg_\rho \stackrel{\triangle}{=} l = \varsigma(self) let\ _v = \ll exp \gg_\rho\ in \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad self.l \Leftarrow \varsigma(_fresh)_v; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad _v
\end{array}$$

$$\ll l = \textbf{private field}\ : A \gg_\rho \stackrel{\triangle}{=} \_priv_l = \varsigma(self) self.\_priv_l$$

$$\begin{array}{l}
\ll l = \textbf{private field}\ : A := exp \gg_\rho \stackrel{\triangle}{=} _priv_l = \varsigma(self) let\ _v = \ll exp \gg_\rho\ in \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad self._priv_l \Leftarrow \varsigma(_fresh)_v; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad _v
\end{array}$$

$$\begin{array}{l}
\ll l = \textbf{method}(x_1 : T_1, \ldots, x_n : T_n) : A\ exp \gg_\rho \stackrel{\triangle}{=} \\
\quad l = \varsigma(self) \ll exp \gg_{\rho\{x_1 \leftarrow self._l_1\} \ldots \{x_n \leftarrow self._l_n\}}, \\
\quad _l_1 = \varsigma(self) self._l_1, \\
\quad \ldots \\
\quad _l_n = \varsigma(self) self._l_n \\
\quad _invoke_l = \varsigma(self)[\ ]
\end{array}$$

$$\begin{array}{l}
\ll l = \textbf{event}(x_1 : T_1, \ldots, x_n : T_n) \gg_\rho \stackrel{\triangle}{=} l = \varsigma(_x)[\ ], \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad _l_1 = \varsigma(self) self._l_1, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad _l_n = \varsigma(self) self._l_n
\end{array}$$

$$\begin{array}{l}
\ll l = \textbf{handler}(x_1 : T_1, \ldots, x_n : T_n) exp \gg_\rho \stackrel{\triangle}{=} \\
\quad l = \varsigma(self)\ let\ _ev = [_body = \varsigma(_y) \ll exp \gg_{\rho\{x_1 \leftarrow self._l_1\} \ldots \{x_n \leftarrow self._l_n\}}, \\
\qquad\qquad\qquad\qquad\qquad\quad _uregevt = \varsigma(_y)[\ ] \\
\qquad\qquad\qquad\qquad\qquad\quad] \\
\qquad\qquad\qquad\qquad in\ (\ self.l \Leftarrow \varsigma(_y)_ev; _ev\ ), \\
\quad _l_1 = \varsigma(self) self._l_1, \\
\quad \ldots \\
\quad _l_n = \varsigma(self) self._l_n
\end{array}$$

Figure 5.7: Translation of Eventua Classes into the $\varrho\varpi\varsigma$-calculus

private variables is introduced, **private**.l (where l would be the private variable), in Figure 5.8. Since the private selection mechanism is an unrestricted expression production, i.e., the use of the select is not syntactically limited as an expression (see Figure 5.1), this gives the potential for private variables to be exposed. However, this potential is mitigated by use of *self* in the translation for the selection. That is, private selection translations use object scope via *self* to ensure that private variables of an object remain private.

The method, event, and handler declaration productions all account for parameter passing by defining extra members in the translated object for each parameter. The naming convention for the parameter members ensures that they will be unique; related to the appropriate method, event, or event handler; and that they will be used according to the order that the parameters were defined in the Eventua declaration. The method translation also creates an object member that is used for announcing when the method is invoked, $\_invoke_l$, where l is the method name. The translation of an event declaration creates a placeholder in the $\varrho\varpi\varsigma$-calculus for event announcement.

For event handlers, the translation turns the member into an object with a body and a placeholder that will be used to signal an unregister event for the handler. As shown in Figure 5.9, Figure 5.10, and Figure 5.11, the result of the $\varrho\varpi\varsigma$-calculus registration of the event handler to its respective event is saved. Then, essentially, another event handler is registered for the unregister event. This handler uses the registration value from the first registration to unregister the handler. Therefore, the translation of an unregister, as shown in Figure 5.8, simply causes the unregister event to be announced, causing the handler to be unregistered from every event it registered to handle.

The **self** keyword is always translated to *self*, since the translations of the methods and event handlers all use *self* as their respective self reference parameters. Formally, it is not valid to use any self reference parameter name more than once, because the definition of the $\varrho\varpi\varsigma$-calculus semantics requires that each of these self parameters have a unique name ([Stack x Regval] rule and [Stack x Object] rule, Figure 4.5). However, it is easy enough to apply an alpha transformation to each respective occurrence of self, translating each to a fresh variable as formally required.

References to variables are translated with respect to the current translation environment, ρ. If the variable is listed in the translation environment, it is replaced according to the environment mapping. If there is no match, however, the variable is left as is.

Method invocation translations start by evaluating the target expression for the invocation and saving the result, hopefully an object value, in a let variable, *_obj*. Then, each actual parameter is set up to be evaluated in a left-to-right order. The actual-to-formal mapping is set up by the translation to be by-value via the use of the let expressions for each assignment to the formal parameter members of the target object. Next, the invocation announcement for the method is set up to be triggered by the assignment to $\_invoke_l$. Finally, the translation allows the actual method to be computed and its result is the result of the Eventua invocation expression.

The field assignment simply translates to a by-value assignment in the $\varrho\varpi\varsigma$-calculus. This translation assumes that the target of the assignment will be a field, but that assumption is not used to preclude this kind of assignment to things other than fields: methods, events, or event handlers. The assignment to a private field is similar to the regular assignment except that it accounts for the way private variable names must be handled.

Translations for explicit event announcement, **announce**, are similar to method invocations. The target expression is evaluated resulting in an object value. Then, the actual parameters for the announcement are evaluated left-to-right and passed over to the appropriate object members that hold the place for the formal parameters. Once the parameters are taken care of, the event is triggered by assignment to $\_obj_l$. Finally, the empty object is returned (events are triggered primarily for side-effects).

The publish and unpublish expressions both have three different forms: event, update, and invocation. The event and update actually have the exact same semantics, but are included for program clarity (events are not updated in Eventua and an update is not an event in Eventua). The invoke form allows for method invocation events to be published.

The translation for a new expression essentially calls the *_new* method on the factory for the objects, which, in turn, creates a new object that conforms to the class specification. The label is translated via the translation environment function to ensure that references to the class from within itself are translated correctly. As already mentioned, references to the class from within the class are allowed only if we translate those references to be the class factory's self parameter from its new method. The translation environment function ensures that this conversion will be made correctly, when appropriate.

There are three different forms for registration translations each of which may have an optional **when** clause

$$\ll \textbf{self} \gg_\rho \stackrel{\triangle}{=} self$$

$$\ll x \gg_\rho \stackrel{\triangle}{=} \rho(x)$$

$$\ll exp.l \gg_\rho \stackrel{\triangle}{=} \ll exp \gg_\rho .l$$

$$\ll \textbf{private}.l \gg_\rho \stackrel{\triangle}{=} self.\_priv_l$$

$$\begin{array}{l}\ll exp.l(x_1 : T_1, \ldots, x_n : T_n) exp \gg_\rho \stackrel{\triangle}{=} \\ \quad let\ \_obj = \ll exp \gg_\rho\ \ in \\ \quad\quad let\ \_temp_1 = \ll exp_1 \gg_\rho\ in\ \_obj.\_l_1 \Leftarrow \varsigma(\_y)\_temp_1; \\ \quad\quad \ldots \\ \quad\quad let\ \_temp_n = \ll exp_n \gg_\rho\ in\ \_obj.\_l_n \Leftarrow \varsigma(\_y)\_temp_n; \\ \quad\quad \_obj.\_invoke_l \Leftarrow \varsigma(\_y)[\ ]; \\ \quad\quad \_obj.l\end{array}$$

$$\begin{array}{l}\ll exp_1.l\ :=\ exp_2 \gg_\rho \stackrel{\triangle}{=} let\ \_temp = \ll exp_2 \gg_\rho\ \ in \\ \quad\quad\quad\quad \ll exp_1 \gg_\rho .l \Leftarrow \varsigma(x)\_temp\end{array}$$

$$\begin{array}{l}\ll \textbf{private}.l\ :=\ exp \gg_\rho \stackrel{\triangle}{=} let\ \_temp = \ll exp \gg_\rho \\ \quad\quad\quad\quad self.\_priv_l \Leftarrow \varsigma(\_x)\_temp\end{array}$$

$$\ll \textbf{let}\ x = exp_1\ \textbf{in}\ exp_2 \gg_\rho \stackrel{\triangle}{=} let\ x = \ll exp_1 \gg_\rho\ in \ll exp_2 \gg_\rho$$

$$\begin{array}{l}\ll \textbf{announce}\ exp.l(exp_1, \ldots, exp_n) \gg_\rho \stackrel{\triangle}{=} \\ \quad let\ \_obj = \ll exp \gg_\rho\ \ in \\ \quad\quad let\ \_temp_1 = \ll exp_1 \gg_\rho\ in\ \_obj.\_l_1 \Leftarrow \varsigma(x)\_temp_1; \\ \quad\quad \ldots \\ \quad\quad let\ \_temp_n = \ll exp_n \gg_\rho\ in\ \_obj.\_l_n \Leftarrow \varsigma(x)\_temp_n; \\ \quad\quad \_obj.l \Leftarrow \varsigma(x)[\ ]; \\ \quad\quad [\ ]\end{array}$$

$$\ll \textbf{unregister}\ exp.l \gg_\rho \stackrel{\triangle}{=} \ll exp \gg_\rho .l.\_uregevt \Leftarrow \varsigma(x)[\ ];\ [\ ]$$

$$\ll \textbf{publish event}\ exp.l \gg_\rho \stackrel{\triangle}{=} \varpi(\ll exp \gg_\rho .l)$$

$$\ll \textbf{publish update of}\ exp.l \gg_\rho \stackrel{\triangle}{=} \varpi(\ll exp \gg_\rho .l)$$

$$\ll \textbf{publish invocation of}\ exp.l \gg_\rho \stackrel{\triangle}{=} \varpi(\ll exp \gg_\rho .\_invoke_l)$$

$$\ll \textbf{unpublish event}\ exp.l \gg_\rho \stackrel{\triangle}{=} \not\varpi(\ll exp \gg_\rho .l)$$

$$\ll \textbf{unpublish update of}\ exp.l \gg_\rho \stackrel{\triangle}{=} \not\varpi(\ll exp \gg_\rho .l)$$

$$\ll \textbf{unpublish invocation of}\ exp.l \gg_\rho \stackrel{\triangle}{=} \not\varpi(\ll exp \gg_\rho .\_invoke_l)$$

$$\ll \textbf{new}\ l \gg_\rho \stackrel{\triangle}{=} \rho(l).\_new$$

Figure 5.8: Translation of Eventua Expressions into the $\varrho\varpi\varsigma$-calculus

$$\begin{array}{l}\ll \textbf{register}\ exp_1.handler\ \textbf{for update of}\ exp_2.member\ \textbf{when}\ test_1 \equiv test_2 \gg_\rho \stackrel{\triangle}{=} \\ \quad let\ \_v_1 = \ll exp_1 \gg_\rho\ in \\ \quad let\ \_v_2 = \ll exp_2 \gg_\rho\ in \\ \quad let\ \_rv = \varrho(\_v_2.member, \ll test_1 \gg_\rho \equiv \ll test_2 \gg_\rho)\_v_1.handler.\_body\ in \\ \quad\quad \varpi(\_v_1.handler.\_uregevt); \\ \quad\quad let\ \_rvureg = [\ val = \varsigma(\_x)\_x.val\ ]\ in \\ \quad\quad let\ \_rvtemp = \varrho(\_v_1.handler.\_uregevt, [\ ] \equiv [\ ])\ (\ \not\varrho(\_rv); \not\varrho(\_rvureg.val)\ )\ \ in \\ \quad\quad\quad \_rvureg.val \Leftarrow \varsigma(\_x)\_rvtemp; \\ \quad\quad\quad [\ ]\end{array}$$

Figure 5.9: Translation of the Register for Updates Expression

$$\left(\begin{array}{l}\text{Where,}\\ \quad a = \#\textbf{\textit{param}}(\_v_1, handler)\\ \quad b = \#\textbf{\textit{param}}(\_v_2, event)\\ \quad p = \textbf{\textit{min}}(a, b)\end{array}\right)$$

$$\begin{array}{l}\ll \textbf{register}\ exp_1.handler\ \textbf{for event}\ exp_2.event\ \textbf{when}\ test_1 \equiv test_2 \gg_\rho \stackrel{\triangle}{=}\\ \quad let\ \_v_1 = \ll exp_1 \gg_\rho\ in\\ \quad let\ \_v_2 = \ll exp_2 \gg_\rho\ in\\ \quad let\ \_relay = [\ e = \varsigma(\_x)[\ ]\ ]\ in\\ \quad let\ \_reg_1 = \varrho(\_v_2.event, [\ ] \equiv [\ ])(\\ \qquad\qquad \textbf{\textit{copyp}}(\_v_1, handler, \_v_2, event, 1, p);\\ \qquad\qquad \textbf{\textit{copyp}}(\_v_1, handler, \_v_1, handler, p+1, a);\\ \qquad\qquad \_relay.e \Leftarrow \varsigma(\_x)[\ ]\\ \qquad\qquad)\ in\\ \qquad \varpi(\_relay.e);\\ \qquad \varpi(\_v_1.handler.\_uregevt);\\ \qquad let\ \_reg_2 = \varrho(\_relay.e, \ll test_1 \gg_\rho \equiv \ll test_2 \gg_\rho)\_v_1.handler.\_body\ in\\ \qquad\quad \varrho(\_v_1.handler.\_uregevt, [\ ] \equiv [\ ])(\natural(\_reg_1); \natural(\_reg_2));\\ \qquad\quad [\ ]\end{array}$$

Figure 5.10: Translation of the Register for Event Expression

$$\left(\begin{array}{l}\text{Where,}\\ \quad a = \#\textbf{\textit{param}}(\_v_1, handler)\\ \quad b = \#\textbf{\textit{param}}(\_v_2, method)\\ \quad p = \textbf{\textit{min}}(a, b)\\ \quad \rho' = \rho\{p_1 \leftarrow \_v_2.\_l_{j_1}\} \ldots \{p_k \leftarrow \_v_2.\_l_{j_k}\}\end{array}\right)$$

$$\begin{array}{l}\left\langle\!\!\left\langle\ \begin{array}{l}\textbf{register}\ exp_1.handler\ \ \textbf{for invocation of}\ \ exp_2.method(p_1, \ldots, p_k)\\ \qquad\qquad\qquad\qquad \textbf{when}\ test_1 \equiv test_2\end{array}\ \right\rangle\!\!\right\rangle_\rho \stackrel{\triangle}{=}\\ \quad let\ \_v_1 = \ll exp_1 \gg_\rho\ in\\ \quad let\ \_v_2 = \ll exp_2 \gg_\rho\ in\\ \quad let\ \_relay = [\ e = \varsigma(\_x)[\ ]\ ]\ in\\ \quad let\ \_reg_1 = \varrho(\_v_2.\_invoke_{method}, [\ ] \equiv [\ ])(\\ \qquad\qquad \textbf{\textit{copyp}}(\_v_1, handler, \_v_2, method, 1, p);\\ \qquad\qquad \textbf{\textit{copyp}}(\_v_1, handler, \_v_1, handler, p+1, a);\\ \qquad\qquad \_relay.e \Leftarrow \varsigma(\_x)[\ ]\\ \qquad\qquad)\ in\\ \qquad \varpi(\_relay.e);\\ \qquad \varpi(\_v_1.handler.\_uregevt);\\ \qquad let\ \_reg_2 = \varrho(\_relay.e, \ll test_1 \gg_{\rho'} \equiv \ll test_2 \gg_{\rho'})\_v_1.handler.\_body\ in\\ \qquad\quad \varrho(\_v_1.handler.\_uregevt, [\ ] \equiv [\ ])\natural(\_reg_1); \natural(\_reg_2)\\ \qquad\quad [\ ]\end{array}$$

Figure 5.11: Translation of the Register for Invocation Expression

$\ll$ **register** $exp_1.l_1$ **for update of** $exp_2.l_2 \gg_\rho \stackrel{\triangle}{=}$
$\quad \ll$ **register** $exp_1.l_1$ **for update of** $exp_2.l_2$**when Empty** $\equiv$ **Empty** $\gg_\rho$
$\ll$ **register** $exp_1.l_1$ **for event** $exp_2.l_2 \gg_\rho \stackrel{\triangle}{=}$
$\quad \ll$ **register** $exp_1.l_1$ **for event** $exp_2.l_2$**when Empty** $\equiv$ **Empty** $\gg_\rho$
$\ll$ **register** $exp_1.l_1$ **for invocation of** $exp_2.l_2 \gg_\rho \stackrel{\triangle}{=}$
$\quad \ll$ **register** $exp_1.l_1$ **for invocation of** $exp_2.l_2$**when Empty** $\equiv$ **Empty** $\gg_\rho$
$\ll$ **Empty** $\gg_\rho \stackrel{\triangle}{=} \ll$ **new** *Empty* $\gg_\rho$
where *Empty* is a class without any members

Figure 5.12: Syntactic Sugars for Registration without the When Clause

that allows for conditional event announcement. The semantics of leaving off the **when** clause can be expressed as a syntactic sugar where the condition will always be satisfied. This syntactic sugar translation is shown in Figure 5.12. To ensure that the condition is always satisfied, empty objects are used. Since the definition for $\equiv$, Definition 4.2.1 in Chapter 4, requires that each member of one object refers to the exact same part of the store as the member of the other object, empty objects, having no members, vacuously satisfy the constraints of the definition. The three forms register event handlers for explicit events, updates, and invocations.

The event registration translation for updates, Figure 5.9, first saves the values of the event object expression and the target object expression to avoid multiple evaluations of those expressions. Next, the event handler, $\_v_1.handler.\_body$, is registered for any changes in the event object's member of interest, $\_v_2.member$, and the result is saved as $\_rv$. Then, the unregister event for the handler, $\_v_1.handler.\_uregevt$, is published. The next two let expressions ensure that when the unregister event is triggered, the registration is cleaned up via the unpublish of $\_rv$ and the registration for the unregistration event is also cleaned up. This is accomplished by setting up a dummy object, $\_rvureg$ in this case, that will eventually hold the result of the unregister event registration. Second, the unregistration event is registered and the result is stored in a temporary variable, $\_rvtemp$. Third, the dummy object's member is set to return the value of the previous registration. Thus, the unpublish on the dummy object's member, $\_rvureg.val$, will unregister both the event handler and the unregister event handler. Finally, the empty object is used for the result of the registration.

Event registration for explicit events, Figure 5.10, is different from the registration for updates in that it must handle parameter passing. Also, the translation does not clean up after itself (no unregister event unregistration) as in the previous translation. However, since there is no semantic effect for unregistering a handler that has already been unregistered, this is of no consequence. This extra clean up was put in the first translation and left out of this and the next translation because these are more complicated than the first; thus, avoiding even more complexity in the other translations.

The parameter passing for explicit event registration is handled by using two event announcements. The first is the announcement of the actual event. The second announcement is a "relay" event that signals that the parameters are all in place for the event handler to use. Parameter passing is accomplished by setting all of the handlers formal parameter members to return the result of the event's actual parameter members. To allow for cases when the arity (number of parameters) of the event differs from the arity of the event handler, the translation fills as many of the formal parameters as it can with the actual parameters, up to the total number of formal parameters, noted as n here. If there are not enough actual parameters, the rest of the formal parameters are filled with infinite loops (to ensure that the typing will work out properly). The ***copyp*** and ***#param*** functions, Definitions 5.2.2 and 5.2.3, are defined to handle these details. The ***copyp*** function takes two $\varrho\varpi\varsigma$-calculus objects, two associated labels, and two integers that mark the beginning and the end of the parameters to be copied and generates a section of code that will copy parameters associated with the label from one object to the other between the values given. The ***#param*** function takes an Eventua object and a label for a method and returns the number of parameters associated with that method. These functions are defined formally below.

Definition 5.2.2 (*copyp*). Let ***copyp*** be a function mapping $\varrho\varpi\varsigma$-calculus objects a and b, labels s and t, integers *start* and *stop* to a $\varrho\varpi\varsigma$-calculus expression in the following fashion:

$$\textbf{\textit{copyp}}(a, s, b, t, start, stop) = \begin{cases} \left.\begin{array}{c} a.\_s_{start} \Leftarrow \varsigma(\_x)b.\_t_{start}; \\ \vdots \\ a.\_s_{stop} \Leftarrow \varsigma(\_x)b.\_t_{stop} \end{array}\right\} & start \leq stop \\ [\,] & start > stop \end{cases}$$

Definition 5.2.3 (*#param*). Let ***#param*** be a function mapping a $\varrho\varpi\varsigma$-calculus object a and a label l to an integer n in the following fashion:

$\textbf{\textit{\#param}}(a, l) = n$ Where a has $\_l_1, \_l_2, \ldots, \_l_n$ and there is no $\_l_i$ such that $i > n$.

The translation for invocation registration translations is similar to that of explicit event registration, except that the parameters to the handler in the invocation case come from the parameters that were passed into the method. The only other difference is that in the invocation case, the event to be handled is the invoke event, $\_invoke_{l_j}$, instead of the usual, l_j.

The translation of expressions aside from register and announce expressions are relatively straightforward. Register and announce are more complex since some of their functionality, e.g., parameter passing, event handlers that are tied to objects, etc., are not directly supported in the $\varrho\varpi\varsigma$-calculus. Given the lack of support for these features in the calculus, the complexity should not seem unreasonable, that is, the complexity of the translation does not suggest that the calculus is not fit for defining the semantics of Eventua since the complexity comes largely from the details of simulating Eventua's non-essential (but more realistic) features.

In general, all parameter passing is done using a by-value style. That is, the actual parameters are evaluated and their results are passed. By-value parameter passing is the safest parameter passing implementation for Eventua because it will never be known how many times a parameter has been referenced or possibly updated before it is given to the next handler or method. This means that if parameters passed in using a by-reference or by-name style, they would have the potential of being altered before some of the handlers or the method (if it is a method invocation) would get the parameters. This would violate the common notion of what parameter passing usually accomplishes, a value is passed in and the *same* value is accessed by the receiving party (event handler or method), regardless of the passing style.

Handler unregistration may be somewhat problematic since its translation to the $\varrho\varpi\varsigma$-calculus only causes an event, *_uregevt* to be announced. The actual unregistration is accomplished by handlers of the *_uregevt* event that are created in the translation of **register**. An alternative approach to this would be to have a member in the handler's $\varrho\varpi\varsigma$-calculus object that would save the registration id that results in the handler's actual registration. The problem with this, however, is that there is no restriction on how many events to which the handler can be registered. Therefore, we would need to save an entire list of registration ids for each registered event. Here, instead of making an explicit list of ids, we simply register an "unregistration" handler for the *_uregevt* event. Another concern with this approach, is that when we unregister a handler, it unregisters the handler from *all* its associated events, which may cause the unregistration of more than what was intended or known about. While it is almost possible to allow for unregistration of individual registrations based on both the handler identifier and an associated event, rather than only the handler identifier, this will not work since the handler may be registered for the same event twice. For example, objects A and B both register the same handler for the same event. In such a case, it would be necessary to ensure that both A's and B's registrations are undone. This may seem to be better than unregistering the handler from all its associated events, however the general expectation for unregister, if it acted on an event-event handler pair, would be that only one handler would be unregistered; an assumption that would occasionally be violated. With the current approach, there is no danger of misunderstanding — *all* the registrations for the given handler will be undone.

5.3 Examples of the Eventua Language

The purpose of the following examples is to show how Eventua works and that Eventua represents a practical, albeit restricted, object-oriented language that includes support for events. The examples are also constructed to motivate the usefulness of some of the features of the language. We start with a simple example to show how Eventua works. Then, we construct an example that illustrates why events tend to simplify some of the problems around software integration by developing a series of classes that model a directed graph that is similar to the example given in [SN92]. Again, the examples use a ; b in place of **let** x = a **in** b where $x \notin FV(b)$ for the sake of clarity.

5.3.1 *Mail Announcement Example*

To illustrate the way a simple Eventua program might work, we create an example of a mail arrival announcement program. A real mail arrival announcement program would require support for clients and servers. However, the basic idea of the example can be grasped without dealing with this level of detail. For the sake of simplicity, we presume that an object, *out*, is provided for text output and that character strings are handled in an expected manner.

```
class  mailhandler  begin
      checkmail = method() : Object()
            announce  self.mail()
      mail = event()
end
class  mailclient  begin
      mailcall = handler()
            out.print("Check your mail")
end
      let  client = new mailclient  in
      let  handlr = new mailhandler  in
            register  client.mailcall  for event handlr.mail;
            publish  event handlr.mail;
            handlr.checkmail()
```

The *mailhandler* class acts as the server in this example and the *mailclient* class is the client. The body of the program simply consists of the creation of the client and server, the registration of the client's handler for the server's event, and the publication of the server's event. At this point the program should let the server periodically check for mail to arrive and announce its *mail* message if it finds some mail. The example simulates this series by simply calling *checkmail*, which, in turn, announces that mail has arrived. To make the example more useful, a user parameter could be added to the event that would allow subscribers to tell if the mail announcement is for them or not.

5.3.2 *Graph Relationship Example*

This example implements part of the graph example given in [SN92]. Specifically, classes for edges and vertices are implemented which show how the graph relation $(v_1, v_2) \in \text{ES} \Rightarrow v_1, v_2 \in \text{VS}$, is maintained by using events. We later expand the example to include a counter class, N, that keeps track of the number of edges in ES. We will demonstrate that adding this functionality to the program will not require any changes to the ES class.

For the example, we have a limited form of natural numbers, **Nat**. We also have a Boolean type, **Bool**, along with constants, **True** and **False**. These types are defined in Appendix B. The **Bool** type is parameterized to avoid multiple definitions of similar types. The parameter for **Bool** is used to ensure that the if-then-else sugar is typed correctly.

We will use a limited form of arrays. The type for an array is also parameterized and will be noted as, **Array**(T) where T is the type of the elements of the array. For the sake of simplifying the examples, the size of the arrays and the subset of natural numbers will be left off of their respective types. Technically, the size for these types are important because different sizes will have different types. However, for the sake of clarity and simplicity, only one size will be used throughout the examples, thus no type ambiguity will be introduced by leaving size distinction off of the types.

The examples will also rely on some syntactic sugars, i.e., syntactic translations that make programs easier to read. The sugars include an if-then-else construct; a while-do loop; true and false literals; **Nat** literals, e.g., 1, 2, 3; '+' and '−'; a comparison operator '=='; and array dereferncing and updates '[n]'. Refer to Appendix C for the details on these sugars.

```
class  VS  begin
      answer = private field : Bool(Empty)
      count = private field : Nat := 0
```

```
verts = private field : Array(Nat)
insert = method(vertex : Nat) : Empty
        if not self.isMember(vertex) then
                private.verts[ private.count ] := vertex
                private.count := private.count + 1;
        else
                new Empty;
        new Empty
isMember = method(vertex : Nat) : Bool(Empty)
        let  curpos = 0  in
                private.answer := false
                while not curpos == private.count do
                        if private.verts[curpos] == vertex then
                                private.answer := true;
                                new Empty
                        else
                                new Empty;
                private.answer
        newEdge = handler(vert1 : Nat, vert2 : Nat)
                self.insert(vert1);
                self.insert(vert2)
end
class  ES  begin
        answer = private field : Bool(Empty)
        count = private field : Nat := 0
        first = private field : Array(Nat)
        second = private field : Array(Nat)
        edgeAdded = event(vert1 : Nat, vert2 : Nat)
        insert = method(vert1 : Nat, vert2 : Nat) : Empty
                if not self.isMember(vert1, vert2) then
                        private.first[private.count] := vert1;
                        private.second[private.count] := vert2;
                        private.count := private.count + 1;
                        announce  self.edgeAdded(vert1, vert2)
                else
                        new Empty;
                new Empty
        isMember = method(vert1 : Nat, vert2 : Nat) : Bool(Empty)
                let  curpos = 0  in
                        private.answer := false;
                        while not curpos == private.count do
                                if private.first[ curpos ] == vert1 and
                                   private.second[ curpos ] == vert2 then
                                        private.answer := true;
                                        new Empty
                                else
                                        new Empty;
                        private.answer
        end
                let  es = new ES  in
                        publish  event  es.edgeAdded;
                        let  vs = new VS  in
                                register  vs.newEdge  for event  es.edgeAdded;
                                es.insert(5, 6)
```

The example starts by defining classes for a vertex set, VS, and a edge set, ES. Both classes are very similar to each other. The insert method in VS takes a number and includes it as a member, if it was not already included. The isMember method takes a number and returns **true** if that number has been included as a vertex. This is accomplished by simply walking the verts array to find the vertex using the **while-do** loop. The event handler for VS, newEdge, simply includes the vertices for the edge if necessary. The insert and isMember methods for ES are similar to VS's methods with two differences. First, the ES methods take two integers, representing an edge. Second, ES has a edgeAdded event member and the insert method announces this event.

The body of the example first creates a new instance of ES, es, and then publishes its edgeAdded event. Next, a new instance of VS, vs, is created and vs's newEdge event handler is registered for es's edgeAdded event. Finally, es's insert method is invoked and given (5, 6) as its edge. This insertion will first add the edge to the edge set. It will then announce the edgeAdded event with the two vertices for its parameters. This will then trigger the newEdge event handler for vs, adding both vertices to the vertex set. Therefore, if we did vs.isMember(5) as the last step, the result of the program would be **true**, even though we never *explicitly* invoked vs.insert(5).

Given the above program, if we wanted to count the number of edges we could leave the code exactly the way it is now and add a counter class with a handler to handle *edgeAdded* events. We also need to add code that creates a new counter object and registers its handler for the *edgeAdded* event. These modifications are given below: first, the counting class, N, then the new body of the program.

```
class N begin
    inchandle = handler()
        self.count := self.count + 1
    count = field : Nat := 0
end
    let es = new ES in
        publish event es.edgeAdded;
        let vs = new VS in
        let edgeCount = new N in
            register vs.newEdge for event es.edgeAdded;
            register edgeCount.incHandle for event es.edgeAdded;
            es.insert(5, 6)
```

The only differences in the new program body from the old are a new instance of N, edgeCount and a new event registration. So, if, at the end of this new body, we were to evaluate edgeAdded.count, the result would be 1. This, again, despite the fact that the program did not *explicitly* tell edgeCount to increment its counter.

5.3.3 *Maintaining a Count Relationship for the Vertex Set*

So far, setting up our programs to implicitly maintain our set relationship and our count relationship for edges has been relatively straightforward. Instances of ES have an edgeAdded event for registering handlers. Therefore, maintaining relationships that are affected by the addition of an edge is quite easy to do. Since adding a vertex did not require that a new edge be added, no event was created for adding vertices. Granted, it would be easy enough to simply add an event to VS and announce it every time there a new vertex is added.

In general, however, it can be difficult to add an event after the object has been implemented, much in the same way it is difficult to add in error handling to a "complete" implementation. Further, a client implementer in need of a particular kind of event announcement should not be forced to change the implementation of the class they are using to get the results they require. For these reasons it is desirable to use the other, i.e., implicit, event publication and registration forms. They are *implicit events* because they are not explicitly declared as events (not to be confused with implicit invocation, accomplished by both explicit and implicit events and their registered handlers). In order to add a vertex counter to our example, we will use implicit events to cover for the absence of an explicit vertexAdded event. Specifically, we will register the counter's incHandle event to handle the case where the insert method is invoked. Here is the new program body which counts both edges and vertices using events.

```
let es = new ES in
    publish event es.edgeAdded;
    let vs = new VS in
    let edgeCount = new N in
    let vertexCount = new N in
        publish invocation of vs.insert;
        register vs.newEdge for event es.edgeAdded;
        register edgeCount.incHandle for event es.edgeAdded;
        register vertexCount.incHandle for invocation of vs.insert(vertex)
                    when vs.isMember(vertex) ≡ false;
        es.insert(5, 6)
```

The only change from the old program body to this one is the addition of the vertexCount declaration and the conditional registration of its handler for when vs's insert method is invoked. In this case, the condition under which the handler will be evaluated is when the vertex that is being inserted is not already in the vertex set. Note that the arity of insert is one, i.e., insert takes one argument, but the arity of the event handler is 0. This difference is handled in the semantics by simply ignoring the parameter in the event handler. Specifically, the translation will only use up to the number of parameters required by the handler. Since the conditional expressions (unlike the handler) will be translated within the context of the registration itself, it will have direct access to the "event side" parameters. Thus, the parameter "hand-off" between the event and the handler does not effect the conditional expressions. This is shown in Figure 5.11 with the ρ' translation environment for the conditional expressions; making use of the event's parameter member.

For the sake of clarity in the example, the conditional part of the event registration is left technically incorrect, as the equivalence condition, ≡, would never be satisfied. This is due to the fact that **false**, defined in appendix A, always creates a new object. Since new objects always have unique store locations, the equivalence test, which compares objects' store locations, will fail. The correct, albeit convoluted, way to do this is to define two constants with identical types, e.g., allow and disallow. Then, wrap the isMember invocation in a if-then-else which returns disallow if isMember returns a true value and an allow when it returns a false value, as done here:

```
when
    if vs.isMember(vertex) then
        disallow
    else
        allow
≡
    allow
```

5.4 Eventua Types: Rules and Translation to the $\varrho\varpi\varsigma$-calculus

We have shown in the previous sections that the $\varrho\varpi\varsigma$-calculus is useful for defining a practical core language such as Eventua. Now, we will show that the $\varrho\varpi\varsigma$-calculus type system is useful for proving that the type system for Eventua is sound using type translations. Similar to the syntax and semantics for Eventua, the rules for the type system are defined, then the translation of those types to $\varrho\varpi\varsigma$-calculus types are defined. Once we have both a syntactic translation and a type translation, we can use them to show that the properties of the $\varrho\varpi\varsigma$-calculus type system holds for Eventua's type system, if the two translations remain faithful. That is, if the translated Eventua syntax is shown to have the same type as the translated Eventua type, then the properties of the $\varrho\varpi\varsigma$-calculus type system will apply to the Eventua type system. Proving that the translations are faithful, e.g., type preserving, is left for future work.

5.4.1 *Eventua Type Rules*

As with the type rules for the $\varrho\varpi\varsigma$-calculus, Eventua's type rules should ensure that the every legal Eventua construct that can be typed has well-defined semantics. That is, a derivation of the semantics of a Eventua

$$\text{[Env } \phi\text{]} \quad \frac{}{\phi \vdash \diamond} \qquad \text{[Env x]} \quad \frac{E \vdash A}{E, x : A \vdash \diamond} \qquad \text{[Env Self]} \quad \frac{E \vdash A}{E, \mathbf{self} : A \vdash \diamond}$$

$$\text{[Type } \textit{Top}\text{]} \quad \frac{E \vdash \diamond}{E \vdash \mathbf{Top}} \qquad \text{[Type } K\text{]} \quad \frac{E \vdash \diamond}{E \vdash K} \qquad \text{[Type Class]} \quad \frac{E \vdash A}{E \vdash \mathbf{Class}(A)}$$

$$\text{[Type Object]} \quad \frac{E \vdash B_i \quad \forall i \in i..n}{E \vdash \mathbf{Object}(l_i : B_i \ ^{i \in 1..n})} \qquad \text{[Type Field]} \quad \frac{E \vdash A}{E \vdash \mathbf{Field}(A)}$$

$$\text{[Type Private]} \quad \frac{E \vdash A}{E \vdash \mathbf{Private}(A)} \qquad \text{[Type Method]} \quad \frac{E \vdash A \quad E \vdash T_i \quad \forall i \in 1..n}{E \vdash \mathbf{Method}(A; T_i \ ^{i \in 1..n})}$$

$$\text{[Type Event]} \quad \frac{E \vdash T_i \quad \forall i \in 1..n}{E \vdash \mathbf{Event}(T_i \ ^{i \in 1..n})} \qquad \text{[Type Handler]} \quad \frac{E \vdash T_i \quad \forall i \in 1..n}{E \vdash \mathbf{Handler}(T_i \ ^{i in 1..n})}$$

Figure 5.13: Eventua Type Environment and Well-Formedness Rules

syntactic construct must exist if the construct can be typed. First, the type environment and environment well-formedness rules are defined in Figure 5.13. Then, the rest of the rules are defined top down, starting with a rule that assigns a type to an entire program, the [Val Program] rule, and working down to rules for individual declarations and expressions. Figure 5.14 contains the [Val Program] rule as well as all the rules for class declarations.

The type environment and well-formedness rules ensure that types are not malformed and that the type environments are correct, similar to Figure 4.11 for the $\varrho\varpi\varsigma$-calculus. The [Env x] rule works to unwind the type environment where [Env ϕ] is the base case for that rule, stating that an empty type environment is well-formed. [Type K] handles constant types, for example, integer types. There are no constant types defined for Eventua in our formalism — the rule was included for completeness. The [Type *Top*] rule handles the **Top** type when it comes up. The rest of the rules ensure that all the types included in class, object, and class member types are also all well-formed.

The [Val Program] rule states that a program has a type, A, if every class in the program has a valid type, the C_k clauses and, given an environment with all the classes and their respective types, the resulting type of the body of the program is A.

To prove that a class has a valid type, the [Val Class] rule is applied. This rule states that a class has a type, **Class**(A), where A is the object type that the class defines. Further, the rule ensures that each declaration in the class has a valid type, B_i in the figure, corresponding to the definition of A. In order to allow the members of the class to reference its own class and instance, the type environment includes proper typings for the class being defined and for *self*. That is, *self* and the class, *cl*, are assumed to have the type that we are attempting to prove that they have.

The rules for class declarations are all very similar. The type judgments for [Val Field] and [Val Private] are simply based on the type given in the actual declaration and the judgment is valid just when the type environment, E, is well-formed. [Val Field Init] and [Val Private Init] are the same as their counterparts except that the initialization expression must have a valid type judgment that results in the same type as the declared type. Types for methods are based on the types of their parameters and result type. In this way, [Val Method] is similar in nature to the Init rules. Event types are also based on the types expected for parameters. Thus, [Val Event] must simply ensure that the type environment is well formed. Types for handlers are similar to that of events, except that the body of the handler must be checked to ensure that it has a valid typing, which is why the hypothesis of the [Val Handler] rule ensures that the type of the handler body is at least **Top**.

Registration expression typing rules are listed in Figure 5.15. These rules all ensure that the target of the

[Val Program] (where $cl_i = \textbf{class}\ cl_i\ \textbf{begin}\ decl_{i_j}\ ^{j\in 1..m_i}\ \textbf{end}$)

$$\frac{E,\ cl_k : C_k\ ^{k\in 1..(i-1)} \vdash cl_i : C_i \quad \forall i \in 1..n \quad E,\ cl_i : C_i\ ^{i\in 1..n} \vdash exp : A}{E \vdash cl_i\ ^{i\in 1..n}\ exp : A}$$

[Val Class] (where $decl_i = (l_i = \mathbf{x}_i)$, $\mathbf{x}_i$ is a class member declaration, and $A = \textbf{Object}(l_i : B_i\ ^{i\in 1..n})$)

$$\frac{E,\ \textbf{self} : A,\ cl : \textbf{Class}(A) \vdash decl_i : B_i \quad \forall i \in 1..n}{E \vdash \textbf{class}\ cl\ \textbf{begin}\ decl_i\ ^{i\in 1..n}\ \textbf{end} : \textbf{Class}(A)}$$

[Val Field]

$$\frac{E \vdash \textbf{Field}(A)}{E \vdash (l = \ \textbf{field}\ : A) : \textbf{Field}(A)}$$

[Val Field Init]

$$\frac{E \vdash exp : A}{E \vdash (l = \ \textbf{field}\ : A := exp) : \textbf{Field}(A)}$$

[Val Private]

$$\frac{E \vdash \textbf{Private}(A)}{E \vdash (l = \ \textbf{private field}\ : A) : \textbf{Private}(A)}$$

[Val Private Init]

$$\frac{E \vdash exp : A}{E \vdash (l = \ \textbf{private field}\ : A := exp) : \textbf{Private}(A)}$$

[Val Method]

$$\frac{E, x_1 : T_1, \ldots, x_n : T_n \vdash exp : A \quad E \vdash \textbf{Method}(A; T_1, T_2, \ldots, T_n)}{E \vdash (l = \ \textbf{method}(x_1 : T_1, \ldots, x_n : T_n) : A\ exp) : \textbf{Method}(A; T_1, T_2, \ldots, T_n)}$$

[Val Event]

$$\frac{E \vdash \textbf{Event}(T_1, T_2, \ldots, T_n)}{E \vdash (l = \ \textbf{event}(x_1 : T_1, \ldots, x_n : T_n)) : \textbf{Event}(T_1, T_2, \ldots, T_n)}$$

[Val Handler]

$$\frac{E, x_1 : T_1, \ldots, x_n : T_n \vdash exp : \textbf{Top} \quad E \vdash \textbf{Handler}(T_1, T_2, \ldots, T_n)}{E \vdash l = \ \textbf{handler}(x_1 : T_1, \ldots, x_n : T_n)\ exp : \textbf{Handler}(T_1, T_2, \ldots, T_n)}$$

Figure 5.14: Eventua Type Rules for Classes

[Val Register Event] (where $j_1 \in 1..n_1$, $j_2 \in 1..n_2$, $m \geq n$,
$B_{j_1} = \mathbf{Handler}(T_1, T_2, \ldots, T_n)$,
and $C_{j_2} = \mathbf{Event}(T'_1,\ T'_2,\ \ldots,\ T'_m)$)

$$\frac{E \vdash exp_1 : \mathbf{Object}(l_i : B_i\ ^{i \in 1..n_1}) \quad E \vdash exp_2 : \mathbf{Object}(l'_i : C_i\ ^{i \in 1..n_2}) \quad T'_k <: T_k \quad \forall k \in 1..n \quad E \vdash exp_3 : \mathbf{Top} \quad E \vdash exp_4 : \mathbf{Top}}{E \vdash \mathbf{register}\ exp_1.l_{j_1}\ \mathbf{for\ event}\ exp_2.l'_{j_2}\ \mathbf{when}\ exp_3 \equiv exp_4 : \mathbf{Object}()}$$

[Val Register Update] (where $j_1 \in 1..n_1$, $j_2 \in 1..n_2$,
and $B_{j_1} = \mathbf{Handler}(T_1, T_2, \ldots, T_n)$)

$$\frac{E \vdash exp_1 : \mathbf{Object}(l_i : B_i\ ^{i \in 1..n_1}) \quad E \vdash exp_2 : \mathbf{Object}(l'_i : C_i\ ^{i \in 1..n_2}) \quad E \vdash exp_3 : \mathbf{Top} \quad E \vdash exp_4 : \mathbf{Top}}{E \vdash \mathbf{register}\ exp_1.l_{j_1}\ \mathbf{for\ update\ of}\ exp_2.l'_{j_2}\ \mathbf{when}\ exp_3 \equiv exp_4 : \mathbf{Object}()}$$

[Val Register Invocation] (where $j_1 \in 1..n_1$, $j_2 \in 1..n_2$, $m \geq n$, $k = m$
$B_{j_1} = \mathbf{Handler}(T_1, T_2, \ldots, T_n)$,
$C_{j_2} = \mathbf{Method}(V; T'_1,\ T'_2,\ \ldots,\ T'_m)$, and
$E' = E, p_1 : T'_1, \ldots, p_m : T'_m$)

$$\frac{E \vdash exp_1 : \mathbf{Object}(l_i : B_i\ ^{i \in 1..n_1}) \quad E \vdash exp_2 : \mathbf{Object}(l'_i : C_i\ ^{i \in 1..n_2}) \quad T'_h <: T_h \quad \forall h \in 1..n \quad E' \vdash exp_3 : \mathbf{Top} \quad E' \vdash exp_4 : \mathbf{Top}}{E \vdash \mathbf{register}\ exp_1.l_{j_1}\ \mathbf{for\ invocation\ of}\ exp_2.l'_{j_2}(p_1, \ldots, p_k)\ \mathbf{when}\ exp_3 \equiv exp_4 : \mathbf{Object}()}$$

Figure 5.15: Eventua Type Rules for Registration Expressions

registration, the handler, is, in fact, a handler. They also ensure the basic type structure is in place: the "trigger" and the "target" are both valid members of objects; the types for the conditional parts, expressions three and four, are well-defined; and that the parameter passing, if there is any, will maintain type correctness. The parameter passing checks ensure that the "trigger" will be expecting a subtype of what its "target", or handler, expects and that there are no parameter passing underruns, that is, the "trigger" expects at least as many arguments as the "target" is expecting. This prevents the translation of the actual event announcements, Figure 5.10 and Figure 5.11, from filling in any parameter passing gaps with infinite loops (since there would be no parameter passing gaps).

The [Val Register Event] rule deviates from the other rules by ensuring that its "trigger" is an **Event** typed member of an object. Likewise, the [Val Register Invocation] rule, ensures that the "trigger" has a **Method** type. These additional constraints on the triggers ensure that translations will be type safe by not allowing them to attempt to access members of objects that do not exist. On the other hand, the [Val Register Update] rule has no need for additional constraints for the trigger because update registrations neither propagate parameters nor require special members that, for example, provide the actual event trigger in the $\varrho\varpi\varsigma$-calculus.

The type rules for expressions that operate on objects, for example, assignment, field selection, etc., are listed in Figure 5.16. The [Val Send] rule forces the types of the arguments to the method to be properly typed with respect to the method's declaration and that the method called is actually a member of the object. The select and assign rules ([Val Field Select], [Val Assign], [Val Private Select], [Val Private Assign]) are similar to the [Val Send] rule, except, of course, the parameter type checking. The assign rules differ in that the expression being assigned to the field is required to be of the proper type with respect to the type of the object. Also, the result of the assignment is the target object, thus the type of an assignment expression is the same as the type for the object.

The type system for Eventua allows for subtyping as shown in Figure 5.17. The [Sub Reflexive] rule allows types that are exactly the same to be subtypes of each other while the [Sub Transitive] rule states the typical transitive property rule for relations: when A is a subclass of B and B is a subclass of C, then A is a subclass of C. The [Sub Top] rule establishes that type Top is a supertype to every type in the Eventua type system.

The rule that defines subtypes for Eventua is the [Sub Object] rule. This rule states that an object is a

[Val Send] (where $j \in 1..m$, $B_j = \mathbf{Method}(A; T_1, T_2, \ldots, T_n)$)

$$\frac{E \vdash a : \mathbf{Object}(l_k : B_k \ ^{k \in 1..m}) \quad E \vdash exp_i : T_i \quad \forall i \in 1..n}{E \vdash a.l_j(exp_1, \ldots, exp_n) : A}$$

[Val Field Select] (where $j \in 1..n$, $B_j = \mathbf{Field}(A)$)

$$\frac{E \vdash a : \mathbf{Object}(l_i : B_i \ ^{i \in 1..n})}{E \vdash a.l_j : A}$$

[Val Assign] (where $j \in 1..n$, $A = \mathbf{Object}(l_i : B_i \ ^{i \in 1..n})$, and $B_j = \mathbf{Field}(B)$)

$$\frac{E \vdash exp_1 : A \quad E \vdash exp_2 : B}{E \vdash exp_1.l_j \ := \ exp_2 : A}$$

[Val Private Select] (where $j \in 1..n$, and $B_j = \mathbf{Private}(A)$)

$$\frac{E,\ \mathbf{self} : \mathbf{Object}(l_i : B_i \ ^{i \in 1..n}),\ E' \vdash \diamond}{E,\ \mathbf{self} : \mathbf{Object}(l_i : B_i \ ^{i \in 1..n}),\ E' \vdash \mathbf{private}.l_j : A}$$

[Val Private Assign] (where $j \in 1..n$ and $B_j = \mathbf{Private}(B)$)

$$\frac{E,\ \mathbf{self} : \mathbf{Object}(l_i : B_i \ ^{i \in 1..n}),\ E' \vdash exp : B}{E,\ \mathbf{self} : \mathbf{Object}(l_i : B_i \ ^{i \in 1..n}),\ E' \vdash \mathbf{private}.l_j \ := \ exp : \mathbf{Object}(l_i : B_i \ ^{i \in 1..n})}$$

Figure 5.16: Eventua Type Rules for Object Manipulation Expressions

[Sub Reflexive]

$$\frac{E \vdash A}{E \vdash A <: A}$$

[Sub Transitive]

$$\frac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C}$$

[Sub Top] (where $A = \mathbf{Object}(l_i : B_i \ ^{i \in 1..n})$)

$$\frac{E \vdash A}{E \vdash A <: Top}$$

[Sub Object]

$$\frac{E \vdash B_i \quad \forall i \in 1..n+m}{E \vdash \mathbf{Object}(l_i : B_i \ ^{i \in 1..(n+m)}) <: \mathbf{Object}(l_i : B_i \ ^{i \in 1..n})}$$

[Val Subsumption]

$$\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$$

Figure 5.17: Eventua Subtyping Rules

[Val Publish Event] (where $A = \mathbf{Object}(l_i : B_i\ ^{i\in 1..n})$, $j \in 1..n$, and $B_j = \mathbf{Event}(T_1, T_2, \ldots, T_n)$)

$$\frac{E \vdash exp : A}{E \vdash \mathbf{publish\ event}\ exp.l_j : \mathbf{Object}()}$$

[Val Publish Member] (where $A = \mathbf{Object}(l_i : B_i\ ^{i\in 1..n})$, $j \in 1..n$, and $B_j = \mathbf{Method}(V; T_1, T_2, \ldots, T_n)$, or $B_j = \mathbf{Field}(V)$)

$$\frac{E \vdash exp : A}{E \vdash \mathbf{publish\ update\ of}\ exp.l_j : \mathbf{Object}()}$$

[Val Publish Invocation] (where $A = \mathbf{Object}(l_i : B_i\ ^{i\in 1..n})$, $j \in 1..n$, and $B_j = \mathbf{Method}(V; T_1, T_2, \ldots, T_n)$)

$$\frac{E \vdash exp : A}{E \vdash \mathbf{publish\ invocation\ of}\ exp.l_j : \mathbf{Object}()}$$

[Val Unpublish Event] (where $A = \mathbf{Object}(l_i : B_i\ ^{i\in 1..n})$, $j \in 1..n$, and $B_j = \mathbf{Event}(T_1, T_2, \ldots, T_n)$)

$$\frac{E \vdash exp : A}{E \vdash \mathbf{unpublish\ event}\ exp.l_j : \mathbf{Object}()}$$

[Val Unpublish Member] (where $A = \mathbf{Object}(l_i : B_i\ ^{i\in 1..n})$, $j \in 1..n$, and $B_j = \mathbf{Method}(V; T_1, T_2, \ldots, T_n)$ or $B_j = \mathbf{Field}(V)$)

$$\frac{E \vdash exp : A}{E \vdash \mathbf{unpublish\ update\ of}\ exp.l_j : \mathbf{Object}()}$$

[Val Unpublish Invocation] (where $A = \mathbf{Object}(l_i : B_i\ ^{i\in 1..n})$, $j \in 1..n$, and $B_j = \mathbf{Method}(V; T_1, T_2, \ldots, T_n)$)

$$\frac{E \vdash exp : A}{E \vdash \mathbf{unpublish\ invocation\ of}\ exp.l_j : \mathbf{Object}()}$$

Figure 5.18: Eventua Type Rules for Publish and Unpublish

subtype of another object when it has the same members as the super type plus potentially other members. This kind of subtyping is often referred to as width subtyping. The actual application of subtyping comes from the [Sub Subsumption] rule. This rule states that a type can be substituted for another type if it is a supertype of the original type, that is, a supertype can *subsume* its subtypes.

The type rules for the various forms of publish and unpublish expressions, in Figure 5.18, are all very similar. All the rules ensure that the empty object is the result and they all check the targets to ensure that they are of the appropriate type in order to allow the desired effect of the expression.

The last set of type rules, Figure 5.19, covers the rest of the expressions in Eventua. The [Val Self] rule defines the type for self to be the type of the object that self denotes. [Val x] is the rule that provides type information for denoted values given the current environment. [Val Let] defines the type of an Eventua let expression. It ensures that the type for the first expression (the let part) is sound and that the type of the second expression is the same type as the result of the entire let expression. [Val New] ensures that the new object that is the result of the **new** expression is the expected type given the type of the class object that creates the new object. [Val Announce] is the result of an announcement of an event, thus the resulting type is an empty object. However, [Val Announce] ensures that the target of the announcement matches the event announcement as well as any arguments that are passed as part of the event announcement. Finally, the [Val Unregister] rule ensures that we are actually unregistering an event handler and that the result of the unregister is the empty object type.

5.4.2 *Eventua Type Translation*

Now that we have a type system and a translation of the Eventua syntax into the calculus, the final step for a complete type preserving translation is to provide a type translation from the $\varrho\varpi\varsigma$-calculus to the Eventua

[Val Self]
$$\frac{E,\ \mathbf{self} : A,\ E' \vdash \diamond}{E,\ \mathbf{self} : A,\ E' \vdash \mathbf{self} : A}$$

[Val x]
$$\frac{E,\ x : A,\ E' \vdash \diamond}{E,\ x : A,\ E' \vdash x : A}$$

[Val Let]
$$\frac{E \vdash exp_1 : B \quad E,\ x : B \vdash exp_2 : A}{E \vdash \mathbf{let}\ x = exp_1\ \mathbf{in}\ exp_2 : A}$$

[Val New]
$$\frac{E,\ l : \mathbf{Class}(A),\ E' \vdash \diamond}{E,\ l : \mathbf{Class}(A),\ E' \vdash \mathbf{new}\ l : A}$$

[Val Announce] (where $A = \mathbf{Object}(l_i : B_i\ ^{i \in 1..m})$, $j \in 1..m$, and $B_j = \mathbf{Event}(T_1, T_2, \ldots, T_n)$)
$$\frac{E \vdash exp : A \quad E \vdash exp_i : T_i \quad \forall i \in 1..n}{E \vdash \mathbf{announce}\ exp.l_j(exp_1, exp_2, \ldots, exp_n) : \mathbf{Object}()}$$

[Val Unregister] (where $A = \mathbf{Object}(l_i : B_i\ ^{i \in 1..n})$, $j \in 1..n$, and $B_j = \mathbf{Handler}(T_1, T_2, \ldots, T_n)$)
$$\frac{E \vdash exp : A}{E \vdash \mathbf{unregister}\ exp.l_j : \mathbf{Object}()}$$

Figure 5.19: Eventua Type Rules for Other Expressions

language. This is done through another series of figures in the same style as the translation from Eventua syntax to the $\varrho\varpi\varsigma$-calculus syntax.

The first figure, Figure 5.20, shows how the Eventua types for the environment are translated to the types in the $\varrho\varpi\varsigma$-calculus. The first translation is for Eventua classes. The translation shows that a class type, **Class**(), translates to a typing of a new object with a *_new* method that generates new objects of type, $\mathbf{Object}(l_i : B_i\ ^{i \in 1..n})$. As is demonstrated by the translation, the class typing encapsulates the type of the objects it produces. Next, the object type translation produces a $\varrho\varpi\varsigma$-calculus object type, containing the results of translating the types for each member of the Eventua object.

The object member translations take both the label and the type since some of the translations require knowledge of the label in order to generate the appropriate $\varrho\varpi\varsigma$-calculus equivalent type. The field type translation is straight across from Eventua to the calculus. The private field type translation is similar to the regular field translation except that the label must be changed to suit the access protection scheme in use in the syntax translation. Method type translations are broken up into several members for the calculus, one for the actual method member, one for each of the method's parameters, and one for the invocation event member that is created in the translation. The translation for the event member type is similar, except the type for the event itself is translated as the empty object type. The empty object type is used since the event is simply used to trigger event handlers in the calculus and events are triggered by assigning the event member a method that returns the empty object. Handlers are translated into objects in the calculus, thus the handler type is translated to an object type in the same way. Parameter types are handled in a similar fashion to that of events and methods. Finally, the type translations for the Top type and subtyping are defined as expected.

Next, the translation for the type environment is defined, Figure 5.21. The typing environment for Eventua is given a robust translation into typing environments for the $\varrho\varpi\varsigma$-calculus. The translation is in three parts. First, the empty environment for Eventua is translated into an empty environment for the calculus. Second, the environment translation is given for the Eventua self variable. This turns out to be just a $\varrho\varpi\varsigma$-calculus variable called, *self*. Third, a translation is provided for variables (besides self) in the environment.

The variable translation (third translation) is defined with respect to the translation environment, ρ. If the variable in the translation environment is translated by ρ to a member selection in the calculus, the variable is

$$\ll \mathbf{Class}(\mathbf{Object}(l_i : B_i \ ^{i\in 1..n})) \gg \stackrel{\triangle}{=} [\_new :\ll \mathbf{Object}(l_i : B_i \ ^{i\in 1..n}) \gg]$$

$$\ll \mathbf{Object}(l_i : B_i \ ^{i\in 1..n}) \gg \stackrel{\triangle}{=} [\ll l_i : B_i \gg^{i\in 1..n}]$$

$$\ll l : \mathbf{Field}(A) \gg \stackrel{\triangle}{=} l :\ll A \gg$$

$$\ll l : \mathbf{Private}(A) \gg \stackrel{\triangle}{=} \_priv_l :\ll A \gg$$

$$\ll l : \mathbf{Method}(A; T_1, T_2, \ldots, T_n) \gg \stackrel{\triangle}{=}$$
$$l : A, \_l_1 :\ll T_1 \gg, \_l_2 :\ll T_2 \gg, \ldots, \_l_n :\ll T_n \gg,$$
$$\_invoke_l : Top$$

$$\ll l : \mathbf{Event}(T_1, T_2, \ldots, T_n) \gg \stackrel{\triangle}{=} l : [\,], \_l_1 :\ll T_1 \gg, \_l_2 :\ll T_2 \gg, \ldots, \_l_n :\ll T_n \gg$$

$$\ll l : \mathbf{Handler}(T_1, T_2, \ldots, T_n) \gg \stackrel{\triangle}{=}$$
$$l : [\_body : Top, \_uregevt : Top],$$
$$\_l_1 :\ll T_1 \gg, \_l_2 :\ll T_2 \gg, \ldots, \_l_n :\ll T_n \gg$$

$$\ll \mathbf{Top} \gg \stackrel{\triangle}{=} Top$$

$$\ll K \gg \stackrel{\triangle}{=} K$$

$$\ll A <: B \gg \stackrel{\triangle}{=} \ll A \gg <:\ll B \gg$$

Figure 5.20: Translation of Eventua Types to $\varrho\varpi\varsigma$-calculus Types

$$\ll \phi \gg_\rho \stackrel{\triangle}{=} \phi$$

$$\ll E,\ \mathbf{self} : A \gg_\rho \stackrel{\triangle}{=} \ll E \gg_\rho,\ self :\ll A \gg$$

$$\ll E,\ x : A \gg_\rho \stackrel{\triangle}{=} \begin{cases} \ll E \gg_\rho & \text{if } x \leftarrow a.y \in \rho, \text{ for any } a \text{ and } y \\ \ll E \gg_\rho, \rho(x) :\ll A \gg & \text{otherwise} \end{cases}$$

Figure 5.21: Translation of Type Environments

completely removed from the translated type environment. Selections must not be allowed into the translated type judgment environment because well formed environments do not include type judgments. There will not be a problem with omitting these variables from the calculus environment since the selections that they translate into all use references to objects that will be in their environment. All other variables in the lookup environment will be left untouched and translated directly into the $\varrho\varpi\varsigma$-calculus.

5.5 Conclusion

We have defined Eventua as a core language with native events with the syntax and semantics one might expect with such a language. By virtue of Eventua's event mechanism, we have shown how events are useful for handling asynchronous occurrences of interest, such as getting new mail. We have also seen how Eventua can be used to aide with software integration by evolving a simple graph implementation example to include extra functionality. Finally, the usefulness of the $\varrho\varpi\varsigma$-calculus was demonstrated by showing how Eventua, as a practical core language, can be defined in terms of that calculus.

Chapter 6

ANALYSIS

The analysis of the calculus will cover both the completeness of the calculus and the usefulness of the calculus. The analysis of the completeness, or generality, of the $\varrho\varpi\varsigma$-calculus will be based primarily on the event framework outlined by Barrett, et. al. [BCTW96] showing how the calculus covers, or can cover, each part of their framework. Specifically, the constructs or set of constructs in the calculus that model the various objects defined in the framework will be evaluated based not only on whether the construct(s) satisfy the particular part of the framework, but also how expressively congruent the construct(s) are in relation to the framework, i.e., whether or not the translation between the calculus and the framework object is straightforward.

The analysis of the usefulness of the calculus will both follow from how general the calculus is and whether it has properties that are desirable for any formal programming language foundation, such as type soundness, etc.

6.1 Completeness of the Calculus

The Event-Based Integration (EBI) framework presented by Barrett, et. al. [BCTW96] has four different parts:

1. The informer which publishes and announces events.
2. The listener which registers for events and handles event announcement.
3. The registrar which tracks which events are published and which handlers are registered.
4. The router which informs the listeners when their events are announced.

The calculus will be analyzed against each of these parts and conclusions drawn about how well the calculus supports each part.

Any object in the $\varrho\varpi\varsigma$-calculus can model an informer or a listener. An object models an informer because methods of objects can be published and a method can be updated, which, in the calculus, models an event announcement. Thus, the informer and listener functionality are provided directly from the calculus, which implies that the calculus models these parts of the EBI framework very well. In other words, no complex expressions need to be built up in order to show that the calculus is capable of modeling these parts of the framework.

The basic role of the registrar, keeping track of the publications and registrations, can be modeled directly by the syntax and semantics of the calculus. Also, the basic role of the router, event delivery, is modeled by the calculus event semantics. On the other hand, there is no direct support in the calculus for the remaining registrar and router functionality defined in the EBI: registration of delivery constraints and event translation. Registration of this information is not directly supported because the calculus has no way, in general, to enforce these kinds of restrictions. However, use of the $\equiv$ part of the event registration syntax provides support for simple routing constraints, e.g., announce if and only if $c \equiv d$. So, while the basic event mechanism in the EBI framework, registration, publication, (simple) announcement, listeners, and informers, are cleanly handled by the calculus, some of the more "advanced" roles defined for these parts of the framework need to be represented by translations, i.e., combinations of calculus expressions.

Message transforming functions translate and filter events and their contents for delivery to listeners. Since the calculus does not allow any "content", e.g., parameters, with events, we give an example of how to simulate parameter passing within the calculus in Figure 6.1. The strategy demonstrated here is to provide an object

associated with each registration that knows how to get the intended contents of an event from the announcing object and update the appropriate methods in the listener with the contents "passed" by the informer. This is the general pattern that Eventua used to express its parameter passing mechanism in the calculus.

The example has two informers and a listener, where each informer has an event and some content and the listener has an event handler and a member for some content. After publishing the informers' events, we hook up the listener's event handler with the first informer via a helper object, *helper1*. The *ihdl* member of *helper1* acts as an intermediate event handler for a *informer1* event announcement; taking some of the content from *informer1* and giving it to the listener before it activates the listener's event handler. Then, similar steps are done to hook up the listener with the second informer. Finally, the updates at the end of the example sets the whole thing into motion. The example shows that accomplishing content passing in the calculus is a little obscure, but a language could be built using the calculus that could create these constructs without exposing these details to its users, similar to what was done for Eventua.

From the content passing example we see how translations to contents could be performed on the contents of events using helper objects. The helper can perform the translation before giving the content to the listener. However, the EBI framework allows for registration of an event first, *followed* by subsequent registrations of translations. It would be difficult to use the content passing pattern to accomplish translation registration directly. On the other hand, it would be possible to allow the event handlers in helper objects to be indirectly updated to reflect translations registered after the listener's original translation, building a chain of translations. This would allow for translations to be unregistered as well by removing a translation out of the chain. Another approach would be to insert or remove helpers between the listener and the original content passing helper, adjusting the event registrations as necessary. Given these approaches, we see that event content translation can be accomplished in the $\varrho\varpi\varsigma$-calculus at the cost of additional indirection.

The event delivery constraints defined in the EBI framework include constraints on event announcement timing and ordering. The calculus does not model timing constraints because it has no way to express timing. The calculus also does not enforce arbitrary policies of event announcement ordering, making event delivery ordering, that is, the order of the execution of the handlers of an event, difficult to accomplish. In fact, events in the calculus are "delivered immediately" to the appropriate registered handlers in an unspecified order. It is possible to overcome this by defining helper objects that handle announced events, thus there would be only one handler per "real" event, and then raise related events for the listeners in the appropriate order.

The $\varrho\varpi\varsigma$-calculus also has no way to directly guarantee the exact order of event triggering; thus, it does not directly model event trigger ordering policies. It would be possible, however, to ensure that one event occurs before another event before a handler is invoked. This could be done with a helper object that handles both events. The helper could then ignore any announcement of a second event that occurs before the first event is announced. An example of this is provided in Figure 6.2.

The resulting computations from the example on *listener* are as follows, after the let declarations and registrations:

- Nothing as a result of the first informer2.evt announcement.
- *listener.handler1* executed as a result of *informer1.evt* announcement.
- *listener.handler2* executed as a result of the next *informer2.evt* announcement.
- nothing as a result of the last *informer2.evt* announcement.

The notion of asynchronous messages, or parallel event handler execution, is not modeled by the calculus due to the fact that the calculus does not support parallel execution as part of its semantics. Thus, the calculus does not model asynchronous messages as defined in the EBI Framework.

Table 6.1 summarizes the degree to which the $\varrho\varpi\varsigma$-calculus models the EBI Framework. The table contains three possible categories for each framework concept: direct, indirect, and none. Concepts that fall into the direct category are concepts that have semantics that are very similar or exactly like an expression or basic construct in the $\varrho\varpi\varsigma$-calculus. Indirect means that the concept's semantics can be modeled with the calculus, however, there is not a corresponding syntactic construct in the calculus with the same semantics. None implies that there is no way to model that framework concept in the calculus.

$let\ informer1 = [evt = \varsigma(x)[\,],$
$\quad cntnt1 = \varsigma(x)x.cntnt1$
$\quad cntnt2 = \varsigma(x)x.cntnt2$
$\quad]\ in$
$let\ informer2 = [evt = \varsigma(x)[\,],$
$\quad cntnt1 = \varsigma(x)x.cntnt1$
$\quad]\ in$
$let\ listener = [evtcntnt = \varsigma(x)[\,],$
$\quad handler = \varsigma(x)x.useTheContent,$
$\quad \ldots$
$\quad]\ in$
$\varpi(informer1.evt);$
$\varpi(informer2.evt);$
—beginning of first registration construct
$let\ helper1 = [listener = \varsigma(x)\ listener$
$\quad informer = \varsigma(x)\ informer1$
$\quad levt = \varsigma(x)\ [\,]$
$\quad ihdl = \varsigma(x)x.listener.evtcntnt \Leftarrow x.informer.cntnt2;$
$\quad\quad x.levt \Leftarrow [\,]$
$\quad]$
$\varpi(helper1.levt);$
$\varrho(informer1.evt, [\,] \equiv [\,])helper1.ihdl;$
$\varrho(helper1.levt, [\,] \equiv [\,])listener.handler;$
—end of first registration construct
—beginning of second registration construct
$let\ helper2 = [listener = \varsigma(x)\ listener$
$\quad informer = \varsigma(x)\ informer2$
$\quad levt = \varsigma(x)\ [\,]$
$\quad ihdl = \varsigma(x)x.listener.evtcntnt \Leftarrow x.informer.cntnt1;$
$\quad\quad x.levt \Leftarrow [\,]$
$\quad]$
$\varpi(helper2.levt);$
$\varrho(informer2.evt, [\,] \equiv [\,])helper2.ihdl;$
$\varrho(helper2.levt, [\,] \equiv [\,])listener.handler;$
—end of second registration construct
$informer1.cntnt1 \Leftarrow \ldots;$
$informer1.cntnt2 \Leftarrow \ldots;$
$informer1.evt \Leftarrow [\,];$
$informer2.cntnt1 \Leftarrow \ldots;$
$informer2.evt \Leftarrow [\,]$

Figure 6.1: Example of Events with Associated Contents

$$
\begin{array}{l}
\mathit{let}\ \mathit{informer1} = [\mathit{evt} = \varsigma(x)[\,],]\ \mathit{in} \\
\quad \mathit{let}\ \mathit{informer2} = [\mathit{evt} = \varsigma(x)[\,],]\ \mathit{in} \\
\quad\quad \mathit{let}\ \mathit{listener} = [\mathit{handler1} = \varsigma(x)x.\ldots, \\
\quad\quad\quad\quad\quad\quad \mathit{handler2} = \varsigma(x)x.\ldots \\
\quad\quad\quad\quad\quad\quad]\ \mathit{in} \\
\quad\quad\quad \mathit{let}\ \mathit{helper} = [\mathit{ihdl1} = \varsigma(x)x.\mathit{state} \Leftarrow \varsigma(x)x.\mathit{true}, \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{ihdl2} = \varsigma(x)x.\mathit{state} \Leftarrow \varsigma(x)x.\mathit{false}; \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad x.\mathit{evt2} \Leftarrow [\,], \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{state} = \varsigma(x)x.\mathit{false}, \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{true} = \varsigma(x)x.\mathit{true}, \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{false} = \varsigma(x)[\,], \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{evt2} = \varsigma(x)[\,], \\
\quad\quad\quad\quad\quad\quad\quad\quad]\ \mathit{in} \\
\quad\quad\quad\quad \varpi(\mathit{informer1.evt}); \\
\quad\quad\quad\quad \varpi(\mathit{informer2.evt}); \\
\quad\quad\quad\quad \varpi(\mathit{helper.evt2}); \\
\quad\quad\quad\quad \mathit{let}\ \mathit{true} = [t = \varsigma(x)[\,]\,]\ \mathit{in} \\
\quad\quad\quad\quad\quad \mathit{helper.true} \Leftarrow \varsigma(x)\mathit{true}; \\
\quad\quad\quad\quad\quad \varrho(\mathit{informer1.evt}, [\,] \equiv [\,])\mathit{listener.handler1}; \\
\quad\quad\quad\quad\quad \varrho(\mathit{informer1.evt}, [\,] \equiv [\,])\mathit{helper.ihdl1}; \\
\quad\quad\quad\quad\quad \varrho(\mathit{informer2.evt}, \mathit{true} \equiv \mathit{helper.state})\mathit{helper.ihdl2}; \\
\quad\quad\quad\quad\quad \varrho(\mathit{helper.evt2}, [\,] \equiv [\,])\mathit{listener.handler2}; \\
\quad\quad\quad\quad\quad \mathit{informer2.evt} \Leftarrow \varsigma(x)[\,]; \\
\quad\quad\quad\quad\quad \mathit{informer1.evt} \Leftarrow \varsigma(x)[\,]; \\
\quad\quad\quad\quad\quad \mathit{informer2.evt} \Leftarrow \varsigma(x)[\,]; \\
\quad\quad\quad\quad\quad \mathit{informer2.evt} \Leftarrow \varsigma(x)[\,]
\end{array}
$$

Figure 6.2: Example of an Event Contingent on a Another Event

Table 6.1: Comparison of $\varrho\varpi\varsigma$-calculus with the EBI Framework [BCTW96]

EBI Framework Concept	**Page**	**Direct**	**Indirect**	**None**
Synchronous Messages	p. 388	√		
Asynchronous Messages	p. 388			√
Listener	p. 388	√		
Informer	p. 388	√		
Registrar	p. 389	√		
Router	p. 390	√	√	
Message Transforming Functions	p. 391		√	
Ordering Delivery Constraints	p. 392		√	
Timing Delivery Constraints	p. 392			√
Group	p. 393		√	

6.2 Usefulness of the $\varrho\varpi\varsigma$-calculus as a Formal Foundation

We have demonstrated, in several different ways, that the $\varrho\varpi\varsigma$-calculus is suitable as a formal foundation for languages that have built-in events. First, via comparison to the EBI framework, the calculus was shown to be general enough to easily model the properties of most event mechanisms. Second, the type system for the $\varrho\varpi\varsigma$-calculus has been shown to be sound (Chapter 4), demonstrating that $\varrho\varpi\varsigma$ expressions can be checked to ensure that reductions on individual expressions will not get stuck; a property that is useful for practical languages. Third, we have demonstrated that the calculus can be used as a formal basis for a core event language, Eventua (Chapter 5). Finally, we have argued and demonstrated that event semantics are beneficial for solving problems related to software integration and for handling events, e.g., e-mail delivery or human-computer interaction.

Showing that the calculus is useful can also be accomplished by demonstrating how it could be used as a formal foundation for practical programming languages with native event support. One such language, C#, is somewhat similar in syntax and semantics to Eventua. Thus, it appears that the semantics of C# is, to some extent, modeled using the $\varrho\varpi\varsigma$-calculus as a foundation through Eventua.

Chapter 7

RELATED WORK

There are three common themes in the related work on event mechanisms: works defining and implementing a general event mechanism, generalizing event mechanisms into frameworks, and providing formal reasoning about event mechanisms. This chapter provides an overview of the existing works in contrast to this work.

7.1 Definitions and Implementations of Event Mechanisms

The research defining and implementing event mechanisms attempts to solve the problem that no general purpose event mechanisms exist in most popular programming environments. Most of the research in this area is focused on implementing an event mechanism using an existing language. The focus of such works is simply to provide an event mechanism. This does not solve the problem, however, of extending the syntax and semantics of the language to include native support for events, the foundational premise of this work. Specifically, Garlan and Scott [GS93] build an Ada-like specification of events and describes how that specification is translated into Ada and how it is used. Notkin, et. al., [NGGS93] implements event mechanisms using three different languages, Ada, C++, and CLOS. However, in exploring ways to implement events, these two works developed a taxonomy of event mechanism design tradeoffs, which can be used for any problem related to support for events.

The Cambridge Event Architecture [BMB+00] is an event system based on descriptions of events using ODMG's Object Definition Language. Also, the ECO (Events, Constraints, and Objects) model [HMN+00] builds an event mechanism on top of C++ to include events. The focus of these two works is to solve problems related to interoperability [EGD01], and not exploring the benefits and theory of including events as language constructs.

The work that is closest to the goals of this work is Eugster, et. al., [EGD01]. Eugster, et. at., extend Java and explore the benefits and tradeoffs of adding events natively to a language, similar to this research. However, Eugster, et. al., merely provide an informal design taxonomy for event mechanisms, similar in scope to Garlan and Scott and Notkin, et. al., instead of attempting to provide a language formalization of an event mechanism as done here.

Fiege, et. al., [FMMB02] provide a formalism for structuring event availability and notification. The work by Fiege, et. al., is useful for thinking about how to manage and modularize events. However, the work does not investigate how to extend a language to support such concepts. As such, they are not attempting to solve the problem of native support for events in a language as we have here.

7.2 General Event Frameworks

The work that defines general event frameworks provide a basis to compare and ascertain various properties of event mechanisms. One approach taken by Garlan and Notkin [GN91], uses the Z specification language to define an event framework. They then use the specification to prove various properties of these mechanisms and compare existing mechanisms with each other. Berret, et. al., [BCTW96] creates an abstract, object-oriented framework to model event based integration. They define objects for each role of event based integration: routers (an event announcement manager), registrars (for registry and publication management), informers (event announcers), and participants (event handlers). This framework is then used, in a similar manner as [GN91], to compare different event mechanisms and ascertain various properties of these mechanisms. Frameworks such as these are useful tools for comparing existing and new event mechanisms with each other, and showing how general a particular mechanism or model is in relation to related mechanisms. For this research, Berret, et. al., [BCTW96]

is used to evaluate ability of the $\varrho\varpi\varsigma$-calculus to model common event mechanisms. The framework approach differs from this work because frameworks attempt to explain how event mechanisms work, whereas, we have shown how to create a specific event mechanism by building up language constructs.

7.3 Formal Reasoning About Events

Reasoning formally about the results of event announcement, as well as other event properties, has been researched with one approach: defining the semantics of event mechanism using approaches developed to solve other problems. Dingel, et. al. [DGJN98a] create a hybrid language using UNITY [CM88] as the basis for the semantics and a CCS-like syntax [Mil80] for communication to model events. They then use this language and ideas such as trace semantics and linear temporal logic to define a framework for reasoning about their event mechanism. While the framework proves to be useful for reasoning about their event mechanism, the event mechanism lacks some properties that would make the language more general, such as, dynamic publication of events and dynamic registration of handlers and other properties found in the $\varrho\varpi\varsigma$-calculus. That is, these works were focused on developing languages that helped reasoning about event mechanisms. The focus of the development of the $\varrho\varpi\varsigma$-calculus, on the other hand, was to create a calculus that would describe a generic notion of events in terms of programming language constructs. Dingel, et. al., also define a syntax and semantics that is similar those that model concurrent systems [DGJN98b]. Then, they use rely/guarantee reasoning [Jon83, Stø91], originally developed for concurrent systems, to help reason about their event mechanism. This approach is also useful for reasoning about some event mechanisms but again leaves out useful features such as explicit publication of events and registration of handlers, solving a different problem from this work.

Specifically, our research differs from [DGJN98a, DGJN98b] by providing syntax for dynamic binding of handlers to events (publication, unpublication, registration, and unregistration). Further, we provide structural operational semantics and a sound type system for the $\varrho\varpi\varsigma$-calculus which is absent from these other formalisms. On the other hand, a formal treatment for reasoning about event mechanisms is not covered in this research, since the focus is on providing a general, formal semantics model for event mechanisms rather than formal reasoning about events.

Chapter 8

FUTURE WORK AND CONCLUSIONS

The $\varrho\varpi\varsigma$-calculus has been shown to be a viable, formal foundation for programming languages that include built-in support for events. We have shown that the calculus has a sound type system. Further, the calculus is a conservative extension of Abadi and Cardelli's [AC96] **imp**ς-calculus, a well-known formal foundation for object-oriented languages, demonstrating the palatability of adding events to an existing object-oriented language. Use of the $\varrho\varpi\varsigma$-calculus to define a core event based language, Eventua, shows the viability of the calculus as a theoretical foundation for event based languages. The viability of the calculus as a theoretical foundation has also been demonstrated via successful comparison to the Barrett, et. al., EBI framework, [BCTW96].

For future work, investigation should be done to attempt to formally describe a significant subset of C# using Eventua since there are some similarities between the two. Another area of future work is to handle some of the issues raised in the discussion, Chapter 3, such as formally specifying interfaces that include events. Event interfaces, i.e., event scoping [FMMB02] should be explored in the context of how well the $\varrho\varpi\varsigma$-calculus supports the scoping notions that Fiege, et. al., present. Attempting to formally distinguish between internal (private) and external (public) events would be another worthwhile investigation.

The results of the analysis, Chapter 6, also raised some issues worth exploring further with respect to the EBI framework. Specifically, the calculus should be extended to support concurrency as well as some form of timing to round out support for every part of the EBI. Also, Eventua constructs for advanced features of event routing, such as event content and event ordering filtering, should be investigated. Research into adding composition of participants and events into groups, as defined in the EBI, or scopes, as defined by Fiege, et. al. [FMMB02], to Eventua would also be a good way to provide understanding for how (operationally) one could structure events and participants to reduce complexity and increase the ability to reason about a given system.

Another direction to take the $\varrho\varpi\varsigma$-calculus and Eventua would be to turn the formal specifications into a real implementation. The first step in this process would be to show that the translation of Eventua to the $\varrho\varpi\varsigma$-calculus is sound by showing that the type information is preserved in the translation. Then, take Eventua one step further by using it to describe a practical language with desirable features such as numbers and at least semi-recursive typing, e.g., a linked-list node contains a linked-list node reference. Finally, show how Eventua relates to a language that had events, such as C#, or define a new language, demonstrating its practicality by examples of solutions of real world problems.

Formal reasoning for the $\varrho\varpi\varsigma$-calculus should also be investigated. One approach would be to provide a translation to one of the languages specified by Dingel, et. al. [DGJN98a, DGJN98b] that can be reasoned about formally. Alternatively, a new framework for formal reasoning could be developed based on the approaches of Dingel, et. al. A different approach would be to use proof-carrying code techniques [Nec97] to allow event announcers to ensure that their handlers are well-behaved given a certain criteria. It may also be beneficial to study informal reasoning about programs that use events. Yet another aspect would be investigation into techniques that prevent some invalid and/or malicious event handlers from handling events during execution. It may be possible to accomplish that using techniques from proof-carrying code along with ideas from the field of computer security, such as those employed by Kerberos, PGP, etc., could be also be used to informally reason about events or programmatically prevent invalid and/or malicious handlers from handling events.

Finally, one of the motivating aspects of this paper, personally, was to make an attempt at a language construct that would be suitable for asynchronous notification outside of the typical setting for a language, e.g., across a network as in the mail delivery announcement example, page 5.3.1. A related topic would be the exploration of encapsulating more of the hardware than the typical language. For example, we talk about processors and languages being Turing complete, i.e., roughly, being able to do anything any other computer could

possibly compute. That is, we understand languages primarily as an abstraction mechanism of the algorithmic features of a microprocessor. However, nearly every processor today has asynchronous notification capabilities as well, e.g., traps, interrupts, etc. Abstracting that functionality into a language feature seems as logical as abstracting the algorithmic capabilities of a processor. To make it clear, the reason this notion is related to distributed computing is due to the fact that one could also view networking as a part of the computer, albeit it is the "greater" computer, not the microprocessor itself. And thus, language constructs that abstract elements of networking, such as use of the network for distributed computing, may be desirable.

Event mechanisms will continue to increase in prominence as the use of component models rise and as the emphasis of reuse and integration grows stronger in industry. Many current implementations of event mechanisms are inconvenient and often difficult to use, which seems to have more to do with the style of implementation rather than poor design. Some of the difficulties and inconveniences that current implementations have can be overcome by building support for an event mechanism into a programming language. The $\varrho\varpi\varsigma$-calculus and Eventua provide a formal basis that could be used as a tool for specifying and guiding the development of the syntax and semantics of built-in events for existing and future programming languages.

Appendix A

$\varrho\varpi\varsigma$-CALCULUS SUGARS AND DEFINITIONS

A.1 Introduction

The sugars and definitions listed in this appendix are some that are used throughout the paper. The purpose of listing them here is to give a more formal presentation of the implied semantics of their use in examples.

Some of the sugars and definitions that are listed here are not used in any of the examples. These are given here as examples of constructs that could be used for convenience in the future.

A.2 Serialization Sugars

```
a ; b         = let fresh = a in b, where fresh is not in FV(b)
begin a end b = let fresh = a in b, where fresh is not in FV(b)
begin a end   = begin a end [ ]
```

A.3 Booleans and If-then-else Sugar [AC96, p. 134]

```
true  = [if = s(x) x.then, then = s(x)x.then, else = s(x)x.else]
false = [if = s(x) x.else, then = s(x)x.then, else = s(x)x.else]

if b then c else d = ((b.then <= c).else <= d).if
a and b            = if a then b else false
a or b             = if a then true else b
not a              = if a then false else true
```

A.4 Boolean Type with T Result Type

```
Bool(T) = [if : T, then : T, else : T]
```

A.5 While-do Control Structure Sugar

```
while b do c =  [
                  do    = s(x) c; x.while,
                  while = s(x) if b then x.do else []
                ].while
```

A.6 Finite Natural Numbers from 0 to N

```
0 = [is0 = true, is1 = false, ..., isN = false, compare = s(x)compare1.is0, compare1 s(x) = x.compare1]
1 = [is0 = false, is1 = true, ..., isN = false, compare = s(x)compare1.is1, compare1 s(x) = x.compare1]
.
.
.
N = [is0 = false, is1 = false, ..., isN = true, compare = s(x)compare1.isN, compare1 s(x) = x.compare1]

Nat(T) = [is0 : Bool(T), is1 : Bool(T), ..., isN : Bool(T), compare : Bool(T), compare1 : NatPreds(T)]
NatPreds(T) = [is0 : Bool(T), is1 : Bool(T), ..., isN : Bool(T)]
```

A.7 Increment and Decrement Object for Finite Natural Numbers

```
math = [
         inc = s(x)
           if x.inc1.is0 then 1
           else if x.inc1.is1 then 2
           else ...
           else if x.inc1.isN then x.inc,
         inc1 = s(x) x.inc1,
         dec = s(x)
           if x.dec1.is0 then x.inc
           else if x.dec1.is1 then 0
           else ...
           else if x.dec1.isN then N-1,
         dec1 = s(x) x.dec1
       ]

Math = [inc : Nat(T), inc1 : Nat(T), dec : Nat(T), dec1 : Nat(T)]
```

A.8 Addition and Subtraction Sugars for Natural Numbers

```
 x + y = [
           plus = s(z) if z.left.is0 then z.result
                       else begin
                         z.result <= s(zz)(math.inc1 <= z.result).inc;
                         z.left   <= s(zz)(math.dec1 <= z.left).dec;
                         z.plus
                       end,
           result = s(z)x,
           left = s(z)y
         ].plus

 x - y = [
           minus = s(z) if z.left.is0 then z.result
                        else begin
                          z.result <= s(zz)(math.inc1 <= z.result).dec;
                          z.left   <= s(zz)(math.dec1 <= z.left).dec;
                          z.minus
                        end,
           result = s(z) x,
           left = s(z) y
         ].minus
```

A.9 Comparison Sugars for Natural Numbers

```
a == b = (a.compare1 <= b).compare
a > b  = [
           check = s(z)
             if z.left == 0 then false
             else begin
               z.left <= s(zz) z.left - 1
               if z.left == z.right then true
               else z.check
             end,
           left = s(z) a,
           right = s(z) b
         ].check
a < b  = b > a
```

A.10 For-do Control Structure Sugar

```
for x = n to m do a =
    let forObject = [ val   = s(y) n ] in
        [
          do  = s(y)
```

```
            let x = forObject.val in a;
            forObject.val <= s(z) n + 1;
            y.for,
          for = s(y)
            if n == m or n > m then []
            else y.do
        ].for
```

A.11 Size N Array of T Type Implementation and Type

```
Array = [
          lookup = s(x)
            if x.lookup1 == 1 then x.e1
            else if x.lookup1 == 2 then x.e2
            ...
            else if x.lookup1 == N then x.eN
            else x.error,
          lookup1 = s(x) x.lookup1
      store = s(x)
        if x.store1 == 1 then let newval = x.store2 in x.e1 <= s(y) newval
        else if x.store1 == 2 then let newval = x.store2 in x.e2 <= s(y) newval
        ...
        else if x.store1 == N then let newval = x.store2 in x.eN <= s(y) newval
        else x.error;
          store1 = s(x) x.store1,
          store2 = s(x) x.store2,
          e1 = s(x) x.e1,
          e2 = s(x) x.e2,
          ...
          eN = s(x) x.eN,
          error = s(x) x.error
        ]

Array(T) = [
            lookup : T, lookup1 : Nat(T), store : Top, store1 : Nat(T), store2: T,
            e1 : T, e2 : T, ..., eN : T, error : T
           ]
```

A.12 Array Access and Update for the Ith Element

```
arr[I]      = (arr.lookup1 <= I).lookup
arr[I] := a = ((arr.store1 <= I).store2 <= a).store
```

Appendix B

STANDARD PREAMBLE FOR EVENTUA EXAMPLES

B.1 Introduction

It is assumed that this preamble precedes every example of an Eventua program in the paper. The type placeholders, `T`, are assumed for each given context in which they occur. In order to get rid of the type placeholders, the preamble would need to be changed for every example to reflect the various types used in those examples.

While it would be more technically correct to include definitions for every type that the placeholder, `T`, represents, the presentation given in this appendix is both formal enough to convey the necessary information and terse enough to communicate the general idea of how the sugars work within the context of this paper.

B.2 Eventua Types Used and Defined in the Preamble

```
Bool(T)      = Object(if : T, then : T, else : T)
Nat(T)       = Object(
                       is0 : Bool(T), is1 : Bool(T), ..., isN : Bool(T),
                       compare : Bool(T)
                      )
NatPreds(T)  = Object(is0 : Bool(T), is1 : Bool(T), ..., isN : Bool(T))
Array(T)     = Object(e0 : T, e1 : T, ..., eN : T, lookup : T, store : Top)
```

B.3 Boolean True and False Values

```
#Actual preamble
class True(T) begin
    if = method() : T
        self.then()
    then = method() : T
        self.then()
    else = method() : T
self.else()
end

class False(T) begin
    if = method() : T
        self.else()
    then = method() : T
        self.then()
    else = method() : T
        self.else()
end
```

B.4 Finite Natural Numbers 0 to N

```
class 0(T) begin
    is0 = method() : Bool(T)
        new True(T)
    is1 = method() : Bool(T)
        new False(T)
```

```
    ...
    isN = method() : Bool(T)
        new False(T)
    compare( other : NatPreds(T) ) : Bool(T)
other.is0()
end

class 1(T) begin
    is0 = method() : Bool(T)
        new False(T)
    is1 = method() : Bool(T)
        new True(T)
    ...
    isN = method() : Bool(T)
        new False(T)
    compare( other : NatPreds(T) ) : Bool(T)
other.is1()
end

.
.
.

class N(T) begin
    is0 = method() : Bool(T)
        new False(T)
    is1 = method() : Bool(T)
        new False(T)
    ...
    isN = method() : Bool(T)
        new True(T)
    compare( other : NatPreds(T) ) : Bool(T)
other.isN()
end

class NatMath(T) begin
    inc = method( op : Nat(T) ) : Nat(T)
        if op.is0 then new 1
        else if op.is1 then new 2
        ...
        else if op.isN then self.inc(op)
    dec = method( op : Nat(T) ) : Nat(T)
        if op.is0 then self.dec(op)
        else if op.is1 then new 0
        ...
        else if op.isN then new (N-1)
    plus = method( result : Nat(T), left : Nat(T) ) : Nat(T)
if left.is0 then
            result
        else
            self.plus(self.inc(result), self.dec(left))
    minus = method( result : Nat(T), left : Nat(T) ) : Nat(T)
if left.is0 then
            result
        else
            self.minus(self.dec(result), self.dec(left))
end
```

B.5 Array Implementation in Eventua

```
class Array(T) begin
    e0 = private : T
    e1 = private : T
    ...
    eN = private : T
    lookup = method( index : Nat(T) ) : T
        if index.is0 then private.e0
```

```
        else if index.is1 then private.e1
        ...
        else if index.isN then private.eN
        else self.lookup(index)
    store = method( index : Nat(T), value : T ) : Top
        if index.is0 then private.e0 := value
        else if index.is1 then private.e1 := value
        ...
        else if index.isN then private.eN := value
        else self.store(index, value) ;
new Empty
end
```

B.6 Empty Object Class Definition

```
class Empty begin
end
```

B.7 While Control Structure Implementation

```
class While begin
    while = method() : Top
        if self.bool then
            self.body(); self.while
        else
            new Empty
    body = method() : Top
        self.body()
    bool = method() : Bool(T)
self.bool()
end
```

Appendix C

SYNTACTIC SUGARS FOR EVENTUA

C.1 Introduction

`false`, `true` and the numbers, `0` thru `N` are defined primarily for the sake of convenience. It is assumed that the placeholder for the type `T` will be clear from the context in which these sugars are used.

C.2 Eventua Syntactic Sugars

```
0                   = new 0(T)
1                   = new 1(T)
...
N                   = new N(T)

x + y               = ( new NatMath(T) ).plus(x, y)
x - y               = ( new NatMath(T) ).minus(x, y)
x == y
x[i]                = x.lookup(i)
x[i] := v           = x.store(i, v)

if b then c else d = b.then := method() c; b.else := method() d; b.if()
not b               = if b then false else true
a and b             = if a then b else false
a or b              = if a then true else b

while b do c        = let wh = new While in
                            wh.body := method() c;
                            wh.bool := method() b;
                            wh.while

false               = new False(T)
true                = new True(T)
```

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, corrected second printing, 1998 edition, 1996.

[BCTW96] D.J. Barrett, L.A. Clarke, P.L. Tarr, and A.E. Wise. A framework for event-based software integration. *ACM Trans. Software Engineering and Methodology*, 5(4):378–421, Oct 1996.

[BMB+00] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri. Generic support for distributed applications. *Computer*, 33(3):68–76, Mar 2000.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison Wesley, Reading, Mass., 1988.

[DGJN98a] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning about implicit invocation. In *Proceedings of the ACM SIGSOFT Sixth Symposum on Foundations of Software Engineering*, pages 209–221. ACM SIGSOFT, Nov. 1998.

[DGJN98b] J. Dingel, D. Garlan, S. Jha, and D Notkin. Towards a formal treatment of implicit invocation using rely/guarantee reasoning. *Formal Aspects of Computing*, 10:193–213, 1998.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proc. 1990 ACM SIGMOD*. ACM SIGMOD, May 1990.

[EGD01] P.Th. Eugster, R. Guerraoui, and C.H. Damm. On objects and events. In *OOPSLA '01 Proceedings*, pages 254–269, New York, Oct 2001. ACM SIGPLAN, ACM.

[FMMB02] L. Fiege, M. Mezini, G. Mühl, and A.P. Buchmann. Engineering event-based systems with scopes. In *16th European Conference on Object-Oriented Programming*, June 2002.

[Ger89] C. Gerety. HP SoftBench: A new generation of software development. Technical Report SESD-89-25, Hewlett-Packard Software Engineering Systems Division, Ft. Collins, Col., Nov 1989.

[GH78] J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, Mass., 1995.

[GN91] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In S. Prhen and W.J. Toetenel, editors, *VDM '91 Formal Software Development Methods, LNCS 551*, volume 1, pages 31–44, Noordwijkerhout, The Netherlands, Oct. 1991. Springer-Verlag.

[GS93] D. Garlan and C. Scott. Adding implicit invocation to traditional programming languages. In *Proc. 15th International Conference on Software Engineering*, pages 447–455. IEEE, 1993.

[GW98] D. Goldin and P. Wegner. Persistence as a form of interaction. Technical Report CS-98-07, Brown University, 1998.

[HGN91] A.N. Habermann, D. Garlan, and D. Notkin. Generation of integrated task-specific software environments. In R.F. Rashid, editor, *CMU Computer Science: A 25th Commemorative*. ACM Press, Reading, Mass., 1991.

[HMN+00] M. Haahr, R. Meier, P. Nixon, V. Cahill, and E. Jul. Filtering and scalability in the eco distributed event model. In *5th International Symposium on Software Engineering for Parallel and Distributed Systems*, 2000.

[Jon83] C.B. Jones. Tentative steps toward a development method for interfering programs. *Trans. on Programming Languages and Systems*, 5(4):569–619, Oct 1983.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. European Conference On Object-Oriented Programming, LNCS 1241*, Finland, June 1997. Springer-Verlag.

[Mil80] R. Milner. *A calculus of communicating systems*. Number 92 in Lecture Notes in Computer Science. Springer-Verlag, New York, 1980.

[Nec97] G.C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symp. Principles of programming languages*, pages 106–119, Paris, France, Jan. 1997. ACM.

[NGGS93] D. Notkin, D. Garlan, W.G. Griswold, and K. Sullivan. Adding implicit invocation to languages: Three approaches. In S. Nishio and A. Yonezawa, editors, *Object Technlogies for Advanced Software, LNCS 742*, pages 489–510, Kanazawa, Japan, Nov. 1993. Japan Society for Software Science and Technology and Japan Advanced Institute of Science and Technology, Springer-Verlag.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, September 1981.

[Rei90] S.P. Reiss. Connecting tools using message passing in the field environment. *IEEE Software*, 7(4):57–67, July 1990.

[Sch94] D.A. Schmidt. *The Structure of Typed Programming Languages*. MIT Press, Cambridge, Mass., 1994.

[SN92] K.J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Trans. Software Engineering and Methodology*, 1(3):229–268, July 1992.

[Stø91] K. Stølen. A method for the development of totally correct shared-state parallel programs. In *CONCUR '91*, pages 510–525. Springer-Verlag, 1991.