A Formalization of the Common Type System and CLS Rules of ECMA Standard 335: "Common Language Infrastructure"

Gerry Butler
gabutler@csse.monash.edu.au

Sea Ling
sling@csse.monash.edu.au

Christine Mingins
cmingins@csse.monash.edu.au

Copyright © 2002 Gerry Butler, Sea Ling, Christine Mingins

Version 1.0

School of Computer Science and Software Engineering
Monash University
PO Box 197 Caulfield East Vic 3145
Australia
September 13, 2002

Abstract

ECMA Standard 335, “Common Language Infrastructure (CLI)”, was published in December 2001. According to Partition I, Section 1, it *defines the Common Language Infrastructure (CLI) in which applications written in multiple high level languages may be executed in different system environments without the need to rewrite the application to take into consideration the unique characteristics of those environments.*

The standard is specified in natural language. Given the limitations of natural language, it is difficult to write a specification free of ambiguity and potential contradiction. Therefore, a formal model has been constructed in Z Notation of a subset of the Standard: the Common Type System (CTS) and many of the Common Language Specification (CLS) Rules. The model provides a framework for reasoning about the CTS and the CLS Rules, and establishes a base on which further areas of the Standard can be formalized.

In the process of writing the formalisms, possible contradictions and ambiguities have been identified, which are discussed either in the body of the text or in the Appendices. In many cases the authors are confident that they have correctly interpreted the intention of the Standard; where they are not confident, the problems are described in Appendix E, *Unresolved Questions.*

Acknowledgements

This work was funded by a grant from Microsoft Corporation.

The authors wish to thank many people at Monash and elsewhere for their participation in numerous discussions: Sita Ramakrishnan, Damien Watkins, Andreas Amentas, Julianto, Adnan Bader and other people. Bohdan Durnota and Damien Watkins reviewed an early draft.

Contents

I Type System **v**

1 Introduction **1**

1.1 Goal . . . 1
1.2 Method . . . 1
1.3 Mapping to the Real World . . . 3
1.4 Language Syntax and the Undefined Constants . . . 4
1.5 Constraining the Real World . . . 5
1.6 Specifying Constraints . . . 5
1.7 Notation . . . 5
1.7.1 Given Set . . . 6
1.7.2 Axiomatic Description . . . 6
1.7.3 Abbreviation Definition . . . 6
1.7.4 Free Type . . . 6
1.7.5 Conventions . . . 7

2 Foundations **8**

2.1 Name . . . 8
2.1.1 System Names . . . 8
2.1.2 Special Names . . . 9
2.2 Conventional Class and Conventional Object . . . 10
2.2.1 Conventional Class . . . 10
2.2.2 Conventional Object . . . 11
2.3 Named Type . . . 12
2.4 Abstract Named Type . . . 13
2.5 Sealed Named Type . . . 14
2.6 Nests . . . 14
2.7 Global Named Type . . . 15
2.8 Assembly . . . 16
2.9 System Named Type . . . 18
2.10 Class, Value Type and Interface . . . 23
2.11 Abstract Class . . . 24
2.12 Object and Value . . . 24
2.13 Reference Type . . . 25

2.14 Builtin Types . . . 25
2.15 Inheritance . . . 27
2.16 Interface Implementation . . . 27

3 Members **29**
3.1 Method, Field, Event, and Property . . . 29
3.2 Method Semantics . . . 31
3.3 Member Attributes . . . 32
3.4 Member Subsets . . . 34

4 Type System **36**
4.1 Type . . . 36
4.2 Instance . . . 40
4.3 Instance Of Type . . . 42
4.4 Array . . . 44
4.4.1 Array Shape . . . 44
4.4.2 Array Type and Array Instance . . . 46
4.5 Boxing . . . 48

5 Signature **58**
5.1 Calling Convention . . . 58
5.2 Signature as a Sequence of Types . . . 58
5.3 Member Signatures . . . 59
5.4 Signature Components . . . 60

6 Signature Dependent Constraints **63**
6.1 Enum Fields . . . 63
6.2 Type Building . . . 64
6.3 Uniqueness of Member Names . . . 65
6.4 Member Hiding . . . 66
6.5 Implements Method . . . 68
6.6 Member Overriding . . . 69
6.7 Implementation of Abstract Methods . . . 71

II CLS Rules **72**

7 CLS Compliant and Non-Compliant Markings **73**
7.1 Explicit Marking . . . 73
7.2 Compliant Assemblies . . . 75
7.3 Acquired Marking . . . 76
7.4 Compliant and Non-Compliant Named Types . . . 78

8 Exposure **80**
8.1 Visibility 80
8.2 Accessibility 82
8.3 Type and Member Exposure 85

9 CLS Rules **87**

III Appendices **108**

A Index of Constants **109**

B Real-World Representation **118**
B.1 Conformance 118
B.2 Types in Assemblies 118
B.3 Types and Instances 118
B.4 Locations 122
B.5 CLS Compliance in the Real World 123
B.6 Accessibility of Properties 123

C Proof of Theorems **124**

D ECMA References **143**

E Unresolved Questions **153**
E.1 Importing Non-uniquely Named Types 153
E.2 Inheriting CLS Compliant and CLS Non-Compliant Attributes 154
E.3 Increasing Accessibility in Nested Named Types 154
E.4 Applicability of CLS Rules Only to Exposed Items 155
E.5 Hide by Signature 156
E.6 Is Rule 12 a CLS Rule? 156
E.7 Can an Unmanaged Pointer Point to Heap? 156
E.8 CLS Compliance of Accessors 157
E.9 Compliance, Visibility, and Accessibility of General Types 157
E.9.1 Compliance 157
E.9.2 Visibility 158
E.9.3 Accessibility 158

Part I

Type System

Chapter 1

Introduction

> *An astronomer, a physicist, and a mathematician were traveling in Scotland. They looked out the window of the train and saw a black sheep standing in a field. The astronomer said, "That's interesting. Scottish sheep are black." The physicist replied, "Well ... some Scottish sheep are black." The mathematician responded, "In Scotland there exists at least one field, containing at least one sheep, at least one side of which is black."*
>
> *Source unknown. Paraphrased from Simon Singh, "Fermat's Last Theorem", Fourth Estate, London, 1997.*

1.1 Goal

To formalize the Type System and the CLS rules of the Common Language Infrastructure (CLI) described in ECMA Standard 335; to establish a framework for reasoning about their consistency and completeness; and to establish a basis from which further statements can be deduced. This document is not a replacement for ECMA or for any part of ECMA; nor is it a formal verification of any part of ECMA. It is a restatement of a subset of ECMA and an instrument to assist in understanding ECMA. Since this document has not been subjected to formal verification, it may contain errors: for example, it may contain internal contradictions, either copied from ECMA or newly introduced. If it contradicts ECMA, then ECMA takes precedence.

1.2 Method

We use the term "ECMA" to mean "ECMA Standard 335". "ECMA n" means "Partition n of ECMA". "ECMA Library" refers to the definitions contained in the ECMA library.

We use Z Notation (Section 1.7) as a formal specification language. Since the terms *name*, *type*, and *function* occur in both Z and ECMA, we use *meta-name*, *meta-type*, and *meta-function* to refer to the Z terms.

The formal specification has been checked for syntax errors and meta-type errors by the Wizard type checker [2]. It has not been subjected to any other formal verification.

The following notions are taken to be sufficiently well understood to form a basis for discussion: the general notion of object-oriented class; concepts from parts of ECMA that we are not formalizing; the syntax of supported languages; the mathematical tool-kit of Z.

We model the Type System and the CLS rules by means of a collection of constants specified in Z. The value of each constant is either a set or an element of a set. The name of each constant is *introduced* as a Z given set, an axiomatic description, an abbreviation definition, or a free type. Each introduction explicitly or implicitly specifies the constant's meta-type. Each introduction or a subsequent predicate constrains the value of the constant; or specifies its value in terms of other constants; or neither. If an introduction or predicate specifies the value of the constant, the constant is *defined*; otherwise it is *undefined*. Each undefined constant has a value determined outside the scope of the model; it is *given*; it must satisfy the constraints, if any. A defined constant's value is specified in terms of one or more defined or undefined constants. Every definition resolves ultimately to undefined constants; otherwise the model would contain at least one circular definition. Recursive definitions are permitted. The given sets of Z are special cases of undefined constants. Many undefined constants can be defined in more primitive terms. Appendix A is an index of constants. Any term used in explanatory text, but not listed in Appendix A, for example, TypeDef, is not a constant and it is outside the scope of the model.

Summary: each constant is introduced; each constant is either defined or undefined; an undefined constant may be constrained; each undefined constant has a given value satisfying the constraints; each defined constant has a value specified in terms of other constants; each definition resolves ultimately to undefined constants.

Our model re-states a subset of ECMA. However, ECMA can be interpreted in more than one way. Wherever ECMA admits multiple interpretations, and we are confident that our interpretation is correct, we explain our reasons, either in the body of the text or in Appendix D. Wherever we are not confident of our interpretation, we pose a question in Appendix E.

No statement about our model can be assumed true unless it is explicitly stated in the model or can be deduced by applying rules of inference. Conversely, we can rely on the validity of any statement that can be deduced by applying rules of inference. However, since the model is only a subset of ECMA, many properties specified in ECMA are not deducible from the model.

We do not model the progression of time. For an example of the immutability of time, see Section 2.5. All assemblies, classes, objects, identities, and other sets are taken to exist instantaneously. It follows that we do not model the development process, since all assemblies and classes are immutable. The next Section describes how our model can be mapped to the real world, where applications are always changing.

1.3 Mapping to the Real World

When an axiomatic system is mapped into a structure in the real world (or in another branch of mathematics), the real world structure is normally called a *model* for the axiomatic system. Our terminology differs from the normal terminology of mathematics: we say that the axiomatic system is a model of the real world.

Our model can be mapped to a multiplicity of parallel "real worlds", each of which corresponds to a distinct universe of discourse. The results are valid in any world in which the constants can be mapped to entities satisfying the constraints. An implementation of a CLI, a collection of programming languages, tools, frameworks, applications, and other software artefacts form such a world. The constants are mapped to entities of the world, and the constraints designate conditions that the entities of the world are required to satisfy. We are allowed to constrain the real world, because it consists of software artefacts that we "intend" to build, and the model is a partial specification of the construction rules.

The universe of discourse consists of a collection of software artefacts that target ECMA. An assembly is the highest level entity modelled. Virtual execution systems, compilers, and other tools are not explicitly modelled, but some axioms impose constraints on them.

There is more than one way to map our model to the real world. The model does not prescribe any particular mapping. It discusses some issues and it identifies some ways a mapping could be constructed; there may be others. Appendix B.1 discusses the conformance of real-world implementations with the model.

Consider a development or maintenance process where a build process produces a new assembly each time it runs. In our model, an assembly has an immutable set of classes and an immutable name constructed from the simple name, version, culture, and signature. If the build process produces a series of assemblies with distinct names, for example, by modifying the version, then there are at least two ways to map the model to the real world: we can consider all assemblies in the series to lie in the same real world, and we can map distinct assemblies in the model to distinct assemblies in the real world; or we can consider that each assembly produced by the build process lies in a different, parallel world. If the build process produces a

series of assemblies with identical names, then only the second mapping is possible, since assembly names in the model are unique in the universe of discourse.

In the real world, a given class evolves through multiple versions. In our model, classes need not have unique names, except special classes such as the class whose name is "`System.Object`". There are at least two ways to map multiple class versions to the real world, although our choices here are constrained by the choice we make in mapping assemblies. We can consider that the set of classes in the model contains all versions of all classes. This is possible because class names need only be unique within an assembly. (This is an over-simplification, as it does not recognize nested classes and the importation of classes from other assemblies.) Alternatively, we can consider that each version lies in a different, parallel world. However, multiple versions of special classes, such as the class whose name is "`System.Object`", must lie in different worlds, since their names are unique in the universe of discourse.

Since our model requires unique assembly names, it does not seem to support assemblies that do not have a strong name, as there is a possibility of non-unique simple names. However, since we are not modelling the assembly naming mechanism, the lack of support for assemblies without a strong name is not a shortcoming of our model, but a question of mapping from the model to the real world. The problem could be resolved by deeming that real world assemblies have an implied strong name constructed from, if necessary, a unique "virtual" signature. Alternatively, assemblies with non-unique names could be considered to lie in different, parallel real worlds. Other solutions may be possible, especially if our model is later extended to include the assembly naming mechanism.

Some sets are infinite. For example, since our model is abstract, we can imagine an infinity of assemblies and an infinity of named types. Even if all other sets are finite, *Type* is infinite (Section 4.1).

1.4 Language Syntax and the Undefined Constants

This document does not model language syntax. Therefore, many terms that could be defined by reference to source code or language syntax are undefined. For example, Section 3.1 introduces *MEMBER* as a given set and *Method*, *Field*, *Event*, and *Property* as a partition of *MEMBER*. However, it does not provide any means to determine whether a particular member is a method, field, event, or property. That is, method, field, event, and property are undefined, but constrained: they are members, and they are distinct from each other, and certain relationships exist (for example, a method may be a setter for some property), but the partitioning of *MEMBER* is given.

In many cases, the undefined terms are well defined by language syntax.

For example, language keywords clearly identify members and clearly identify whether a member is a method, field, event, or property. See Appendix D (II 9.1.3) for a similar example based on the inheritance graph.

1.5 Constraining the Real World

We noted above that some axioms impose constraints on real-world software artefacts such as virtual execution systems and compilers. Consider, for example, the nesting graph of named types (Section 2.6). The nesting graph is undefined, since language syntax determines the elements of *DirectlyNestedIn*. It is also constrained: a named type may be directly nested in, at most, one other named type, and the nesting graph must be acyclic. This imposes a constraint on compilers: a compiler must not allow a named type to be directly nested in more than one other named type, and it must enforce an acyclic nesting graph. In this case, the constraint is normally enforced by the lexical structure of languages: a nested named type is lexically enclosed by the named type it is directly nested in, so it is impossible to write a named type directly nested in more than one named type, and it is impossible to construct a nesting graph containing a cycle. However, we can imagine a language that defines nesting by some mechanism other than lexical enclosure. Note that ECMA II 21.29, by using the word *typical*, also allows this possibility. Therefore, Section 2.6 does not make any assumptions about languages; it does not assume that it is impossible to write a named type directly nested in more than one named type; or to construct a nesting graph containing a cycle.

We try not to assume that, in the real world, particular relationships cannot possibly exist, or must necessarily exist. If the existence or non-existence of a relationship matters, then we explicitly state it.

1.6 Specifying Constraints

We adopt the principle that our model should specify the minimum constraints necessary to achieve its ends, since every constraint potentially restricts the freedom and flexibility of implementations. We specify constraints only where a constraint is required to maintain consistency with the letter or the spirit of ECMA. We do not specify a constraint unless there is a compelling reason to do so.

1.7 Notation

We use a subset of Z Notation as defined by [4]. We introduce each constant as a given set, an axiomatic description, an abbreviation definition, or a free

type. These constructs are described below; their descriptions have been specialized to match the way we use each construct in our model.

1.7.1 Given Set

Each *given set* introduces an undefined constant that, for the purposes of our model, has no internal structure. Given sets are pairwise disjoint. They are basic meta-types. They are denoted by brackets, for example $[\mathit{UNICODE}]$.

1.7.2 Axiomatic Description

Each axiomatic description has one or more *declarations* and zero or more *predicates*. Each declaration introduces one or more (defined or undefined) constants. A declaration may implicitly constrain an undefined constant it introduces; for example, declaring a constant to be a function implicitly constrains it. Each predicate either constrains an undefined constant, or specifies the value of a defined constant in terms of other constants.

Each undefined constant may be further constrained by one or more additional predicates following the axiomatic description and separate from it. The value of each defined constant may be specified by one predicate following the axiomatic description and separate from it, instead of a predicate contained in the axiomatic description (e.g., Z4.41 introduces *InstanceOf* and Z4.48 defines its value).

Axiomatic descriptions are denoted by a vertical line on the left hand side; the predicate, if present, is written below a horizontal line; for example:

Z: 1.1

$$\mathit{VisibilityGreater} : \mathit{Visibility} \leftrightarrow \mathit{Visibility}$$

$$\mathit{VisibilityGreater} = \{\mathit{PublicVisibility} \mapsto \mathit{AssemblyVisibility}\}$$

1.7.3 Abbreviation Definition

Each abbreviation definition introduces one constant and defines it in terms of other constants. It is denoted by the operator "==". For example:

Z: 1.2

$$\mathit{AbstractNamedType} == \mathit{NamedType} \setminus \mathit{NonAbstractNamedType}$$

1.7.4 Free Type

A free type has the following form:

Z: 1.3

$$T ::= c_1 \mid \ldots \mid c_m \mid d_1 \langle\!\langle E_1[T] \rangle\!\rangle \mid \ldots \mid d_n \langle\!\langle E_n[T] \rangle\!\rangle$$

The set T is introduced as an undefined constant. Each term c_i introduces a defined constant as an element of T. Each term $d_j \langle\!\langle E_j[T] \rangle\!\rangle$ introduces an undefined constant d_j as an injective function from the set identified by the expression $E_j[T]$ to T. Each expression $E_j[T]$ may optionally reference T; if it does, then the free type is recursive; a recursive free type must contain at least one term that does not reference T.

1.7.5 Conventions

The meta-names of Z given sets are upper case. All other meta-names are proper case: the first character is upper case and they may contain embedded upper case characters. Undefined constants are listed after their introduction.

Our model consists of formal statements interspersed with explanatory text. The formal statements are in Z Notation. Everything else is explanatory. The explanatory text is a partial re-statement and explanation of the formal text. Although the explanatory text is not complete, subject to the limitations of natural language and the skills of the authors, it should not be contradictory or ambiguous; if it is, then the formal text takes precedence.

Diagrams are explanatory. Diagrams do not necessarily show all the relationships that exist among the entities on the diagram. For example, Figure 4.1 shows an *ObjectInst* relationship between the subsets *Object* and *ObjectInstance*. There is also an *ObjectInst* relationship, not shown, between the subsets *Value* and *ObjectInstance*.

If *Sss* or *SSS* is the meta-name of a set, in explanatory text we use the lower case name sss, without italics, to denote an element of *Sss*. If *Sss* contains internal upper case characters, in the lower case form we replace them with lower case characters preceded by spaces. For example, an element of the set *NamedType* is a named type.

In explanatory text we use the terminology *element of set* rather than *member of set* to avoid overloading *member*. Although *element* is overloaded (e.g., *element of array*), it is less used than *member*. We refrain from adopting the term *meta-element*.

Chapter 2

Foundations

2.1 Name

Z: 2.1
$[\mathit{UNICODE}]$

Z: 2.2
$\mathit{Name} == \mathrm{seq}_1\ \mathit{UNICODE}$

Undefined: *UNICODE*

UNICODE is the set of unicode characters specified by [1]. A name is a non-empty sequence of unicode characters. Each assembly, named type, method, field, event, and property has a name (Sections 2.8, 2.3, 3.1, ECMA I 7.5, II 6). Except in CLS Rules 4, 5, 6, 28 and 33, our model does not constrain the characters in a name. ECMA may specify additional constraints. Appendix D (I 7.5) discusses an issue relating to type names.

2.1.1 System Names

Z: 2.3
SystemObjectName, *SystemValueTypeName*,
SystemEnumName, *SystemTypedReferenceName*,
SystemArrayName, *SystemDelegateName*,
SystemStringName, *SystemSByteName*,
SystemByteName, *SystemInt16Name*,
SystemUInt16Name, *SystemInt32Name*,
SystemUInt32Name, *SystemInt64Name*,
SystemUInt64Name, *SystemIntPtrName*,
SystemUIntPtrName, *SystemBooleanName*,
SystemSingleName, *SystemDoubleName*,
SystemCharName : *Name*

Z: 2.4
$SystemName == \{SystemObjectName, SystemValueTypeName,$
$\quad SystemEnumName, SystemTypedReferenceName,$
$\quad SystemArrayName, SystemDelegateName, SystemStringName,$
$\quad SystemSByteName, SystemByteName, SystemInt16Name,$
$\quad SystemUInt16Name, SystemInt32Name, SystemUInt32Name,$
$\quad SystemInt64Name, SystemUInt64Name, SystemIntPtrName,$
$\quad SystemUIntPtrName, SystemBooleanName, SystemSingleName,$
$\quad SystemDoubleName, SystemCharName\}$

Undefined: $SystemObjectName, SystemValueTypeName,$
$SystemEnumName, SystemTypedReferenceName,$
$SystemArrayName, SystemDelegateName, SystemStringName,$
$SystemSByteName, SystemByteName, SystemInt16Name,$
$SystemUInt16Name, SystemInt32Name, SystemUInt32Name,$
$SystemInt64Name, SystemUInt64Name, SystemIntPtrName,$
$SystemUIntPtrName, SystemBooleanName, SystemSingleName,$
$SystemDoubleName, SystemCharName$

The above names are undefined because we have not specified *UNICODE* in sufficient detail to enable a definition. Each of the above names is introduced as a non-empty finite sequence of Unicode characters, but we do not formally specify which characters. Informally, we specify *SystemObjectName* as the given sequence "`System.Object`". Likewise, *SystemValueTypeName* is the given sequence "`System.ValueType`". The remaining system names are constructed by following the same pattern.

2.1.2 Special Names

Z: 2.5
$GlobalName, InstanceConstructorName,$
$\quad TypeConstructorName, GetName, SetName,$
$\quad AddName, RemoveName, RaiseName : Name$

Z: 2.6
$SpecialName == \{GlobalName, InstanceConstructorName,$
$\quad TypeConstructorName, GetName, SetName, AddName,$
$\quad RemoveName, RaiseName\}$

Undefined: *GlobalName*, *InstanceConstructorName*, *TypeConstructorName*, *GetName*, *SetName*, *AddName*, *RemoveName*, *RaiseName*

Informally: *GlobalName* is the sequence "<Module>". *InstanceConstructorName* is the sequence ".ctor". *TypeConstructorName* is the sequence ".cctor". *GetName* is the sequence "get_". *SetName* is the sequence "set_". *AddName* is the sequence "add_". *RemoveName* is the sequence "remove_". *RaiseName* is the sequence "raise_".

2.2 Conventional Class and Conventional Object

We take the general notions of *object* and *class* to be well understood from object-oriented theory. We explicitly state some properties that we use in our model. Initially our terminology is similar to the terminology of object-oriented theory: the meta-name *conventional object* refers to an object and *conventional class* refers to a class. After we formalise some properties, we adopt a terminology closer to ECMA; for example, we introduce the meta-name *named type* as a synonym for conventional class.

2.2.1 Conventional Class

Z: 2.7
$[CONVENTIONAL_CLASS]$

Z: 2.8
$HasConventionalClassName : CONVENTIONAL_CLASS \rightarrow Name$

Z: 2.9
$DirectlyInherits : CONVENTIONAL_CLASS \leftrightarrow CONVENTIONAL_CLASS$

Z: 2.10
$Inherits == DirectlyInherits^{+}$

Z: 2.11
$\forall\, i : Inherits \bullet \neg\, (second(i)\ \underline{DirectlyInherits}\ first(i))$

Undefined: *CONVENTIONAL_CLASS*, *HasConventionalClassName*, *DirectlyInherits*

CONVENTIONAL_CLASS is the set of conventional classes, as understood in general object-oriented theory. Conventional classes generally originate as source code in various languages that target ECMA; language syntax is outside the scope of our model. Some conventional classes may not be represented in source code, for example, conventional classes associated with array types, which may be generated by the virtual execution system. See Appendix B.3 for a discussion of the real-world representation of conventional classes. Each conventional class has one name. Our model does not recognize the notion of *namespace* (ECMA II 6). See Appendix D (II 6) for further discussion. If source code uses a namespace name, then the namespace name, followed by ".", is part of the name of each conventional class contained in the namespace. *HasConventionalClassName* is undefined; language syntax determines its elements.

Some conventional classes directly inherit other conventional classes. *DirectlyInherits* is undefined; language syntax determines its elements; if necessary, languages have an implicit rule that some conventional classes, by default, directly inherit the conventional class whose name is *SystemObjectName*. See Appendix D (II 9.1.3) for further discussion. *Inherits* is the transitive closure of *DirectlyInherits*. For example, if a directly inherits b, and b directly inherits c, then a inherits c. The inheritance graph is acyclic: if a inherits b, then b does not directly inherit a.

If a inherits b, in explanatory text we say that a is *derived* and b is a *base*. We do not define *derived* and *base* formally, because we do not use them in formal text.

2.2.2 Conventional Object

Z: 2.12
[*CONVENTIONAL_OBJECT*]

Z: 2.13
[*IDENTITY*]

Z: 2.14
[*STATE*]

Z: 2.15
$ConventionalObjectIdentity : IDENTITY \rightarrowtail\!\!\!\!\twoheadrightarrow CONVENTIONAL_OBJECT$

Z: 2.16
$ExactInstanceOf : CONVENTIONAL_OBJECT \rightarrow CONVENTIONAL_CLASS$

Z: 2.17
$ConventionalObjectState : STATE \leftrightarrow CONVENTIONAL_OBJECT$

Z: 2.18
$ObjectReference == IDENTITY$

Undefined: $CONVENTIONAL_OBJECT$, $IDENTITY$, $STATE$, $ConventionalObjectIdentity$, $ExactInstanceOf$, $ConventionalObjectState$

CONVENTIONAL_OBJECT is the set of conventional objects of all conventional classes. *IDENTITY* is the set of identities of all conventional objects. *STATE* is the set of states that any conventional object could potentially be in. The bijection *ConventionalObjectIdentity* relates each identity with a distinct conventional object; its elements are given. Since we do not model the progression of time, it follows that we do not model the creation and destruction of conventional objects. All conventional objects exist instantaneously, their identities are in *IDENTITY*, and their identities are immutable. Each conventional object is the exact instance of one conventional class. *ExactInstanceOf* is undefined; its elements are given.

Each conventional object has a state. There is a relationship between an object's state and the fields defined by and inherited by the conventional class it is an exact instance of. We do not model this relationship. Although we do not model the progression of time, in the real world, state changes. The relation *ConventionalObjectState* models the association between each conventional object and the states it could potentially be in: each conventional object is associated with multiple states, and multiple conventional objects could be in the same state. We do not model transitions between states. See Appendix B.3 for a discussion of the real-world representation of conventional objects, identities, and states.

To reflect ECMA terminology as closely as possible, we introduce *ObjectReference* as a synonym of *IDENTITY*. See Appendix B.3 for a description of the real-world representation of identities and object references.

2.3 Named Type

Z: 2.19
$NamedType == CONVENTIONAL_CLASS$

Z: 2.20
$HasNamedTypeName == HasConventionalClassName$

We introduce *NamedType* as a synonym for *CONVENTIONAL_CLASS*. Hereafter we do not use the meta-name *CONVENTIONAL_CLASS* except when we refer to an early part of our model.

In explanatory text we say that a named type is a *scope*, and that it *scopes* the names of members defined by it and nested named types directly nested in it. We do not define *scope* formally, because we do not use it in formal text.

2.4 Abstract Named Type

Z: 2.21
$NonAbstractNamedType == \operatorname{ran} ExactInstanceOf$

Z: 2.22
$AbstractNamedType == NamedType \setminus NonAbstractNamedType$

We partition *NamedType* into *NonAbstractNamedType* and *Abstract-NamedType* (ECMA I 7.9.6.2). Non-abstract named types have at least one exact instance. Since we do not model the progression of time, and *ConventionalObject* contains all conventional objects that could potentially exist, this constitutes a definition of *NonAbstractNamedType*.

Our notion of *abstract* is equivalent to ECMA's, but its definition differs. ECMA has the notion of an *abstract attribute* and a *concrete attribute* attached to classes and value types (ECMA I 7.9.6.2). Our model defines *abstract* as a subset, not an attachment; further, *AbstractNamedType* and *NonAbstractNamedType* are a partition of *NamedType*, not attributes of classes and value types. After we introduce *Interface* and *ValueType* (Section 2.10), we constrain each interface to be an abstract named type and each value type to be a non-abstract named type. We use the term *non-abstract named type* in place of ECMA's term *concrete*, except *non-abstract named type* is defined on all named types, while ECMA I 7.9.6.2 defines *concrete* only on classes and value types. See Appendix B.3 for the real world representation. See Appendix D (I 7.9.6.2) for further discussion.

Section 3.4 constrains each abstract method to be defined by an abstract named type. It follows that each named type that defines at least one abstract method is an abstract named type. Neither our model nor ECMA constrains a named type with no abstract methods to be a non-abstract named type.

Our model and ECMA both constrain each interface not to be a sealed named type (Section 2.10, ECMA I 7.9.9, II 21.34 item 27); it follows from

the definition of *SealedNamedType* (Section 2.5) that our model constrains each interface to be inherited by some non-abstract named type. Neither our model nor ECMA constrains each abstract class to be inherited by some non-abstract class. (In the real world, there are abstract classes from which no non-abstract class inherits. In our model, since *NamedType* contains all classes in the universe of discourse, it may not be sensible to have an abstract class from which no non-abstract class inherits. Nevertheless, since ECMA does not have such a constraint, and there is no compelling reason to include it in our model, we omit it in accordance with our principle of minimizing the number of constraints (Section 1.6).)

2.5 Sealed Named Type

Z: 2.23
$SealedNamedType == NamedType \setminus (\mathrm{ran}\ DirectlyInherits)$

A sealed named type is a named type from which no named type directly inherits (ECMA I 7.9.9). ECMA has the notion of a *sealed attribute* attached to a class or value type to indicate that no other class or value type can directly inherit it. Our model views *sealed* as a subset, not an attachment; since we do not model the progression of time (Section 1.2), and *NamedType* (instantaneously) contains every named type in the universe of discourse, it is sufficient to define *SealedNamedType* as the subset of *NamedType* from which no named type directly inherits. This approach is consistent with ECMA, since the named types that have ECMA's *sealed attribute* form a subset from which no named type directly inherits. After we introduce *Interface* and *ValueType* (Section 2.10), we prohibit each interface from being a sealed named type and we require each value type to be a sealed named type. See (Appendix B.3) for the real world representation. See Appendix D (I 7.9.9) for further discussion.

2.6 Nests

Z: 2.24
$DirectlyNestedIn : NamedType \nrightarrow NamedType$

Z: 2.25
$NestedIn == DirectlyNestedIn^{+}$

Z: 2.26
$\forall\, n : NestedIn \bullet \neg\, (second(n)\ \underline{DirectlyNestedIn}\ first(n))$

Z: 2.27
$\mathit{NestedNamedType} == \mathrm{dom}\ \mathit{DirectlyNestedIn}$

Z: 2.28
$\mathit{TopLevelNamedType} == \mathit{NamedType} \setminus \mathit{NestedNamedType}$

Z: 2.29
$$\forall t_0, t_1, t_2 : \mathit{NamedType} \mid$$
$$(t_1 \neq t_2 \wedge t_1 \ \underline{\mathit{DirectlyNestedIn}}\ t_0 \wedge t_2 \ \underline{\mathit{DirectlyNestedIn}}\ t_0) \bullet$$
$$\mathit{HasNamedTypeName}(t_1) \neq \mathit{HasNamedTypeName}(t_2)$$

Undefined: *DirectlyNestedIn*

Some named types are directly nested in, at most, one other named type (ECMA I 7.5.3.4, II 21.29). The elements of *DirectlyNestedIn* are given; language syntax determines its elements. *NestedIn* is the transitive closure of *DirectlyNestedIn*. For example, if a is directly nested in b, and b is directly nested in c, then a is nested in c. The nesting graph is acyclic: if a is nested in b, then b is not directly nested in a. A top level named type is a named type that is not nested. Named types directly nested in the same named type have distinct names (ECMA I 7.5.3.4).

If a is directly nested in b, in explanatory text we say that b is a *scope*, and b *scopes* a's name. See also Section 2.3. We do not define *scope* formally, because we do not use it in formal text.

See Section 1.5 and Appendix D (I 7.5.2 and I 7.5.3.4) for further discussion.

2.7 Global Named Type

Z: 2.30
$$\mathit{GlobalNamedType} == \{t : \mathit{NamedType} \mid$$
$$t \ \underline{\mathit{HasNamedTypeName}}\ \mathit{GlobalName}\}$$

Z: 2.31
$\mathrm{disjoint}\langle \mathrm{dom}(\mathit{DirectlyInherits}), \mathit{GlobalNamedType}\rangle$

Z: 2.32
$\mathrm{disjoint}\langle \mathrm{ran}(\mathit{DirectlyInherits}), \mathit{GlobalNamedType}\rangle$

Z: 2.33
$\mathit{GlobalNamedType} \subset \mathit{AbstractNamedType}$

Z: 2.34
$\mathit{GlobalNamedType} \subset \mathit{TopLevelNamedType}$

Theorem 2.1
$GlobalNamedType \subset SealedNamedType$

A global named type is a named type that global members are defined by. Its name is "`<Module>`". Global named types do not directly inherit any named type, and no named type directly inherits them (ECMA II 9.8). Global named types are abstract named types (ECMA II 9.8). Global named types are sealed named types. Global named types are top level named types.

2.8 Assembly

Z: 2.35
$[ASSEMBLY]$

Z: 2.36
$HasAssemblyName : ASSEMBLY \rightarrowtail Name$

Z: 2.37
$ContainedBy : NamedType \rightarrow ASSEMBLY$

Z: 2.38
$ExportedFrom : TopLevelNamedType \nrightarrow ASSEMBLY$

Z: 2.39
$ImportedInto : TopLevelNamedType \leftrightarrow ASSEMBLY$

Z: 2.40
$$\forall t_1, t_2 : TopLevelNamedType \mid t_1 \neq t_2 \land ContainedBy(t_1) = ContainedBy(t_2) \bullet HasNamedTypeName(t_1) \neq HasNamedTypeName(t_2)$$

Z: 2.41
$$\forall e : ExportedFrom \bullet ContainedBy(first(e)) = second(e)$$

Z: 2.42

$$\forall\, i : \mathit{ImportedInto} \bullet \\ \quad \exists\, e : \mathit{ExportedFrom} \bullet \\ \quad\quad (\mathit{first}(i) = \mathit{first}(e) \land \mathit{second}(i) \neq \mathit{second}(e))$$

Z: 2.43

$$\forall\, t_1 : \mathit{NamedType};\ t_2 : \mathit{TopLevelNamedType} \mid \\ \quad t_1\ \underline{\mathit{NestedIn}}\ t_2 \bullet \\ \quad \mathit{ContainedBy}(t_1) = \mathit{ContainedBy}(t_2)$$

Z: 2.44

$$\forall\, a : \mathit{ASSEMBLY} \bullet \\ \quad \exists_1\, t : \mathit{NamedType} \bullet \\ \quad\quad (\mathit{ContainedBy}(t) = a \land \\ \quad\quad t \in \mathit{GlobalNamedType})$$

Z: 2.45

$$\forall\, a : \mathit{ASSEMBLY};\ t : \mathit{GlobalNamedType} \mid \\ \quad t\ \underline{\mathit{ContainedBy}}\ a \bullet \\ \quad t\ \underline{\mathit{ExportedFrom}}\ a$$

Undefined: *ASSEMBLY*, *HasAssemblyName*, *ContainedBy*, *ExportedFrom*, *ImportedInto*

We assume that the general notion of assembly is understood from descriptions elsewhere in ECMA. We refine the notion by modelling some of the properties of assembly. ECMA recognizes properties and components outside the scope of our model, for example, resources.

ASSEMBLY is the set of assemblies determined by an assembly configuration mechanism. *HasAssemblyName* is undefined; the assembly configuration determines its elements. Each assembly has a distinct name. Since we are not formalizing the assembly naming mechanism, we take an assembly name to be a sequence of unicode characters derived from the simple name, version, culture, and signature (ECMA II 6).

Each (top level or nested) named type is contained by one assembly. Top level named types have unique names within the assembly they are contained by. Each assembly exports a (possibly empty) subset of the top level named types contained by it (ECMA I 7.5.3.1). Each assembly imports a (possibly empty) subset of the top level named types exported from other assemblies. *ContainedBy*, *ExportedFrom*, and *ImportedInto* are undefined; language syntax and the assembly configuration determine their elements. Each nested named type is contained by the same assembly that contains the top level named type it is nested in. Each assembly contains one global

named type; this is the named type that global members are defined by; it is exported from every assembly (ECMA II 9.8).

See Section 2.2.1 for a discussion of namespace. See Appendix E.1 for further discussion. For real-world mapping details, see Appendix B.2.

In explanatory text we say that an assembly is a *scope*, and that it *scopes* the names of the top level named types it contains. In ECMA, an assembly also scopes the names of other components, for example resources but, in our model, named types are the only components of an assembly. We do not define *scope* formally, because we do not use it in formal text.

2.9 System Named Type

Z: 2.46

$C_J : NamedType$

$$HasNamedTypeName(C_J) = SystemObjectName$$
$$\exists_1\, t : NamedType \bullet t = C_J$$
$$\neg\, (\exists\, t_1, t_2 : NamedType \bullet t_1 = C_J \land t_1 \; \underline{DirectlyInherits} \; t_2)$$

Z: 2.47

$C_V : NamedType$

$$HasNamedTypeName(C_V) = SystemValueTypeName$$
$$\exists_1\, t : NamedType \bullet t = C_V$$
$$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_V \land t_2 = C_J \land t_1 \; \underline{DirectlyInherits} \; t_2$$
$$C_V \in AbstractNamedType$$

Z: 2.48

$C_E : NamedType$

$$HasNamedTypeName(C_E) = SystemEnumName$$
$$\exists_1\, t : NamedType \bullet t = C_E$$
$$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_E \land t_2 = C_V \land t_1 \; \underline{DirectlyInherits} \; t_2$$
$$C_E \in AbstractNamedType$$

Z: 2.49

$C_R : NamedType$

$HasNamedTypeName(C_R) = SystemTypedReferenceName$

$\exists_1\, t : NamedType \bullet t = C_R$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_R \land t_2 = C_V \land$
$\quad t_1 \ \underline{DirectlyInherits}\ t_2$

Z: 2.50

$C_A : NamedType$

$HasNamedTypeName(C_A) = SystemArrayName$

$\exists_1\, t : NamedType \bullet t = C_A$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_A \land t_2 = C_J \land$
$\quad t_1 \ \underline{DirectlyInherits}\ t_2$

$C_A \in AbstractNamedType$

Z: 2.51

$C_D : NamedType$

$HasNamedTypeName(C_D) = SystemDelegateName$

$\exists_1\, t : NamedType \bullet t = C_D$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_D \land t_2 = C_J \land$
$\quad t_1 \ \underline{DirectlyInherits}\ t_2$

$C_D \in AbstractNamedType$

Z: 2.52

$C_S : NamedType$

$HasNamedTypeName(C_S) = SystemStringName$

$\exists_1\, t : NamedType \bullet t = C_S$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_S \land t_2 = C_J \land$
$\quad t_1 \ \underline{DirectlyInherits}\ t_2$

Z: 2.53

$C_I8 : NamedType$

$HasNamedTypeName(C_I8) = SystemSByteName$

$\exists_1\, t : NamedType \bullet t = C_I8$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_I8 \land t_2 = C_V \land$
$\quad t_1 \ \underline{DirectlyInherits}\ t_2$

Z: 2.54

$C_U8 : NamedType$

$HasNamedTypeName(C_U8) = SystemByteName$

$\exists_1\, t : NamedType \bullet t = C_U8$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_U8 \land t_2 = C_V \land$
$\quad t_1 \underline{DirectlyInherits}\ t_2$

Z: 2.55

$C_I16 : NamedType$

$HasNamedTypeName(C_I16) = SystemInt16Name$

$\exists_1\, t : NamedType \bullet t = C_I16$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_I16 \land t_2 = C_V \land$
$\quad t_1 \underline{DirectlyInherits}\ t_2$

Z: 2.56

$C_U16 : NamedType$

$HasNamedTypeName(C_U16) = SystemUInt16Name$

$\exists_1\, t : NamedType \bullet t = C_U16$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_U16 \land t_2 = C_V \land$
$\quad t_1 \underline{DirectlyInherits}\ t_2$

Z: 2.57

$C_I32 : NamedType$

$HasNamedTypeName(C_I32) = SystemInt32Name$

$\exists_1\, t : NamedType \bullet t = C_I32$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_I32 \land t_2 = C_V \land$
$\quad t_1 \underline{DirectlyInherits}\ t_2$

Z: 2.58

$C_U32 : NamedType$

$HasNamedTypeName(C_U32) = SystemUInt32Name$

$\exists_1\, t : NamedType \bullet t = C_U32$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_U32 \land t_2 = C_V \land$
$\quad t_1 \underline{DirectlyInherits}\ t_2$

Z: 2.59

$C_I64 : NamedType$

$HasNamedTypeName(C_I64) = SystemInt64Name$

$\exists_1\, t : NamedType \bullet t = C_I64$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_I64 \land t_2 = C_V \land$
$\quad t_1 \; \underline{DirectlyInherits} \; t_2$

Z: 2.60

$C_U64 : NamedType$

$HasNamedTypeName(C_U64) = SystemUInt64Name$

$\exists_1\, t : NamedType \bullet t = C_U64$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_U64 \land t_2 = C_V \land$
$\quad t_1 \; \underline{DirectlyInherits} \; t_2$

Z: 2.61

$C_I : NamedType$

$HasNamedTypeName(C_I) = SystemIntPtrName$

$\exists_1\, t : NamedType \bullet t = C_I$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_I \land t_2 = C_V \land$
$\quad t_1 \; \underline{DirectlyInherits} \; t_2$

Z: 2.62

$C_U : NamedType$

$HasNamedTypeName(C_U) = SystemUIntPtrName$

$\exists_1\, t : NamedType \bullet t = C_U$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_U \land t_2 = C_V \land$
$\quad t_1 \; \underline{DirectlyInherits} \; t_2$

Z: 2.63

$C_B : NamedType$

$HasNamedTypeName(C_B) = SystemBooleanName$

$\exists_1\, t : NamedType \bullet t = C_B$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_B \land t_2 = C_V \land$
$\quad t_1 \; \underline{DirectlyInherits} \; t_2$

Z: 2.64

$C_F32 : NamedType$

$HasNamedTypeName(C_F32) = SystemSingleName$

$\exists_1\, t : NamedType \bullet t = C_F32$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_F32 \land t_2 = C_V \land$
$\quad t_1 \underline{DirectlyInherits}\ t_2$

Z: 2.65

$C_F64 : NamedType$

$HasNamedTypeName(C_F64) = SystemDoubleName$

$\exists_1\, t : NamedType \bullet t = C_F64$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_F64 \land t_2 = C_V \land$
$\quad t_1 \underline{DirectlyInherits}\ t_2$

Z: 2.66

$C_C : NamedType$

$HasNamedTypeName(C_C) = SystemCharName$

$\exists_1\, t : NamedType \bullet t = C_C$

$\exists\, t_1, t_2 : NamedType \bullet t_1 = C_C \land t_2 = C_V \land$
$\quad t_1 \underline{DirectlyInherits}\ t_2$

Z: 2.67

$SystemNamedType == \{C_J, C_V, C_E, C_R, C_A,$
$\quad C_D, C_S, C_I8, C_U8, C_I16, C_U16, C_I32,$
$\quad C_U32, C_I64, C_U64, C_I, C_U, C_B, C_F32,$
$\quad C_F64, C_C\}$

We introduce the named types forming the core of the type system, which we call system named types. These include the builtin types listed in ECMA I 7.2.2. The system named types (and many others) are fully defined in the ECMA Library. We model only a subset of the properties defined by the ECMA Library. For example, we specify that C_V has no instances, and that C_J directly inherits nothing. The names of the system named types are unique in the universe of discourse, guaranteeing that they can be imported into any assembly without causing a name clash. In Section 8.3 we constrain system named types to be exposed top level named types.

2.10 Class, Value Type and Interface

Z: 2.68

$$Class == \{t : NamedType \mid t = C_E \lor t \in GlobalNamedType \lor ((t \mapsto C_J \in Inherits^*) \land \neg (t \mapsto C_V \in Inherits))\}$$

Z: 2.69

$$ValueType == \{t : NamedType \mid t \neq C_E \land (t \mapsto C_V \in Inherits)\}$$

Z: 2.70

$$Interface == \{t : NamedType \mid t \notin GlobalNamedType \land \neg (t \mapsto C_J \in Inherits^*)\}$$

Z: 2.71

$$\neg (\exists t_0, t_1, t_2 : NamedType \bullet t_1 \neq t_2 \land (t_0 \mapsto t_1 \in DirectlyInherits) \land (t_0 \mapsto t_2 \in DirectlyInherits) \land (t_1 \in Class \cup ValueType) \land (t_2 \in Class \cup ValueType))$$

Z: 2.72

$$ValueType \subset SealedNamedType$$

Z: 2.73

$$ValueType \subset NonAbstractNamedType$$

Z: 2.74

$$Interface \subset AbstractNamedType$$

Z: 2.75

$$\mathrm{disjoint}\langle SealedNamedType, Interface\rangle$$

Z: 2.76

$$\mathrm{disjoint}\langle \mathrm{ran}((Class \cup ValueType) \lhd DirectlyNestedIn), Interface\rangle$$

Theorem 2.2

$$\langle Class, ValueType, Interface\rangle \ partitions \ NamedType$$

Based on the inheritance graph, we partition *NamedType* into *Class*, *ValueType*, and *Interface* (ECMA II 9.1.3). See Figure 2.1. A class is C_J, or C_E, or a global named type, or a named type that inherits C_J but does not inherit C_V. Note that C_J and C_V are classes, and that C_E is explicitly defined to be a class, although it inherits C_V. Appendix D (II 9.1.3) discusses the reasons for explicitly defining C_E to be a class. A value type is a named type, other than C_E, that inherits C_V. An interface is a named type, other than C_J or a global named type, that does not inherit C_J.

Classes and value types do not directly inherit multiple named types unless every base except one is an interface (ECMA I 7.9.9, I 7.9.11, I 7.9.6.4). Each value type is a sealed named type (ECMA I 7.9.10, II 21.34 item 35), and a non-abstract named type. Each interface is an abstract named type (ECMA II 21.34 item 23), and no interface is a sealed named type (ECMA I 7.9.9, II 21.34 item 27). Classes and value types are not nested in interfaces (ECMA II 9.6); no other nesting combinations are prohibited.

Our approach differs from ECMA II 9.1.3, which identifies an interface from the setting of the interface flag in a TypeDef row: we distinguish between class and value type on the basis of their inheritance (Appendix D (II 9.1.3)).

2.11 Abstract Class

Z: 2.77
$NonAbstractClass == NonAbstractNamedType \cap Class$

Z: 2.78
$AbstractClass == Class \setminus NonAbstractClass$

2.12 Object and Value

Z: 2.79
$Object == \mathrm{dom}(ExactInstanceOf \rhd Class)$

Z: 2.80
$Value == \mathrm{dom}(ExactInstanceOf \rhd ValueType)$

Theorem 2.3
$\langle Object, Value\rangle$ *partitions* $CONVENTIONAL_OBJECT$

Based on the *NamedType* partition and the *ExactInstanceOf* meta-function, we partition *CONVENTIONAL_OBJECT* into *Object* and *Value*. An object is an exact instance of some class. A value is an exact instance of some value type.

Hereafter, we do not use the meta-name *CONVENTIONAL_OBJECT* except when we refer to an early part of our model.

2.13 Reference Type

Z: 2.81
$ReferenceType == Class \cup Interface$

2.14 Builtin Types

Z: 2.82
$IntegerType == \{C_I8, C_U8, C_I16, C_U16, C_I32, C_U32,$
$\quad C_I64, C_U64, C_I, C_U\}$

Z: 2.83
$BooleanType == \{C_B\}$

Z: 2.84
$FloatingType == \{C_F32, C_F64\}$

Z: 2.85
$CharType == \{C_C\}$

Z: 2.86
$TypedrefType == \{C_R\}$

Z: 2.87
$AbstractValueType == \{C_E\}$

Z: 2.88
$BuiltinValueType == \bigcup\{IntegerType, BooleanType, FloatingType,$
$\quad CharType, TypedrefType\}$

Z: 2.89
$UserDefinedValueType ==$
$\quad ValueType \setminus BuiltinValueType \setminus AbstractValueType$

Z: 2.90
$EnumType == \mathrm{dom}(DirectlyInherits \rhd \{C_E\})$

Z: 2.91
$UserDefinedOrdinaryValueType ==$
$\quad UserDefinedValueType \setminus EnumType$

Z: 2.92
$BuiltinReferenceType == \{C_J, C_S\}$

Theorem 2.4
$IntegerType \subset ValueType$

Theorem 2.5
$BooleanType \subset ValueType$

Theorem 2.6
$FloatingType \subset ValueType$

Theorem 2.7
$CharType \subset ValueType$

Theorem 2.8
$TypedrefType \subset ValueType$

Theorem 2.9
$AbstractValueType \subset Class$

Theorem 2.10
$BuiltinValueType \subset ValueType$

Theorem 2.11
$EnumType \subset ValueType$

Theorem 2.12
$EnumType \subset UserDefinedValueType$

Theorem 2.13
$BuiltinReferenceType \subset Class$

Theorem 2.14
$disjoint\langle AbstractValueType, BuiltinValueType, UserDefinedValueType\rangle$

We introduce subsets of named types. For example, a floating type is C_F32 or C_F64. Each enum type directly inherits C_E (ECMA II 13.3, II 21.34). For convenience, we introduce new terminology not used by ECMA: *AbstractValueType*, *BuiltinValueType*, *UserDefinedValueType*, *UserDefinedOrdinaryValueType*, *BuiltinReferenceType*.

2.15 Inheritance

Theorem 2.15
$\neg \exists c : Class;\ v : ValueType \bullet c \mapsto v \in Inherits$

Theorem 2.16
$\neg \exists i : Interface;\ c : Class \bullet i \mapsto c \in Inherits$

Theorem 2.17
$\neg \exists i : Interface;\ v : ValueType \bullet i \mapsto v \in Inherits$

Theorem 2.18
$\neg \exists v : ValueType;\ c : Class \bullet$
$(c \neq C_J) \wedge c \neq C_V) \wedge (c \neq C_E) \wedge (v \mapsto c \in Inherits)$

Theorem 2.19
$\neg \exists v_1, v_2 : ValueType \bullet v_1 \mapsto v_2 \in Inherits$

Classes inherit only from other classes or interfaces. Interfaces inherit only from other interfaces. Value types inherit only from C_V, or C_E, or interfaces.

2.16 Interface Implementation

Z: 2.93
$RequiresImplementationOf == \{i : Inherits \mid$
$first(i) \in Interface \wedge second(i) \in Interface\}$

Z: 2.94
$Implements == \{i : Inherits \mid$
$first(i) \notin Interface \wedge second(i) \in Interface\}$

We define the meta-names *RequiresImplementationOf* and *Implements* as synonyms for *Inherits* when the base is an interface. For example, if c is a class and i is an interface and c inherits i, we say c implements i, even if c is an abstract named type, and there exists a member m such that m is defined by i and m is not defined by c. If i_1 and i_2 are interfaces and i_1 inherits i_2, we say i_1 requires implementation of i_2.

Our model does not formally prohibit use of the meta-name *Inherits* when the base is an interface: to conform with ECMA, it allows use of the meta-names *Implements* (ECMA I 7.9.4) and *RequiresImplementationOf* (ECMA I 7.9.4, II 21.34), and hereafter, by choice, we use them whenever they are allowed, except when it is simpler and clearer to use *Inherits*. ECMA, too, often uses *Inherits* when the base may be an interface (e.g., ECMA I 7.9.8, I 7.9.11). It is nearly always simpler and clearer to use *Inherits*, because *Inherits* includes cases where the base is a class or a value type.

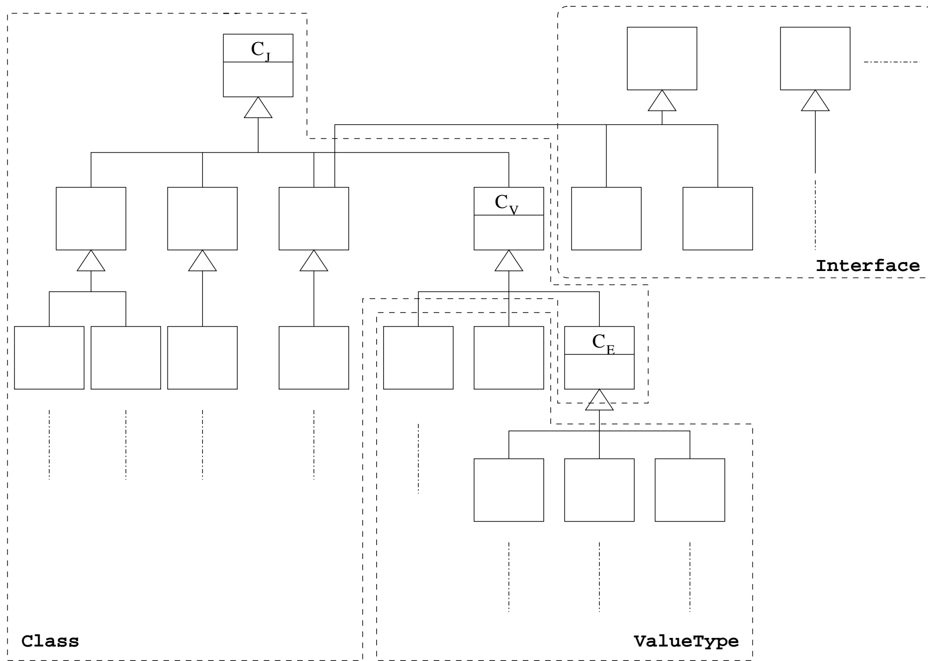

Figure 2.1: Class, ValueType and Interface

Chapter 3

Members

3.1 Method, Field, Event, and Property

Z: 3.1
$[MEMBER]$

Z: 3.2
$Method, Field, Event, Property : \mathbb{P}\ MEMBER$

Z: 3.3
$DefinedBy :$
$\quad MEMBER \rightarrow CONVENTIONAL_CLASS$

Z: 3.4
$HasMethodName : Method \rightarrow Name$

Z: 3.5
$HasFieldName : Field \rightarrow Name$

Z: 3.6
$HasEventName : Event \rightarrow Name$

Z: 3.7
$HasPropertyName : Property \rightarrow Name$

Z: 3.8
$\langle Method, Field, Event, Property \rangle$ partitions $MEMBER$

Z: 3.9
$HasMemberName ==$
$\bigcup\{HasMethodName, HasFieldName, HasEventName,$
$HasPropertyName\}$

Undefined: $MEMBER, Method, Field, Event, Property,$
$DefinedBy, HasMethodName, HasFieldName, HasEventName,$
$HasPropertyName$

Theorem 3.1
$dom\, HasMemberName = MEMBER$

Theorem 3.2
$\forall\, m_1, m_2 : MEMBER \mid m_1 = m_2 \bullet$
$HasMemberName(m_1) = HasMemberName(m_2)$

MEMBER is the set of members of all named types. *MEMBER* is given and *Method*, *Field*, *Event*, and *Property* are a given partition of *MEMBER*; language syntax defines a member and determines whether a particular member is a method, field, event, or property (ECMA I 7.4, I 7.9.6.3, I 7.11). Each member is defined by one conventional class. *DefinedBy* is undefined; language syntax determines its elements. Each method, field, event, and property has one name. Elements of *HasMethodName*, *HasFieldName*, *HasEventName*, and *HasPropertyName* are given; language syntax associates a method, field, event, or property with its name. See Section 6.3 for constraints on the uniqueness of member names. *HasMemberName* is a total function from *MEMBER* to *Name*.

In explanatory text we call method, field, event, property, or nested named type a *kind* (ECMA I 7.5.2). We do not define *kind* formally, because we do not use it in formal text. If we use *kind* in a context in which it cannot apply to a nested named type, we do not generally explain that *kind* does not apply to nested named type. If we use *kind* in a context in which it could apply to a nested named type, but we specifically intend that it does not apply in this context, then we explicitly state that it does not apply.

A member defined by a global named type is a *global member*. We do not define *global member* formally, because we do not use it in formal text.

Some names are declared in a method, for example, names of parameters. Neither our model nor ECMA uses the term *scope* when referring to a collection of names declared in a method.

3.2 Method Semantics

Z: 3.10
$SetterOf : Method \pinj Property$

Z: 3.11
$GetterOf : Method \pinj Property$

Z: 3.12
$PropertyOtherOf : Method \pfun Property$

Z: 3.13
$AddOnOf : Method \pinj Event$

Z: 3.14
$RemoveOnOf : Method \pinj Event$

Z: 3.15
$FireOf : Method \pinj Event$

Z: 3.16
$EventOtherOf : Method \pfun Event$

Z: 3.17

$$MethodSemantics == \bigcup\{SetterOf, GetterOf, PropertyOtherOf, AddOnOf, RemoveOnOf, FireOf, EventOtherOf\}$$

Z: 3.18

$$\forall s : MethodSemantics \bullet DefinedBy(first(s)) = DefinedBy(second(s))$$

Undefined: $SetterOf$, $GetterOf$, $PropertyOtherOf$, $AddOnOf$, $RemoveOnOf$, $FireOf$, $EventOtherOf$

Some methods are related to an event or a property. For example, a method may be a setter method for a property, or a fire method for an event. We call such relationships *method semantics* (ECMA II 21.13, II 21.26, II 21.31). If a method is a method semantics for an event or a property, then the method and the event or property are defined by the same named type (ECMA II 21.26). CLS rules impose additional restrictions on method semantics (Rules 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 37, 38), and ECMA imposes additional restrictions outside the scope of our model. The elements of *MethodSemantics* are given; language syntax determines its elements.

3.3 Member Attributes

Z: 3.19
$MemberAttribute ::= SpecialNameMemberAttribute$

Z: 3.20
$MethodAttribute ::= StaticMethodAttribute \mid$
$VirtualMethodAttribute \mid$
$NewslotMethodAttribute \mid$
$FinalMethodAttribute \mid$
$AbstractMethodAttribute \mid$
$HideBySignatureMethodAttribute$

Z: 3.21
$FieldAttribute ::= LiteralFieldAttribute \mid$
$StaticFieldAttribute$

Z: 3.22
$HasMemberAttribute : MEMBER \leftrightarrow MemberAttribute$

Z: 3.23
$HasMethodAttribute : Method \leftrightarrow MethodAttribute$

Z: 3.24
$HasFieldAttribute : Field \leftrightarrow FieldAttribute$

Z: 3.25
$\forall m : Method \mid$
$m \underline{HasMethodAttribute} StaticMethodAttribute \bullet$
$\neg (m \underline{HasMethodAttribute} VirtualMethodAttribute)$

Z: 3.26

$\forall\, m : Method \mid$
$\quad m\ \underline{HasMethodAttribute}\ FinalMethodAttribute \bullet$
$\quad \neg\,(m\ \underline{HasMethodAttribute}\ AbstractMethodAttribute)$

Z: 3.27

$\forall\, m : Method \mid$
$\quad m\ \underline{HasMethodAttribute}\ AbstractMethodAttribute \bullet$
$\quad m\ \underline{HasMethodAttribute}\ VirtualMethodAttribute$

Z: 3.28

$\forall\, m : Method \mid$
$\quad m\ \underline{HasMethodAttribute}\ FinalMethodAttribute \bullet$
$\quad m\ \underline{HasMethodAttribute}\ VirtualMethodAttribute$

Z: 3.29

$\forall\, m : Method \mid$
$\quad m\ \underline{HasMethodAttribute}\ NewslotMethodAttribute \bullet$
$\quad m\ \underline{HasMethodAttribute}\ VirtualMethodAttribute$

Z: 3.30

$\forall\, m : Method \mid$
$\quad m\ \underline{HasMethodAttribute}\ HideBySignatureMethodAttribute \bullet$
$\quad m\ \underline{HasMethodAttribute}\ VirtualMethodAttribute$

Z: 3.31

$\forall\, m : Field \mid$
$\quad m\ \underline{HasFieldAttribute}\ LiteralFieldAttribute \bullet$
$\quad m\ \underline{HasFieldAttribute}\ StaticFieldAttribute$

Undefined: $HasMemberAttribute$, $HasMethodAttribute$, $HasFieldAttribute$

Theorem 3.3

$\forall\, m : Method \mid$
$\quad m\ \underline{HasMethodAttribute}\ StaticMethodAttribute \bullet$
$\quad \neg\,(m\ \underline{HasMethodAttribute}\ FinalMethodAttribute)$

Theorem 3.4

$\forall\, m : Method \mid$
$\quad m\ \underline{HasMethodAttribute}\ StaticMethodAttribute \bullet$
$\quad \neg\,(m\ \underline{HasMethodAttribute}\ NewslotMethodAttribute)$

We introduce member attributes, for example, *StaticMethodAttribute*, *LiteralFieldAttribute*, and *StaticFieldAttribute* (ECMA II 22.1.4, II 22.1.5, II 22.1.9, II 22.1.13). Each member has zero or more attributes. Elements of *HasMemberAttribute*, etc. are given; language syntax determines their elements. ECMA defines other member attributes outside the scope of our model.

No method has both a static method attribute and a final method attribute; or a static method attribute and a virtual method attribute; or a static method attribute and a newslot method attribute; or a final method attribute and an abstract method attribute (ECMA II 21.24 item 6).

Each method that has an abstract method attribute, or a final method attribute, or a newslot method attribute, or a hide by signature method attribute also has a virtual method attribute (ECMA II 21.24 item 7, item 23).

Each field that has a literal field attribute, also has a static field attribute (ECMA II 21.15 item 7).

Although ECMA explicitly states each of the above constraints, some constraints are deducible from others. We state the dependent constraints as theorems.

See Appendix E.5 for a discussion of *HideBySignatureMethodAttribute* in relation to fields, events, and properties.

3.4 Member Subsets

Z: 3.32

$$\mathit{AbstractMethod} == \{m : \mathit{Method} \mid \\ \quad m \ \underline{\mathit{HasMethodAttribute}} \ \mathit{AbstractMethodAttribute}\}$$

Z: 3.33

$$\mathit{NonAbstractMethod} == \mathit{Method} \setminus \mathit{AbstractMethod}$$

Z: 3.34

$$\mathit{StaticMethod} == \{m : \mathit{Method} \mid \\ \quad m \ \underline{\mathit{HasMethodAttribute}} \ \mathit{StaticMethodAttribute}\}$$

Z: 3.35

$$\mathit{VirtualMethod} == \{m : \mathit{Method} \mid \\ \quad m \ \underline{\mathit{HasMethodAttribute}} \ \mathit{VirtualMethodAttribute}\}$$

Z: 3.36

$$\mathit{InstanceMethod} == \mathit{Method} \setminus \mathit{StaticMethod} \setminus \mathit{VirtualMethod}$$

Z: 3.37
$StaticField == \{m : Field \mid$
$\quad m \ \underline{HasFieldAttribute} \ StaticFieldAttribute\}$

Z: 3.38
$InstanceField == Field \setminus StaticField$

Z: 3.39
$\operatorname{ran}(AbstractMethod \lhd DefinedBy) \subseteq AbstractNamedType$

Theorem 3.5
$\langle StaticMethod, VirtualMethod, InstanceMethod \rangle \ partitions \ Method$

We define subsets of MEMBER based on the presence of certain combinations of attributes.

Each abstract method is defined by one abstract named type.

Chapter 4

Type System

We take the general notion of *type* to be sufficiently well understood to provide a basis for discussion. We specialize the notion by modelling an ECMA type system based on named types, object references, managed and unmanaged pointers, and function pointers. See Appendix B.3 for a discussion of the real-world representation of types.

4.1 Type

Z: 4.1
$[\mathit{Type}]$

Z: 4.2
$\mathit{Void} : \mathit{Type}$

Z: 4.3
$\mathit{VoidType} == \{\mathit{Void}\}$

Z: 4.4
$\mathit{Atomic} : \mathit{NamedType} \rightarrowtail \mathit{Type}$

Z: 4.5
$\mathit{AtomicNamedType} == \mathrm{ran}\ \mathit{Atomic}$

Z: 4.6
$\mathit{Ubvt} : \mathit{ValueType} \rightarrowtail \mathit{Type}$

Z: 4.7
$UnboxedValueType == \operatorname{ran} Ubvt$

Z: 4.8
$ObjRef : NonAbstractNamedType \rightarrowtail Type$

Z: 4.9
$ObjectReferenceType == \operatorname{ran} ObjRef$

Z: 4.10
$ManPtr :$
$\quad (UnboxedValueType \cup ObjectReferenceType) \rightarrowtail Type$

Z: 4.11
$ManagedPointerType == \operatorname{ran} ManPtr$

Z: 4.12
$UnmanPtr : Type \rightarrowtail Type$

Z: 4.13
$UnmanagedPointerType == \operatorname{ran} UnmanPtr$

Z: 4.14
$FnPtr : NonAbstractMethod \rightarrowtail Type$

Z: 4.15
$FunctionPointerType == \operatorname{ran} FnPtr$

Z: 4.16
$\langle VoidType, AtomicNamedType, UnboxedValueType,$
$\quad ObjectReferenceType, ManagedPointerType,$
$\quad UnmanagedPointerType, FunctionPointerType\rangle$
$\quad \text{partitions } Type$

Z: 4.17
$Elementary == \bigcup\{Atomic, Ubvt, ObjRef\}$

Z: 4.18
$ElementaryType == VoidType \cup \operatorname{ran} Elementary$

Z: 4.19
$PointerType == \bigcup\{ManagedPointerType,$
$\quad UnmanagedPointerType, FunctionPointerType\}$

Z: 4.20
$PassableType == \bigcup\{UnboxedValueType,$
$\quad ObjectReferenceType, ManagedPointerType,$
$\quad UnmanagedPointerType, FunctionPointerType\}$

Z: 4.21
$CompoundComponentType ==$
$\quad PassableType \setminus ManagedPointerType \setminus \{Ubvt(C_R)\}$

Z: 4.22
$ReturnableType == PassableType \cup VoidType$

Z: 4.23
$DelegateType == \{t : AtomicNamedType \mid Atomic^{\sim}(t) \ \underline{Inherits} \ C_D\}$

Undefined: *Type*, *Atomic*, *Ubvt*, *ObjRef*, *ManPtr*, *UnmanPtr*, *FnPtr*

Theorem 4.1
$dom\, Elementary = NamedType$

Theorem 4.2
$\forall e_1, e_2 : Elementary \mid first(e_1) \neq first(e_2) \bullet$
$\quad second(e_1) \neq second(e_2)$

Type introduces a given set. *NamedType* and *Type* are disjoint, and they are of different meta-types; in particular, *NamedType* is not a subset of *Type*. However, through the *Atomic* total injection, each named type is associated with a distinct type. *Atomic*, *Ubvt*, *ObjRef*, *ManPtr*, *UnmanPtr*, and *FnPtr* introduce total injections associating each element of their domains with a type. See Figures 4.1 to 4.8.

A type is a void type, or an atomic named type, or an unboxed value type, or an object reference type, or a managed pointer type, or an unmanaged pointer type, or a function pointer type.

The return type of a method may be void (ECMA II 22.2.11). An unmanaged pointer may be a pointer to void (ECMA II 22.2.12, Appendix

D (II 22.2.12)). A pointer type is a managed pointer type, an unmanaged pointer type, or a function pointer type (Appendix D (II 22.2.12)). Since each function pointer type is associated with a non-abstract method, all information in the method signature is associated with the function pointer type (Section 5.3). A passable type is a type that can be a parameter type (Section 5.2). A compound component type is a type that can be an array element type or a field signature type (Sections 4.4.2 and 5.2). A returnable type is a type that can be a method return type. (Section 5.2). A delegate type is a type that can be an event signature. (Section 5.2).

An elementary type is a void type, or an atomic named type, or an unboxed value type, or an object reference type. There may exist types that are not built ultimately from elementary types (Section 6.2). (A function pointer type is not an elementary type, because it is built from the return type of a method and the types of the parameters.)

The definition of *Type* is similar to a Z free-type definition, where *Atomic*, *Ubvt*, *ObjRef*, *ManPtr*, *UnmanPtr*, and *FnPtr* are called *type builders*. (Z uses the terminology *constructor*; we use *builder* to avoid overloading *constructor*.) We cannot explicitly use a free-type definition, because the introduction of *ManPtr* relies on *Ubvt* and *ObjRef*, which would not be declared until after the free-type definition.

Type is an infinite set, even if *NamedType* is finite: the type builder *UnmanPtr* may be repeatedly applied to its own output (Figure 4.7). Compare this with the generation of the set of natural numbers by repeated application of *successor*. Since *Type* is an infinite set, and its definition is similar to a recursive free-type, we need to satisfy ourselves that it is well-formed. [3] gives a rule of thumb, which we paraphrase as: *ensure that every element of the expression representing the domain of each total injection is made from a finite number of elements of Type*. Our definition passes this test.

Example: since each non-abstract named type is also a named type, it is associated with two distinct types: an atomic named type, through the *Atomic* total injection; and an object reference type, through the *ObjRef* total injection (Figure 4.2).

Example: each unboxed value type and each object reference type are associated through the *ManPtr* total injection with one managed pointer type (Figures 4.5 and 4.6).

Type builders are restricted, as follows. *Void* is given (ECMA Library). An atomic named type is built from a named type. An unboxed value type is built from a value type. An object reference type is built from a non-abstract named type (ECMA I 7.9.6.2, I 11.1, I 11.1.1.2). A managed pointer type is built from an unboxed value type or an object reference type. See Appendix D (I 11.1.1.2 and II 13.4.2) for information about where a managed pointer can point, what it can point to, and where it can be stored. An unmanaged pointer type is built from any type (ECMA II 13.4.1). A function pointer type is built from a non-abstract method (ECMA II 7.1, II 13.5). We do not

model function pointer types that are not related to methods, for example, native functions called via "platform invoke".

4.2 Instance

Z: 4.24
$[\mathit{Instance}]$

Z: 4.25
$\mathit{Null} : \mathit{Instance}$

Z: 4.26
$\mathit{NullInstance} == \{\mathit{Null}\}$

Z: 4.27
$\mathit{ObjectInst} : (\mathit{Object} \cup \mathit{Value}) \rightarrowtail \mathit{Instance}$

Z: 4.28
$\mathit{ObjectInstance} == \mathrm{ran}\ \mathit{ObjectInst}$

Z: 4.29
$\mathit{UbvtInst} : \mathit{Value} \rightarrowtail \mathit{Instance}$

Z: 4.30
$\mathit{UnboxedValueInstance} == \mathrm{ran}\ \mathit{UbvtInst}$

Z: 4.31
$\mathit{ObjRefInst} : \mathit{ObjectReference} \rightarrowtail \mathit{Instance}$

Z: 4.32
$\mathit{ObjectReferenceInstance} == \mathrm{ran}\ \mathit{ObjRefInst}$

Z: 4.33
$\mathit{ManPtrInst} : (\mathit{UnboxedValueInstance} \cup$
$\quad \mathit{ObjectReferenceInstance}) \rightarrowtail \mathit{Instance}$

Z: 4.34
$ManagedPointerInstance == \operatorname{ran} ManPtrInst$

Z: 4.35
$UnmanPtrInst : Instance \rightarrowtail Instance$

Z: 4.36
$UnmanagedPointerInstance == \operatorname{ran} UnmanPtrInst$

Z: 4.37
$FnPtrInst : NonAbstractMethod \rightarrowtail Instance$

Z: 4.38
$FunctionPointerInstance == \operatorname{ran} FnPtrInst$

Z: 4.39
$$\langle NullInstance, ObjectInstance, UnboxedValueInstance, ObjectReferenceInstance, ManagedPointerInstance, UnmanagedPointerInstance, FunctionPointerInstance \rangle \text{ partitions } Instance$$

Undefined: *Instance*, *ObjectInst*, *UbvtInst*, *ObjRefInst*, *ManPtrInst*, *UnmanPtrInst*, *FnPtrInst*

We take the general notion of *instance of type* to be sufficiently well understood to form a basis for discussion, and we specialize the notion with a definition of what it means for i to be an instance of each type.

Instance introduces a given set. *Value* and *Object* are different metatypes from *Instance*; in particular, they are not subsets of *Instance*. However, through the *ObjectInst* total injection, each value and each object is associated with a distinct instance. *ObjectInst*, *UbvtInst*, *ObjRefInst*, *ManPtrInst*, *UnmanPtrInst*, and *FnPtrInst* introduce total injections associating each element of their domains with an instance. See Figures 4.1 to 4.8.

An instance is a null instance, or an object instance, or an unboxed value instance, or an object reference instance, or a managed pointer instance, or an unmanaged pointer instance, or a function pointer instance.

The definition of *Instance* is similar to a Z free-type definition, where *ObjectInst*, *UbvtInst*, *ObjRefInst*, *ManPtrInst*, *UnmanPtrInst*, and *FnPtrInst* are called *instance builders.* (Z uses the terminology *constructor*; we use

builder to avoid overloading *constructor*.) We cannot explicitly use a free-type definition, because the introduction of *ManPtrInst* relies on *UbvtInst* and *ObjRefInst*, which would not be declared until after the free-type definition.

Instance is an infinite set, even if *Object* and *Value* are finite: the instance builder *UnmanPtrInst* may be repeatedly applied to its own output (Figure 4.7). Since *Instance* is an infinite set, and its definition is similar to a recursive free-type, we need to satisfy ourselves that it is well-formed. [3] gives a rule of thumb, which we paraphrase as: *ensure that every element of the expression representing the domain of each total injection is made from a finite number of elements of Instance.* Our definition passes this test.

Example: each object reference is associated with two distinct instances: an object reference instance, through the *ObjRefInst* total injection; and an object instance, through the *ConventionalObjectIdentity* bijection and the *ObjectInst* total injection (Figure 4.2).

Example: each unboxed value instance and each object reference instance is associated with one managed pointer instance, through the *ManPtrInst* total injection (Figures 4.5 and 4.6).

Example: each instance (of any type, including an unmanaged pointer instance) is associated with one unmanaged pointer instance, through the *UnmanPtrInst* total injection. See Appendix E.7 for a discussion of whether an unmanaged pointer instance can point into the garbage collected heap.

Instance builders are restricted as follows. *Null* is given. An object instance is built from an object or a value. An unboxed value instance is built from a value. An object reference instance is built from an object reference. A managed pointer instance is built from an unboxed value instance or an object reference instance. See Appendix D (I 11.1.1.2 and II 13.4.2) for information about what a managed pointer can point to. An unmanaged pointer instance is built from any instance. A function pointer instance is built from a non-abstract method.

See Appendix B.3 for a discussion of the real-world representation of instances.

4.3 Instance Of Type

Z: 4.40

$NullInstanceOf : Instance \leftrightarrow Type$

$\mathrm{dom}\ NullInstanceOf = NullInstance$

$\mathrm{ran}\ NullInstanceOf = Type \setminus ManagedPointerType \setminus VoidType$

Z: 4.41
$InstanceOf : Instance \leftrightarrow Type$

Z: 4.42
$$ObjectInstanceOf == ObjectInst^{\sim} \mathbin{;} ExactInstanceOf \mathbin{;} Inherits^{*} \mathbin{;} Atomic$$

Z: 4.43
$$UnboxedValueInstanceOf == UbvtInst^{\sim} \mathbin{;} ExactInstanceOf \mathbin{;} Ubvt$$

Z: 4.44
$$ObjectReferenceInstanceOf == ObjRefInst^{\sim} \mathbin{;} ConventionalObjectIdentity \mathbin{;} ExactInstanceOf \mathbin{;} ObjRef$$

Z: 4.45
$$ManagedPointerInstanceOf == ManPtrInst^{\sim} \mathbin{;} (UnboxedValueInstanceOf \cup ObjectReferenceInstanceOf) \mathbin{;} ManPtr$$

Z: 4.46
$$UnmanagedPointerInstanceOf == UnmanPtrInst^{\sim} \mathbin{;} InstanceOf \mathbin{;} UnmanPtr$$

Z: 4.47
$$FunctionPointerInstanceOf == FnPtrInst^{\sim} \mathbin{;} FnPtr$$

Z: 4.48
$$\begin{aligned}
&\forall i : Instance;\ t : Type \bullet \\
&\quad i \mapsto t \in InstanceOf \Leftrightarrow \\
&\qquad (i \mapsto t \in NullInstanceOf \lor \\
&\qquad i \mapsto t \in ObjectInstanceOf \lor \\
&\qquad i \mapsto t \in UnboxedValueInstanceOf \lor \\
&\qquad i \mapsto t \in ObjectReferenceInstanceOf \lor \\
&\qquad i \mapsto t \in ManagedPointerInstanceOf \lor \\
&\qquad i \mapsto t \in UnmanagedPointerInstanceOf \lor \\
&\qquad i \mapsto t \in FunctionPointerInstanceOf)
\end{aligned}$$

A null instance is an instance of any type, except a managed pointer type (ECMA II 13.4.2) or a void type. For all other types we define *xInstanceOf* in terms of compositions of previously defined relations. For example, we

define *ObjectInstanceOf* by following the inverse of the *ObjectInst* total injection from *Instance* to $Value \cup Object$, then the *ExactInstanceOf* total function to *NamedType*, then the reflexive transitive closure of *Inherits*, then the *Atomic* total injection to *Type*. See Figures 4.1 to 4.8.

Note that Z4.41 introduces *InstanceOf* and Z4.48 defines its value. We normally introduce a constant and define its value in the same axiomatic description. In this case we must defer the value-defining predicate until after we introduce *UnmanagedPointerInstanceOf*.

Some types may have no instances: for example, an atomic named type that is associated with an abstract class that is not inherited by any non-abstract class (Section 2.4).

The definition of *UnmanagedPointerInstanceOf* is inductive: Z4.46 defines it in terms of *InstanceOf*, while Z4.41 introduces *InstanceOf* and Z4.48 defines its value in terms of *UnmanagedPointerInstanceOf*. The definition is not circular, since *InstanceOf* resolves ultimately to an instance of an elementary type. Although the definition is inductive, the introductions in Z4.46 and Z4.41 are not: Z does not allow recursive declarations, except in free-types.

An unmanaged pointer instance may point to an instance of a void type. A method return type may be a void type (Section 5.2). A void type does not occur in any other context.

When we follow relational compositions containing *Inherits* and *InstanceOf* in Figures 4.1 to 4.8, we follow all branches of the relations. Thus: an object instance may be an instance of more than one type, because $Inherits^*$ has multiple branches; and an unmanaged pointer instance may be an instance of more than one type, because *InstanceOf* has multiple branches. An unboxed value instance, or an object reference instance, or a managed pointer instance, or a function pointer instance is never an instance of more than one type.

4.4 Array

4.4.1 Array Shape

Z: 4.49
$LowerBound == \mathbb{Z}$

Z: 4.50
$UpperBound == \mathbb{Z}$

Z: 4.51
$Dimension == LowerBound \times UpperBound$

Z: 4.52

$Rank == \mathbb{N}_1$

Z: 4.53

$ArrayShape == \mathrm{seq}_1\ Dimension$

Z: 4.54

$\forall d : Dimension \bullet first(d) \leq second(d)$

Z: 4.55

$HasRank : ArrayShape \to Rank$

$\forall s : ArrayShape;\ r : Rank \bullet s \mapsto r \in HasRank \Leftrightarrow \#s = r$

Z: 4.56

$NthLowerBound : ArrayShape \times \mathbb{N}_1 \to LowerBound$

$$\begin{aligned}&\forall s : ArrayShape;\ n : \mathbb{N}_1;\ l : LowerBound \mid\\&\quad n \leq HasRank(s) \bullet\\&\quad (s \mapsto n) \mapsto l \in NthLowerBound \Leftrightarrow first(s(n)) = l\end{aligned}$$

Z: 4.57

$NthUpperBound : ArrayShape \times \mathbb{N}_1 \to UpperBound$

$$\begin{aligned}&\forall s : ArrayShape;\ n : \mathbb{N}_1;\ u : UpperBound \mid\\&\quad n \leq HasRank(s) \bullet\\&\quad (s \mapsto n) \mapsto u \in NthUpperBound \Leftrightarrow second(s(n)) = u\end{aligned}$$

Z: 4.58

$NthSize : ArrayShape \times \mathbb{N}_1 \to \mathbb{N}_1$

$$\begin{aligned}&\forall s : ArrayShape;\ n_1, n_2 : \mathbb{N}_1 \mid\\&\quad n_1 \leq HasRank(s) \bullet\\&\quad (s \mapsto n_1) \mapsto n_2 \in NthSize \Leftrightarrow\\&\qquad n_2 = NthUpperBound(s, n_1) -\\&\qquad\quad NthLowerBound(s, n_1) + 1\end{aligned}$$

Z: 4.59

$$NthProductSize : ArrayShape \times \mathbb{N}_1 \rightarrow \mathbb{N}_1$$

$$\begin{aligned}
&\forall\, s : ArrayShape;\ n_1, n_2 : \mathbb{N}_1 \mid \\
&\quad n_1 \leq HasRank(s) \bullet \\
&\quad (s \mapsto n_1) \mapsto n_2 \in NthProductSize \Leftrightarrow \\
&\quad (\\
&\qquad (n_1 = 1 \land n_2 = NthSize(s, n_1)) \lor \\
&\qquad (n_1 > 1 \land n_2 = NthSize(s, n_1) * \\
&\qquad\quad NthProductSize(s, n_1 - 1)) \\
&\quad)
\end{aligned}$$

Each array shape consists of a finite, non-empty sequence of dimensions, where each dimension is a pair of integers: a lower bound and an upper bound (ECMA I 7.9.1). ECMA does not explicitly specify that a lower bound must be less than or equal to an upper bound; however, we assume that it is implied by the terminology *lower* and *upper*. It is perhaps unclear whether ECMA permits a lower bound to be equal to an upper bound; we assume it does. The number of dimensions is called the rank.

NthLowerBound is the lower bound of the nth dimension. *NthUpperBound* is the upper bound of the nth dimension. *NthSize* is the size of the nth dimension. *NthProductSize* is the product of all dimension sizes up to and including the nth dimension.

We have declared *HasRank*, *NthLowerBound*, *NthUpperBound*, *NthSize*, and *NthProductSize* to be total functions, and we assert that they are defined constants, but we have not proven that the associated predicates constitute definitions of total functions.

4.4.2 Array Type and Array Instance

Z: 4.60

$$ArrayType == \{t : AtomicNamedType \mid Atomic^{\sim}(t)\ \underline{Inherits}\ C_A\}$$

Z: 4.61

$$ArrayElementType == CompoundComponentType$$

Z: 4.62

$$ArrayInstance == \mathrm{dom}(ObjectInstanceOf \rhd ArrayType)$$

Z: 4.63

$$ArrayElementInstance == \mathrm{dom}(InstanceOf \rhd ArrayElementType)$$

Z: 4.64
$Ary : ArrayShape \times ArrayElementType \rightarrowtail\!\!\!\!\twoheadrightarrow ArrayType$

Z: 4.65
$AryInst : ArrayShape \times \mathrm{seq}_1\ ArrayElementInstance \rightarrowtail\!\!\!\!\twoheadrightarrow$
$\quad ArrayInstance$

Z: 4.66
$\forall s : ArrayShape;\ i : \mathrm{seq}_1\ ArrayElementInstance \mid$
$\quad s \mapsto i \in \mathrm{dom}\ AryInst \bullet$
$\quad NthProductSize(s, HasRank(s)) = \#i$

Z: 4.67
$ArrayInstanceOf : ArrayInstance \leftrightarrow$
$\quad ArrayShape \times ArrayElementType$

$\forall a : ArrayInstance;\ s : ArrayShape;\ t : ArrayElementType \bullet$
$\quad a \mapsto (s \mapsto t) \in ArrayInstanceOf \Leftrightarrow$
$\quad (\exists i : \mathrm{seq}_1\ ArrayElementInstance \bullet$
$\quad\quad (\forall x : \mathrm{ran}\ i \bullet x\ \underline{InstanceOf}\ t) \land$
$\quad\quad a = AryInst(s \mapsto i)\quad)$
$\forall a : ArrayInstance \bullet$
$\quad (\exists t : ArrayElementType;\ s : ArrayShape \bullet$
$\quad\quad a\ \underline{ArrayInstanceOf}\ s \mapsto t)$

Undefined: *Ary*, *AryInst*

Each array type is a special case of an atomic named type: it is an atomic named type related via the *Atomic* total injection to a named type that inherits *C_A*. An array element type is a compound component type (Section 4.1, Appendix D (I 7.9.1)). The bijection *Ary* relates each pair (array shape, array element type) to a distinct array type. Full shape information determines an array type (ECMA I 7.9.1, Appendix D (I 7.9.1 and II 13.2)).

Each array instance is a special case of an object instance: it is an object instance of an array type. An array element instance is an instance of an array element type. The bijection *AryInst* relates each pair (array shape, finite sequence of array element instances) to a distinct array instance.

Each array instance a is related via *ArrayInstanceOf* to one or more pairs (array shape s, array element type t), such that every array element

instance comprising a is an instance of t. There may be more than one t satisfying this constraint, because all array element instances comprising a may be instances of more than one type. A subsequent constraint ensures that at least one t exists. These constraints ensure that all array element instances in a particular array are instances of the same array element type (ECMA I 7.9.1).

The length of the sequence of array element instances is the same as the product of the dimension sizes of the array shape.

4.5 Boxing

Z: 4.68
$BoxedValueType == \mathrm{ran}(ValueType \lhd Atomic)$

Z: 4.69
$BoxedValueInstance == \mathrm{ran}(Value \lhd ObjectInst)$

Z: 4.70
$BoxedValueInstanceOf == ObjectInstanceOf \rhd BoxedValueType$

Z: 4.71
$BoxingType == Ubvt^{\sim} \mathbin{;} Atomic$

Z: 4.72
$Box == UbvtInst^{\sim} \mathbin{;} ObjectInst$

Theorem 4.3
$BoxedValueType \subset AtomicNamedType$

Theorem 4.4
$BoxedValueInstance \subset ObjectInstance$

Theorem 4.5
$dom\, BoxedValueInstanceOf = BoxedValueInstance$

Theorem 4.6
$ran\, BoxingType = BoxedValueType$

Theorem 4.7
$ran\, Box = BoxedValueInstance$

A boxed value type is a special case of an atomic named type. A boxed value instance is a special case of an object instance.

Each value type is associated with two distinct types: an unboxed value type (via *Ubvt*) and an atomic named type (via *Atomic*) (Figure 4.3). We introduce *BoxedValueType* as a synonym for *AtomicNamedType* when the associated named type is a value type. *BoxingType* is a total injection associating each unboxed value type with a distinct boxed value type. *Box* is a total injection associating each unboxed value instance with a distinct boxed value instance. *Box* is the same as the ECMA *box* operation. However, the inverse of *Box* is not the same as ECMA *unbox*, because ECMA *unbox* returns a managed pointer.

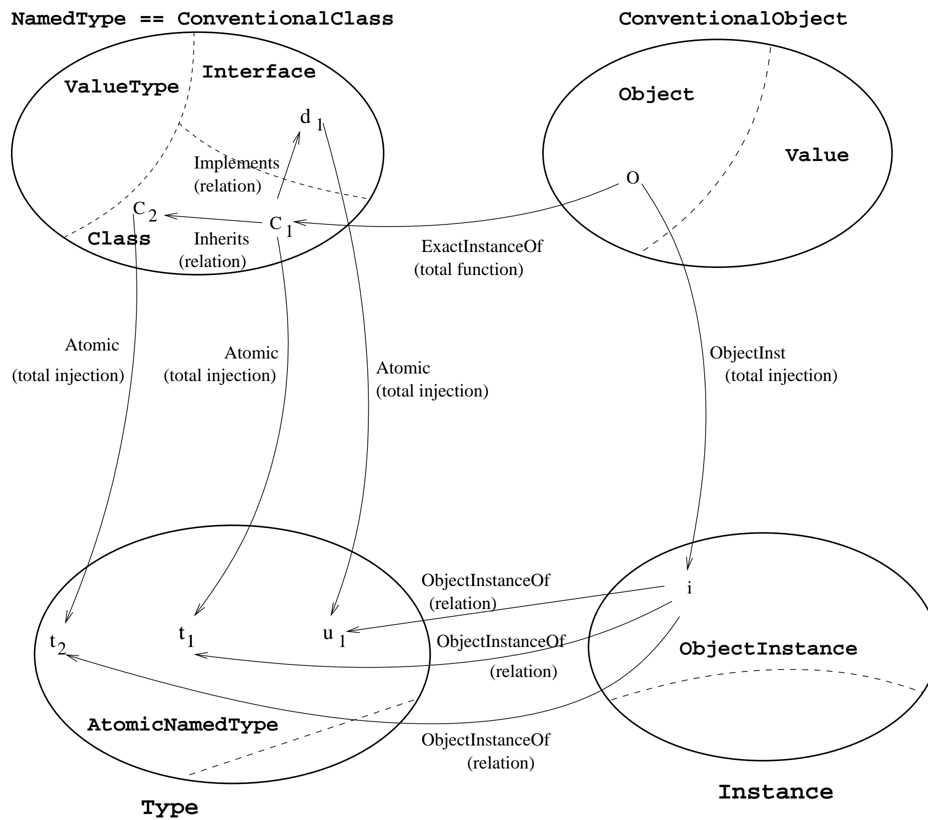

Figure 4.1: Instance of Object

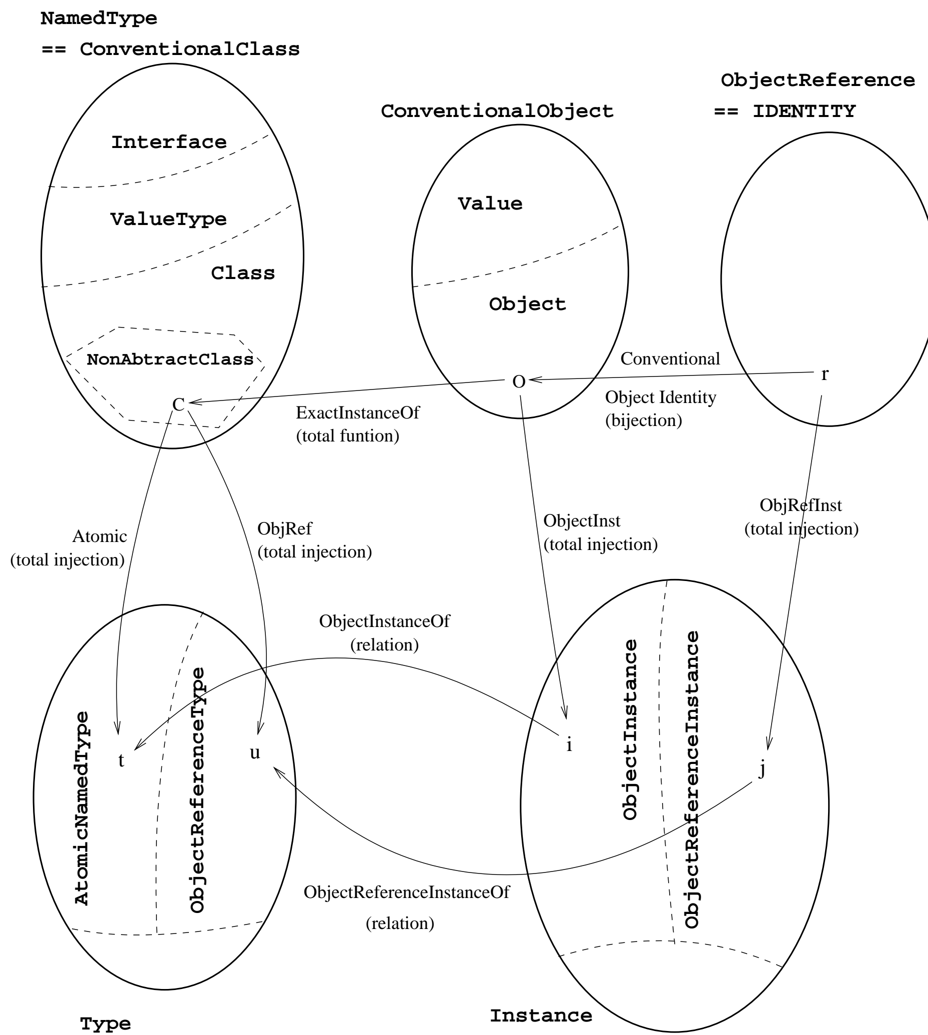

Figure 4.2: Instance of Object Reference

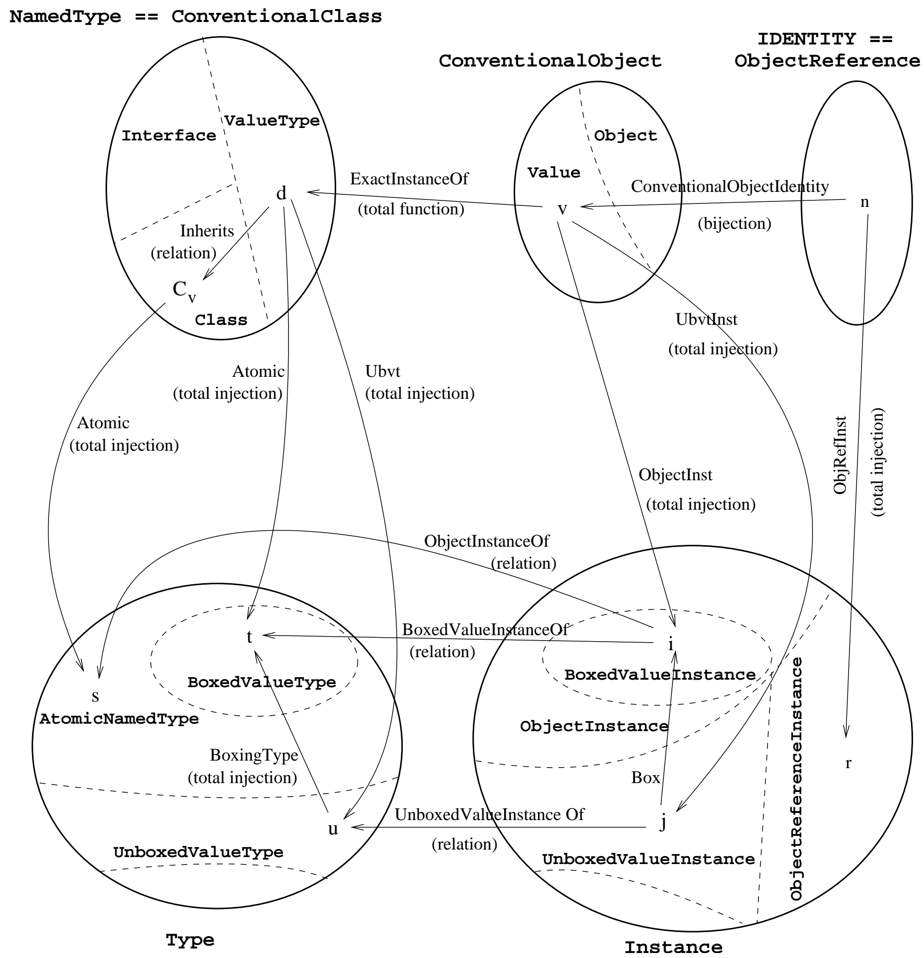

Figure 4.3: Instance of Boxed and Unboxed Value

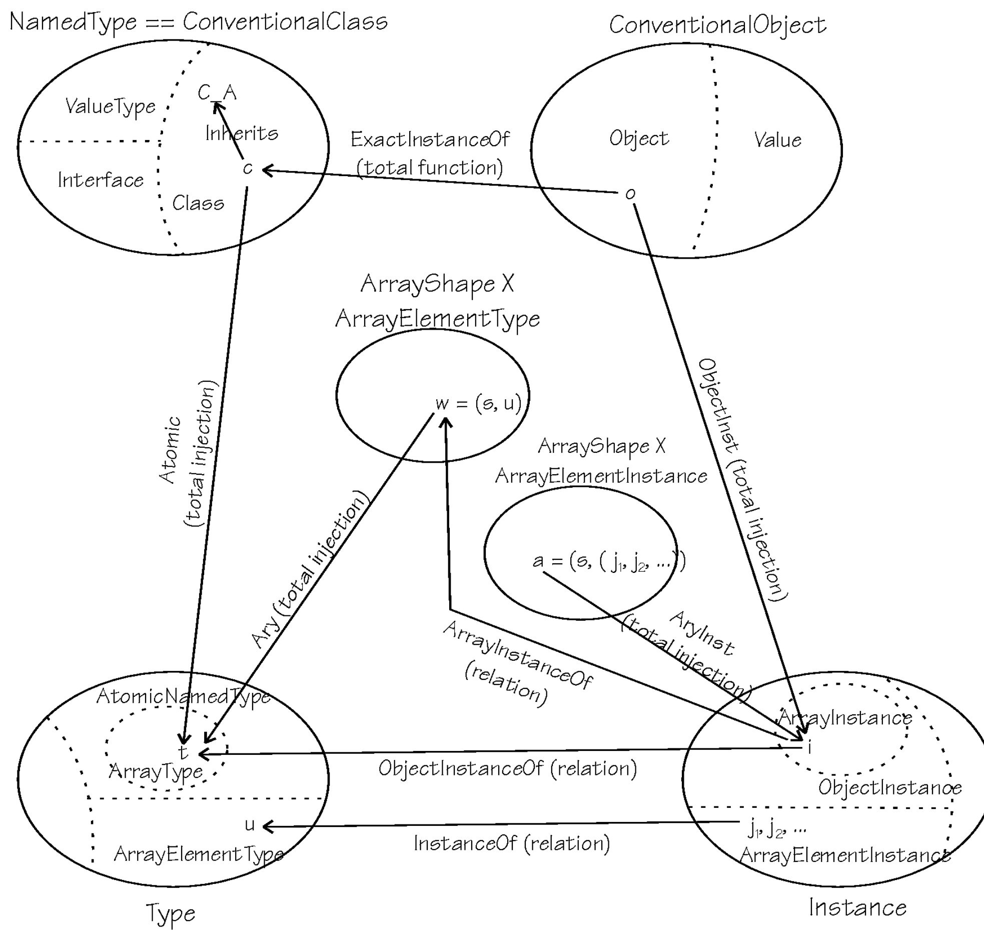

Figure 4.4: Instance of Array

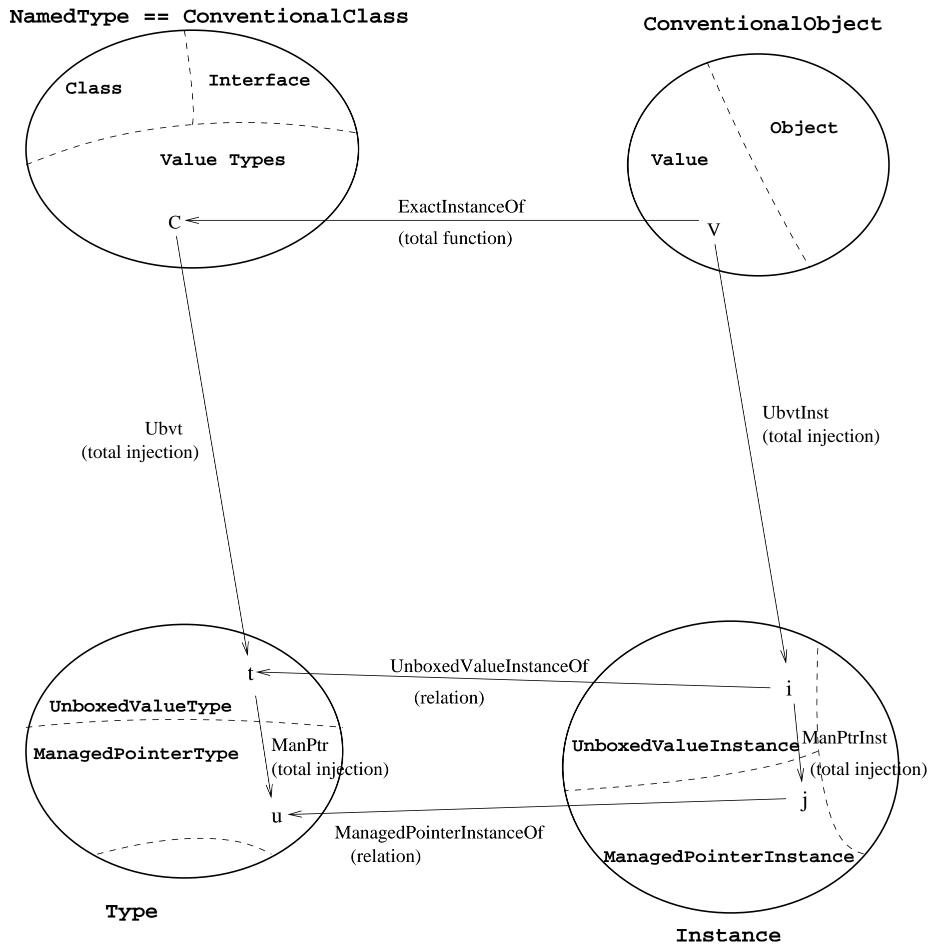

Figure 4.5: Instance of Managed Pointer 1

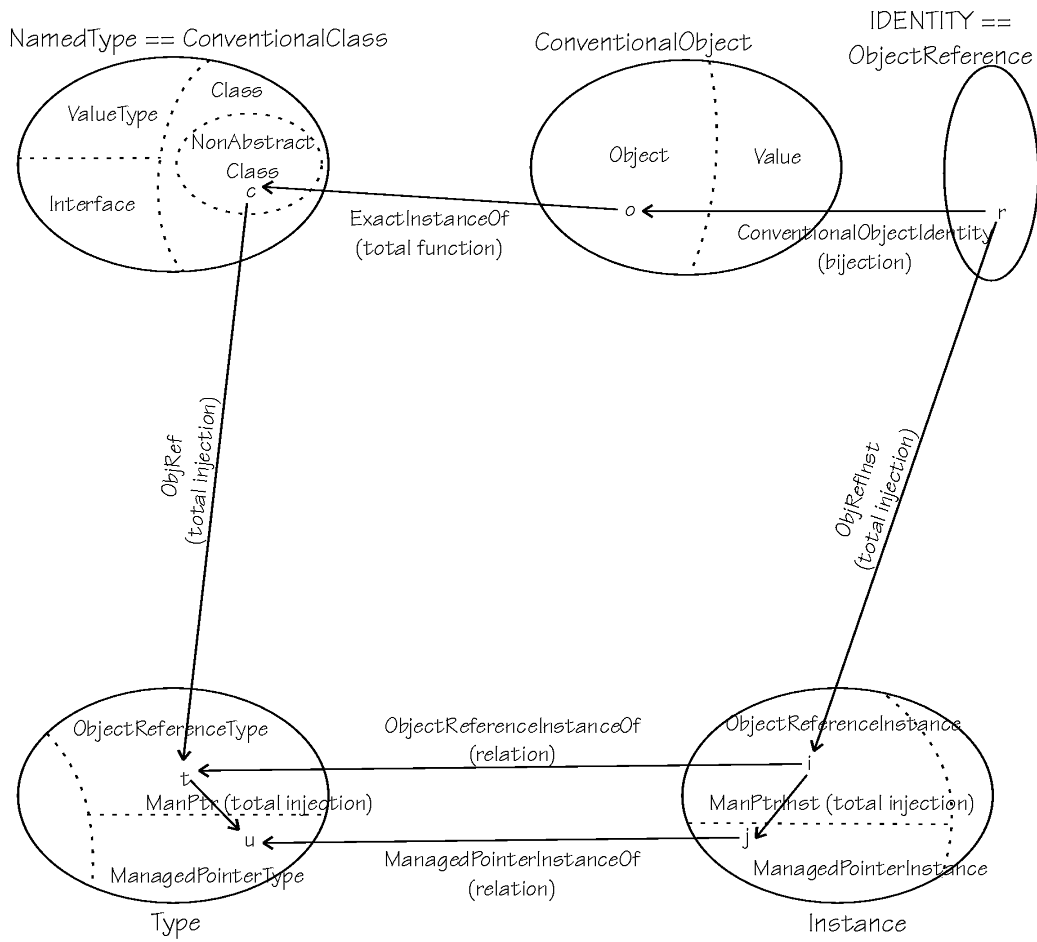

Figure 4.6: Instance of Managed Pointer 2

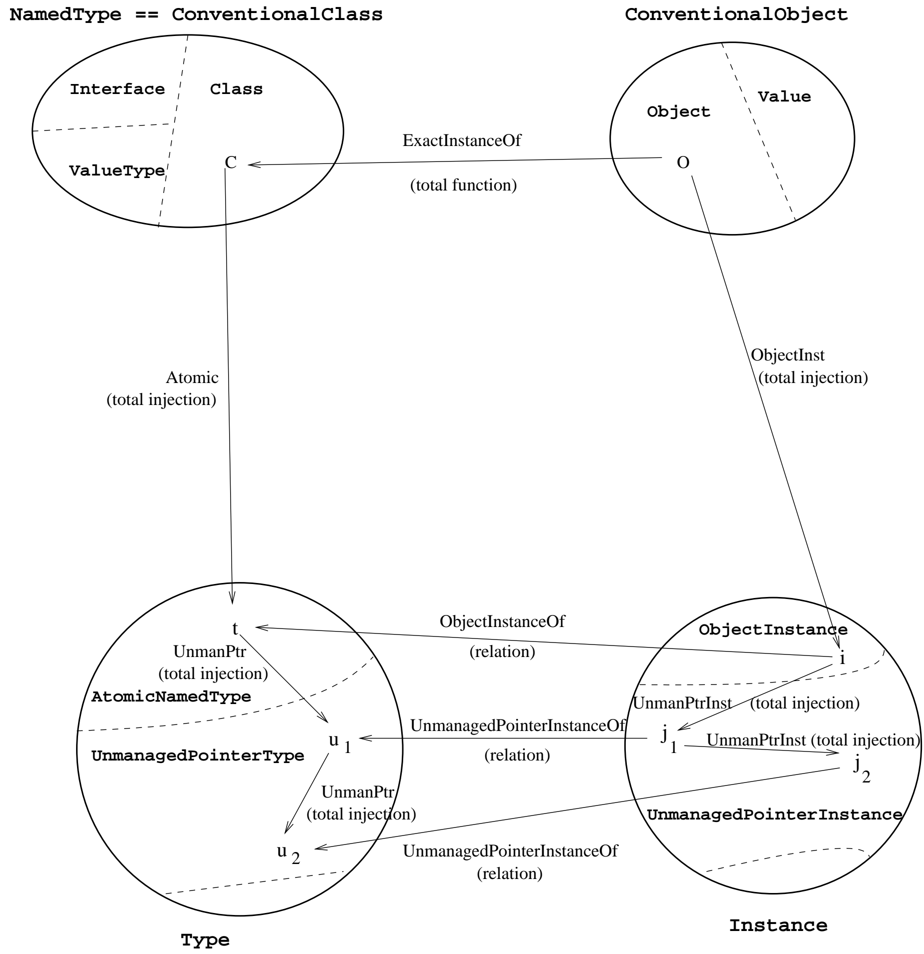

Figure 4.7: Instance of Unmanaged Pointer

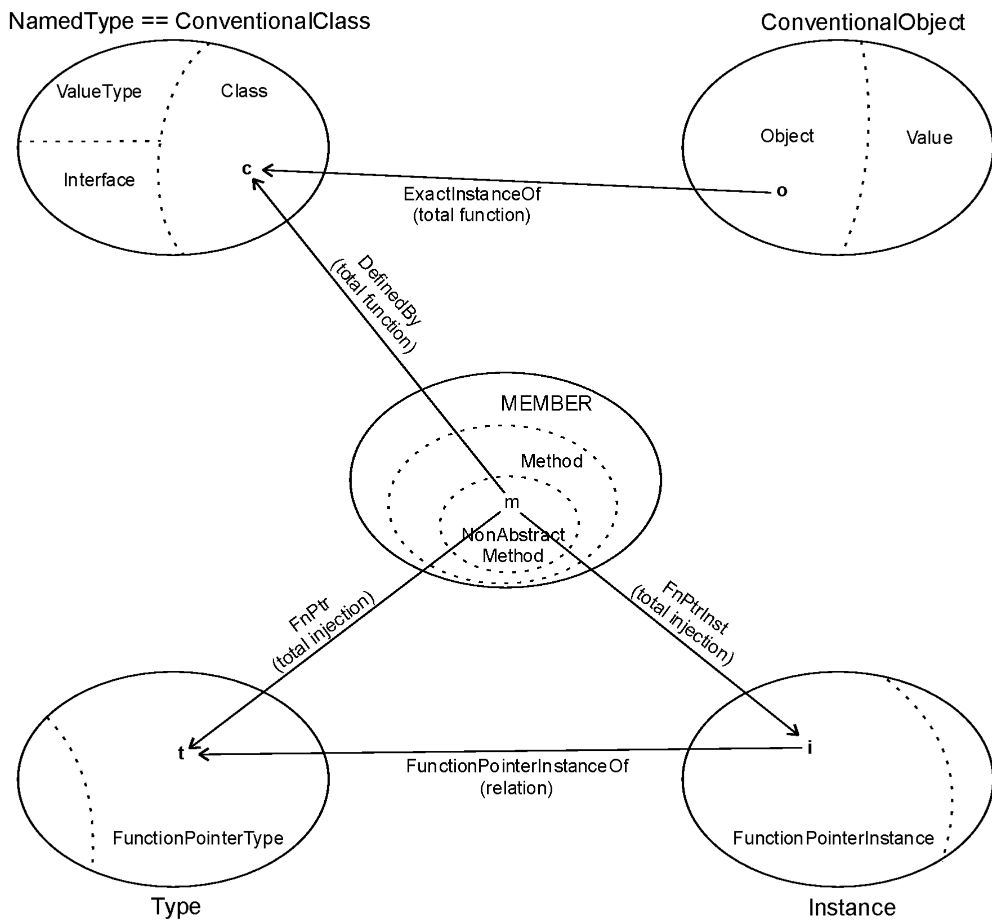

Figure 4.8: Instance of Function Pointer

Chapter 5

Signature

ECMA II 22.2 defines signatures in terms of a sequence of compressed bytes and a grammar that assigns meaning to the bytes. The definition of signature in our model is a subset of ECMA's definition; it defines only those features of signature needed in our model. ECMA II 22.2 does not define event signatures. For consistency and completeness, our model defines event signatures although, in the real world, it may be necessary to deduce their signatures from the signatures of methods.

5.1 Calling Convention

Z: 5.1

$$
\begin{aligned}
&\mathit{CallingConvention} ::= \mathit{DefaultCallingConvention} \mid \\
&\quad \mathit{VarargCallingConvention} \mid \\
&\quad \mathit{CCallingConvention} \mid \\
&\quad \mathit{StdcallCallingConvention} \mid \\
&\quad \mathit{ThiscallCallingConvention} \mid \\
&\quad \mathit{FastcallCallingConvention}
\end{aligned}
$$

Calling convention specifies the transport mechanism for method arguments (ECMA II 14.2). ECMA specifies other components of calling convention, outside the scope of our model.

5.2 Signature as a Sequence of Types

Z: 5.2

$\mathit{ParameterTypes} == \operatorname{seq} \mathit{PassableType}$

Z: 5.3

$$\begin{aligned}&\mathit{Signature} ::= \\&\quad \mathit{MethSig}\langle\langle \mathit{CallingConvention} \times \mathit{ReturnableType} \times \\&\qquad \mathit{ParameterTypes}\rangle\rangle \mid \\&\quad \mathit{FldSig}\langle\langle \mathit{CompoundComponentType}\rangle\rangle \mid \\&\quad \mathit{EvSig}\langle\langle \mathit{DelegateType}\rangle\rangle \mid \\&\quad \mathit{PropSig}\langle\langle \mathit{PassableType} \times \mathit{ParameterTypes}\rangle\rangle\end{aligned}$$

Z: 5.4

$\mathit{MethodSignature} == \operatorname{ran} \mathit{MethSig}$

Z: 5.5

$\mathit{FieldSignature} == \operatorname{ran} \mathit{FldSig}$

Z: 5.6

$\mathit{EventSignature} == \operatorname{ran} \mathit{EvSig}$

Z: 5.7

$\mathit{PropertySignature} == \operatorname{ran} \mathit{PropSig}$

Undefined: $\mathit{Signature}, \mathit{MethSig}, \mathit{FldSig}, \mathit{EvSig}, \mathit{PropSig}$

A method signature is built from a calling convention, a returnable type, and parameter types (ECMA II 22.2.1, 22.2.2, 22.2.3). A field signature is built from a compound component type (ECMA II 22.2.4). An event signature is built from a delegate type. A property signature is built from a passable type and parameter types (ECMA II 22.2.5). ECMA specifies other signature components outside the scope of our model. Language syntax determines the construction of signatures.

5.3 Member Signatures

Z: 5.8

$\mathit{HasMethodSignature} : \mathit{Method} \rightarrow \mathit{MethodSignature}$

Z: 5.9

$\mathit{HasFieldSignature} : \mathit{Field} \rightarrow \mathit{FieldSignature}$

Z: 5.10
$HasEventSignature : Event \rightarrow EventSignature$

Z: 5.11
$HasPropertySignature : Property \rightarrow PropertySignature$

Z: 5.12
$$HasMemberSignature == \bigcup\{HasMethodSignature, HasFieldSignature, HasEventSignature, HasPropertySignature\}$$

Undefined: *HasMethodSignature*, *HasFieldSignature*, *HasEventSignature*, *HasPropertySignature*

Each method, field, event and property has one signature. Multiple methods may have the same signature; so may fields, events, and properties. Each signature is the signature of at least one member.

5.4 Signature Components

Z: 5.13
$MethodCallingConvention : Signature \twoheadrightarrow CallingConvention$

$$\forall s : Signature;\ c : CallingConvention \bullet (s \mapsto c \in MethodCallingConvention \Leftrightarrow (\exists t : Type;\ p : ParameterTypes \bullet MethSig(c, t, p) = s))$$

Z: 5.14
$MethodReturnType : Signature \twoheadrightarrow ReturnableType$

$$\forall s : Signature;\ t : ReturnableType \bullet (s \mapsto t \in MethodReturnType \Leftrightarrow (\exists c : CallingConvention;\ p : ParameterTypes \bullet MethSig(c, t, p) = s))$$

Z: 5.15

$$MethodParameterTypes : Signature \rightarrowtail\!\!\!\!\!\!+ ParameterTypes$$

$$\forall s : Signature;\ p : ParameterTypes \bullet \\ \quad (s \mapsto p \in MethodParameterTypes \\ \quad \Leftrightarrow (\exists c : CallingConvention;\ t : Type \bullet \\ \qquad MethSig(c, t, p) = s))$$

Z: 5.16

$$FieldType : Signature \rightarrowtail\!\!\!\!\!\!+ CompoundComponentType$$

$$FieldType = FldSig^{\sim}$$

Z: 5.17

$$EventType : Signature \rightarrowtail\!\!\!\!\!\!+ DelegateType$$

$$EventType = EvSig^{\sim}$$

Z: 5.18

$$PropertyType : Signature \rightarrowtail\!\!\!\!\!\!+ PassableType$$

$$\forall s : Signature;\ t : PassableType \bullet \\ \quad (s \mapsto t \in PropertyType \\ \quad \Leftrightarrow (\exists p : ParameterTypes \bullet \\ \qquad PropSig(t, p) = s))$$

Z: 5.19

$$PropertyParameterTypes : Signature \rightarrowtail\!\!\!\!\!\!+ ParameterTypes$$

$$\forall s : Signature;\ p : ParameterTypes \bullet \\ \quad (s \mapsto p \in PropertyParameterTypes \\ \quad \Leftrightarrow (\exists t : Type \bullet \\ \qquad PropSig(t, p) = s))$$

Z: 5.20

$$
\begin{array}{l}
\mathit{ContainsType} : \mathit{Signature} \leftrightarrow \mathit{Type} \\
\hline
\forall s : \mathit{Signature};\ t : \mathit{Type} \bullet \\
\quad (s \mapsto t \in \mathit{ContainsType} \Leftrightarrow \\
\quad (\\
\quad\quad (s \underline{\mathit{MethodReturnType}}\ t) \lor \\
\quad\quad (s \underline{\mathit{FieldType}}\ t) \lor \\
\quad\quad (s \underline{\mathit{EventType}}\ t) \lor \\
\quad\quad (s \underline{\mathit{PropertyType}}\ t) \lor \\
\quad\quad (\exists p : \mathit{ParameterTypes} \bullet \\
\quad\quad\quad (\\
\quad\quad\quad\quad ((s \underline{\mathit{MethodParameterTypes}}\ p) \lor \\
\quad\quad\quad\quad\quad (s \underline{\mathit{PropertyParameterTypes}}\ p) \\
\quad\quad\quad\quad) \land \\
\quad\quad\quad\quad t \in \operatorname{ran} p \\
\quad\quad\quad) \\
\quad\quad) \\
\quad) \\
\quad)
\end{array}
$$

MethodCallingConvention, *MethodReturnType*, etc extract the components from each signature.

We have declared *MethodCallingConvention*, *MethodReturnType*, *MethodParameterTypes*, *FieldType*, *EventType*, *PropertyType*, and *PropertyParameterTypes* to be partial functions, and we assert that they are defined constants, but we have not proven that the associated predicates constitute definitions of partial functions.

Chapter 6

Signature Dependent Constraints

6.1 Enum Fields

Z: 6.1

$$\begin{array}{l} \mathit{EnumValueField} == \{ f : \mathit{Field} \mid \\ \quad \mathit{DefinedBy}(f) \in \mathit{EnumType} \land \\ \quad \mathit{Atomic}^{\sim}(\mathit{FieldType}(\mathit{HasFieldSignature}(f))) \in \mathit{IntegerType} \land \\ \quad \neg\, (f \ \underline{\mathit{HasFieldAttribute}} \ \mathit{StaticFieldAttribute})\} \end{array}$$

Z: 6.2

$$\begin{array}{l} \mathit{EnumLiteralField} == \{ f : \mathit{Field} \mid \\ \quad \mathit{DefinedBy}(f) \in \mathit{EnumType} \land \\ \quad f \ \underline{\mathit{HasFieldAttribute}} \ \mathit{LiteralFieldAttribute}\} \end{array}$$

Z: 6.3

$$\begin{array}{l} \forall\, t : \mathit{EnumType} \bullet \\ \quad \exists_1 f : \mathit{EnumValueField} \bullet f \ \underline{\mathit{DefinedBy}} \ t \end{array}$$

Z: 6.4

$$\begin{array}{l} \forall\, t : \mathit{EnumType} \bullet \\ \quad \exists\, f : \mathit{EnumLiteralField} \bullet f \ \underline{\mathit{DefinedBy}} \ t \end{array}$$

Z: 6.5

$$\begin{array}{l} \neg\, (\exists\, t : \mathit{EnumType};\ m : \mathit{MEMBER} \bullet \\ \quad m \ \underline{\mathit{DefinedBy}} \ t \land \\ \quad (m \in \mathit{Method} \lor m \in \mathit{Event} \lor m \in \mathit{Property}) \\ ) \end{array}$$

Z: 6.6

$$\neg\,(\exists\, t : EnumType;\ m : Field \bullet \\ \quad m\ \underline{DefinedBy}\ t \land \\ \quad m \notin EnumValueField \land \\ \quad m \notin EnumLiteralField \\ )$$

Z: 6.7

$$\forall\, t : EnumType;\ v : EnumValueField;\ l : EnumLiteralField\ | \\ \quad v\ \underline{DefinedBy}\ t \land l\ \underline{DefinedBy}\ t \bullet \\ \quad HasFieldSignature(v) = HasFieldSignature(l)$$

In our model, unlike ECMA I 7.5.2, an enum type is not an alternate name for an existing type (Appendix D (I 7.5.2)): an enum type is an element of *NamedType*, distinct from every other element. Each enum type has: one instance field (ECMA I 7.5.2, II 13.3); at least one literal static field (ECMA II 21.34 item 33); no other fields (ECMA II 13.3, II 21.34); no methods (ECMA I 7.5.2, II 13.3, II 21.34); no events (ECMA I 7.5.2, II 13.3, II 21.34); no properties (ECMA I 7.5.2, II 13.3, II 21.34); no implemented interfaces (which is already implied by "no methods") (ECMA I 7.5.2, II 21.34).

ECMA II 21.34 item 33 requires an enum type to have *at least one instance field, of integral type*, unless the enum type is CLS compliant, in which case it has *exactly one instance field.* In spite of ECMA II 21.34 item 33, each enum type in our model, including each non-compliant enum type, has one instance field.

We constrain an enum value field to be an integer type (ECMA I 7.5.2, II 21.34 item 33), in spite of ECMA II 13.3 (Appendix D (II 13.3)). Each enum literal field has the same field signature as the enum value field defined by the same enum type (ECMA II 13.3).

6.2 Type Building

Z: 6.8

$$DirectlyBuilds : Type \leftrightarrow Type$$

$$\forall\, t_1, t_2 : Type \bullet t_1 \mapsto t_2 \in DirectlyBuilds \Leftrightarrow \\ \quad (t_1 \mapsto t_2 \in ManPtr \lor \\ \quad\quad t_1 \mapsto t_2 \in UnmanPtr \lor \\ \quad\quad (\exists\, m : Method \bullet \\ \quad\quad\quad m \mapsto t_2 \in FnPtr \land \\ \quad\quad\quad HasMethodSignature(m)\ \underline{ContainsType}\ t_1) \\ \quad)$$

Z: 6.9
$Builds == DirectlyBuilds^+$

Some types directly build other types. A type t_1 directly builds a type t_2 if and only if t_1 is related to t_2 through one of the type builders: *ManPrt*, *UnmanPtr*, or *FnPtr*. *Builds* is the transitive closure of *DirectlyBuilds*: if t_1 directly builds t_2, and t_2 directly builds t_3, then t_1 builds t_3. ECMA does not contain any constraint that would prohibit cycles in the *Builds* graph: for example, a method returning a function pointer type to itself induces a cycle. There may exist types that are not built ultimately from elementary types: for example, a function pointer type may be built from a method returning a function pointer type to itself.

6.3 Uniqueness of Member Names

Z: 6.10

$$
\begin{array}{l}
\forall m_1, m_2 : Method \mid \\
\quad (m_1 \neq m_2 \wedge \\
\quad DefinedBy(m_1) = DefinedBy(m_2) \wedge \\
\quad HasMethodSignature(m_1) = HasMethodSignature(m_2)) \bullet \\
\quad HasMethodName(m_1) \neq HasMethodName(m_2)
\end{array}
$$

Z: 6.11

$$
\begin{array}{l}
\forall f_1, f_2 : Field \mid \\
\quad (f_1 \neq f_2 \wedge \\
\quad DefinedBy(f_1) = DefinedBy(f_2) \wedge \\
\quad HasFieldSignature(f_1) = HasFieldSignature(f_2)) \bullet \\
\quad HasFieldName(f_1) \neq HasFieldName(f_2)
\end{array}
$$

Z: 6.12

$$
\begin{array}{l}
\forall e_1, e_2 : Event \mid \\
\quad (e_1 \neq e_2 \wedge \\
\quad DefinedBy(e_1) = DefinedBy(e_2) \wedge \\
\quad HasEventSignature(e_1) = HasEventSignature(e_2)) \bullet \\
\quad HasEventName(e_1) \neq HasEventName(e_2)
\end{array}
$$

Z: 6.13

$$
\begin{array}{l}
\forall p_1, p_2 : Property \mid \\
\quad (p_1 \neq p_2 \wedge \\
\quad DefinedBy(p_1) = DefinedBy(p_2) \wedge \\
\quad HasPropertySignature(p_1) = HasPropertySignature(p_2)) \bullet \\
\quad HasPropertyName(p_1) \neq HasPropertyName(p_2)
\end{array}
$$

Distinct methods defined by the same named type and having the same method signature have distinct method names (ECMA I 7.5.2, Figure 6.1). So do fields, events, and properties. CLS rules impose more stringent restrictions on the uniqueness of member names (Rules 4, 5, 6, 28, 33).

6.4 Member Hiding

Z: 6.14

$$
\begin{array}{l}
\mathit{Hides} : \mathit{MEMBER} \leftrightarrow \mathit{MEMBER} \\
\hline
\forall\, m_1, m_2 : \mathit{MEMBER} \bullet m_1 \mapsto m_2 \in \mathit{Hides} \Leftrightarrow \\
\quad (\\
\quad\quad \{m_1, m_2\} \subset \mathit{Method} \lor \\
\quad\quad \{m_1, m_2\} \subset \mathit{Field} \lor \\
\quad\quad \{m_1, m_2\} \subset \mathit{Event} \lor \\
\quad\quad \{m_1, m_2\} \subset \mathit{Property} \\
\quad) \land \\
\quad (m_1 \notin \mathit{VirtualMethod}) \land \\
\quad (\mathit{DefinedBy}(m_1)\ \underline{\mathit{Inherits}}\ \mathit{DefinedBy}(m_2)) \land \\
\quad (\mathit{HasMemberName}(m_1) = \mathit{HasMemberName}(m_2)) \land \\
\quad (\\
\quad\quad \neg\, (m_1\ \underline{\mathit{HasMethodAttribute}} \\
\quad\quad\quad \mathit{HideBySignatureMethodAttribute}) \lor \\
\quad\quad \mathit{HasMemberSignature}(m_1) = \\
\quad\quad\quad \mathit{HasMemberSignature}(m_2) \\
\quad)
\end{array}
$$

A member m_1 of a derived named type hides a member m_2 of one of its base named types if and only if both members are the same kind, and both members have the same member name, and either m_1 does not have *HideBySignatureMethodAttribute* or both members have the same member signature (ECMA I 7.10.4, II 8.3). If m_1 is a method, it cannot have *VirtualMethodAttribute* (ECMA I 7.10.2).

A particular member may be hidden by multiple members; and a particular member may hide multiple members. For example, consider named types t_1, t_2, t_3, where t_2 directly inherits t_1, and t_3 directly inherits t_2; and t_1 defines a method m_1, and t_2 defines a method m_2, and t_3 defines a method m_3; and m_1, m_2, and m_3 have the same name and signature. Then m_1 is hidden by m_2 and m_3; and m_3 hides m_1 and m_2.

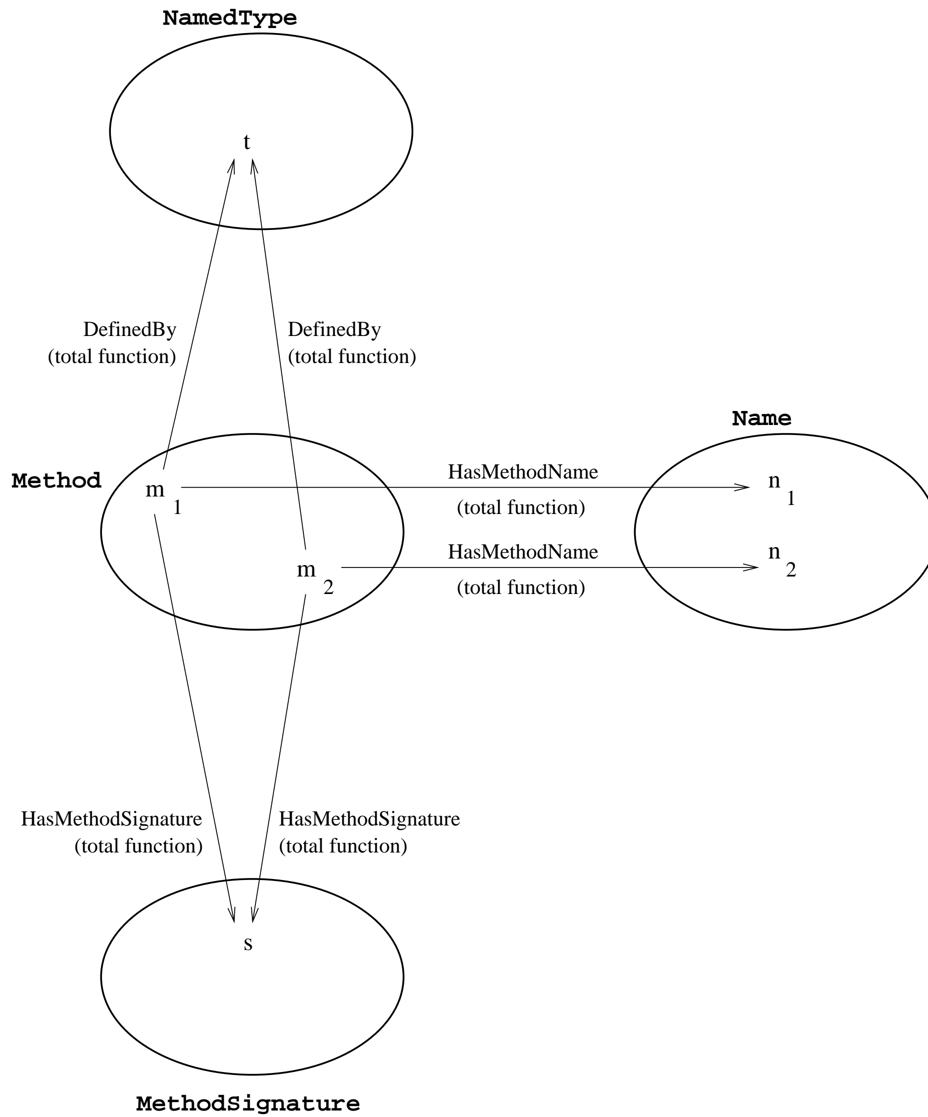

Figure 6.1: Uniqueness Of Method Names

6.5 Implements Method

Z: 6.15

$$
\begin{array}{l}
\mathit{ImplementsMethod} : \mathit{Method} \rightarrowtail \mathit{Method} \\
\hline
\forall m_1, m_2 : \mathit{Method} \bullet \\
\quad m_1 \mapsto m_2 \in \mathit{ImplementsMethod} \Leftrightarrow \\
\quad\quad (\mathit{HasMethodName}(m_1) = \\
\quad\quad\quad \mathit{HasMethodName}(m_2) \wedge \\
\quad\quad \mathit{HasMethodName}(m_1) \neq \\
\quad\quad\quad \mathit{InstanceConstructorName} \wedge \\
\quad\quad \mathit{HasMethodSignature}(m_1) = \\
\quad\quad\quad \mathit{HasMethodSignature}(m_2) \wedge \\
\quad\quad \mathit{DefinedBy}(m_1) \ \underline{\mathit{Inherits}} \ \mathit{DefinedBy}(m_2) \wedge \\
\quad\quad m_1 \in \mathit{NonAbstractMethod} \wedge \\
\quad\quad m_2 \in \mathit{AbstractMethod} \wedge \\
\quad\quad \neg \ (\\
\quad\quad\quad \exists m_3 : \mathit{Method} \bullet \\
\quad\quad\quad\quad (\mathit{HasMethodName}(m_3) = \\
\quad\quad\quad\quad\quad \mathit{HasMethodName}(m_1) \wedge \\
\quad\quad\quad\quad \mathit{HasMethodSignature}(m_3) = \\
\quad\quad\quad\quad\quad \mathit{HasMethodSignature}(m_1) \wedge \\
\quad\quad\quad\quad \mathit{DefinedBy}(m_1) \ \underline{\mathit{Inherits}} \ \mathit{DefinedBy}(m_3) \wedge \\
\quad\quad\quad\quad \mathit{DefinedBy}(m_3) \ \underline{\mathit{Inherits}} \ \mathit{DefinedBy}(m_2) \\
\quad\quad\quad\quad) \\
\quad\quad\quad) \\
\quad\quad)
\end{array}
$$

Theorem 6.1
$\forall i : \mathit{ImplementsMethod} \bullet \mathit{second}(i) \in \mathit{VirtualMethod}$

A method m_1 implements a method m_2 if and only if both methods have the same name, and neither method is an instance constructor, and both methods have the same signature, and the named type that m_1 is defined by inherits the named type that m_2 is defined by, and m_1 does not have an abstract method attribute, and m_2 has an abstract method attribute, and there is no method m_3 between m_1 and m_2 in the inheritance graph satisfying the same conditions. Each method implements at most one method.

Neither our model nor ECMA constrains each abstract method to be implemented. (In the real world, there are abstract methods that are not implemented. In our model, since *Method* contains all methods in the universe of discourse, it may not be sensible to have an abstract method that

is not implemented. Nevertheless, there is no compelling reason to include such a constraint, and we omit it in accordance with our principle of minimizing the number of constraints (Section 1.6).)

We have declared *ImplementsMethod* to be a partial function, and we assert that it is a defined constant, but we have not proven that the associated predicate constitutes a definition of a partial function.

6.6 Member Overriding

Z: 6.16

$OverridesMethod : Method \pfun Method$

$$
\begin{array}{l}
\forall m_1, m_2 : Method \bullet \\
\quad m_1 \mapsto m_2 \in OverridesMethod \Leftrightarrow \\
\quad (\\
\quad\quad HasMethodName(m_1) = \\
\quad\quad\quad HasMethodName(m_2) \land \\
\quad\quad HasMethodName(m_1) \neq \\
\quad\quad\quad InstanceConstructorName \land \\
\quad\quad HasMemberSignature(m_1) = \\
\quad\quad\quad HasMemberSignature(m_2) \land \\
\quad\quad DefinedBy(m_1) \ \underline{Inherits} \ DefinedBy(m_2) \land \\
\quad\quad m_1 \in VirtualMethod \land \\
\quad\quad m_1 \in NonAbstractMethod \land \\
\quad\quad m_2 \in VirtualMethod \land \\
\quad\quad m_2 \in NonAbstractMethod \land \\
\quad\quad \neg (\exists m_3 : Method \bullet \\
\quad\quad\quad HasMethodName(m_3) = \\
\quad\quad\quad\quad HasMethodName(m_1) \land \\
\quad\quad\quad HasMethodSignature(m_3) = \\
\quad\quad\quad\quad HasMethodSignature(m_1) \land \\
\quad\quad\quad DefinedBy(m_1) \ \underline{Inherits} \ DefinedBy(m_3) \land \\
\quad\quad\quad DefinedBy(m_3) \ \underline{Inherits} \ DefinedBy(m_2) \\
\quad\quad) \\
\quad)
\end{array}
$$

Z: 6.17

$$
\begin{array}{l}
\mathit{OverridesField} : \mathit{Field} \pfun \mathit{Field} \\
\hline
\forall m_1, m_2 : \mathit{Field} \bullet \\
\quad m_1 \mapsto m_2 \in \mathit{OverridesField} \Leftrightarrow \\
\quad (\\
\qquad \mathit{HasFieldName}(m_1) = \mathit{HasFieldName}(m_2) \land \\
\qquad \mathit{HasFieldSignature}(m_1) = \mathit{HasFieldSignature}(m_2) \land \\
\qquad \mathit{DefinedBy}(m_1)\ \underline{\mathit{Inherits}}\ \mathit{DefinedBy}(m_2) \land \\
\qquad m_1 \notin \mathit{StaticField} \land \\
\qquad m_2 \notin \mathit{StaticField} \land \\
\qquad \neg\ (\exists\, m_3 : \mathit{Field} \bullet \\
\qquad\quad \mathit{HasFieldName}(m_3) = \\
\qquad\qquad \mathit{HasFieldName}(m_1) \land \\
\qquad\quad \mathit{HasFieldSignature}(m_3) = \\
\qquad\qquad \mathit{HasFieldSignature}(m_1) \land \\
\qquad\quad \mathit{DefinedBy}(m_1)\ \underline{\mathit{Inherits}}\ \mathit{DefinedBy}(m_3) \land \\
\qquad\quad \mathit{DefinedBy}(m_3)\ \underline{\mathit{Inherits}}\ \mathit{DefinedBy}(m_2) \\
\qquad) \\
\quad)
\end{array}
$$

Z: 6.18

$$\mathit{OverridesMember} == \mathit{OverridesMethod} \cup \mathit{OverridesField}$$

A method m_1 overrides a method m_2 if and only if both methods have the same name, and neither method is an instance constructor, and both methods have the same signature, and the named type that m_1 is defined by inherits the named type that m_2 is defined by, and both methods have a virtual method attribute, and neither method has an abstract method attribute, and there is no method m_3 between m_1 and m_2 in the inheritance graph satisfying the same conditions (ECMA I 7.10.4).

A field m_1 overrides a field m_2 if and only if both fields have the same name, and both fields have the same signature, and the named type that m_1 is defined by inherits the named type that m_2 is defined by, and neither field has a static field attribute, and there is no field m_3 between m_1 and m_2 in the inheritance graph satisfying the same conditions (ECMA I 7.10.4).

We have declared *OverridesMethod* and *OverridesField* to be partial functions, and we assert that they are defined constants, but we have not proven that the associated predicates constitute definitions of partial functions.

6.7 Implementation of Abstract Methods

Z: 6.19

$$\forall\, t : \mathit{NonAbstractNamedType};\; m_1 : \mathit{AbstractMethod} \mid$$
$$t \;\underline{\mathit{Inherits}}\; \mathit{DefinedBy}(m_1) \bullet$$
$$\exists\, m_2 : \mathit{Method} \bullet$$
$$(t = \mathit{DefinedBy}(m_2) \lor t \;\underline{\mathit{Inherits}}\; \mathit{DefinedBy}(m_2)) \land$$
$$m_2 \;\underline{\mathit{ImplementsMethod}}\; m_1$$

Each non-abstract named type t that inherits a named type that defines an abstract method m_1 either defines a method m_2 that implements m_1, or inherits a named type that defines a method m_2 that implements m_1 (ECMA I 7.9.4, I 7.9.5).

Part II

CLS Rules

Chapter 7

CLS Compliant and Non-Compliant Markings

7.1 Explicit Marking

Z: 7.1
$ClsAttribute ::= CompliantAttribute \mid NonCompliantAttribute$

Z: 7.2
$HasAssemblyClsAttribute : ASSEMBLY \pfun ClsAttribute$

Z: 7.3
$HasNamedTypeClsAttribute : NamedType \pfun ClsAttribute$

Z: 7.4
$HasMemberClsAttribute : MEMBER \pfun ClsAttribute$

Z: 7.5
$ExplicitlyCompliantAssembly ==$
$\quad \mathrm{dom}(HasAssemblyClsAttribute \rhd \{CompliantAttribute\})$

Z: 7.6
$ExplicitlyNonCompliantAssembly ==$
$\quad \mathrm{dom}(HasAssemblyClsAttribute \rhd \{NonCompliantAttribute\})$

Z: 7.7

$$UnmarkedAssembly == \\ \quad ASSEMBLY \setminus ExplicitlyCompliantAssembly \setminus \\ \quad ExplicitlyNonCompliantAssembly$$

Z: 7.8

$$ExplicitlyCompliantNamedType == \\ \quad \operatorname{dom}(HasNamedTypeClsAttribute \rhd \{CompliantAttribute\})$$

Z: 7.9

$$ExplicitlyCompliantTopLevelNamedType == \\ \quad ExplicitlyCompliantNamedType \cap TopLevelNamedType$$

Z: 7.10

$$ExplicitlyCompliantNestedNamedType == \\ \quad ExplicitlyCompliantNamedType \cap NestedNamedType$$

Z: 7.11

$$ExplicitlyNonCompliantNamedType == \\ \quad \operatorname{dom}(HasNamedTypeClsAttribute \rhd \{NonCompliantAttribute\})$$

Z: 7.12

$$ExplicitlyNonCompliantTopLevelNamedType == \\ \quad ExplicitlyNonCompliantNamedType \cap TopLevelNamedType$$

Z: 7.13

$$ExplicitlyNonCompliantNestedNamedType == \\ \quad ExplicitlyNonCompliantNamedType \cap NestedNamedType$$

Z: 7.14

$$UnmarkedNamedType == \\ \quad NamedType \setminus ExplicitlyCompliantNamedType \setminus \\ \quad ExplicitlyNonCompliantNamedType$$

Z: 7.15

$$UnmarkedTopLevelNamedType == \\ \quad UnmarkedNamedType \cap TopLevelNamedType$$

Z: 7.16

$$UnmarkedNestedNamedType == \\ \quad UnmarkedNamedType \cap NestedNamedType$$

Z: 7.17
$ExplicitlyNonCompliantMember ==$
$\quad \mathrm{dom}(HasMemberClsAttribute \rhd \{NonCompliantAttribute\})$

Z: 7.18
$UnmarkedMember ==$
$\quad MEMBER \setminus ExplicitlyNonCompliantMember$

Z: 7.19
$\mathrm{dom}(HasMemberClsAttribute \rhd \{CompliantAttribute\}) = \varnothing$

Undefined: *HasAssemblyClsAttribute*, *HasNamedTypeClsAttribute*, *HasMemberClsAttribute*

CompliantAttribute and *NonCompliantAttribute* are custom attributes (ECMA I 6.3.1, I 8.7, I 9.6). ECMA specifies a mechanism for constructing custom attributes and attaching them to entities such as assemblies, named types, and members. The custom attribute mechanism is outside the scope of our model; it is sufficient that *CompliantAttribute* and *NonCompliantAttribute* exist and can be attached to assemblies, named types, and members. An assembly or named type having a *CompliantAttribute* is said to be explicitly compliant; an assembly, named type, or member having a *NonCompliantAttribute* is said to be explicitly non-compliant; one having no attribute is said to be unmarked. No member has a *CompliantAttribute* (Appendix D (I 6.3.1)). Named types and members do not inherit a cls attribute via the inheritance graph (Appendix E.2).[1]

7.2 Compliant Assemblies

Z: 7.20
$CompliantAssembly == ExplicitlyCompliantAssembly$

Z: 7.21
$NonCompliantAssembly == ASSEMBLY \setminus CompliantAssembly$

Each explicitly compliant assembly is a compliant assembly; every other assembly is a non-compliant assembly.

[1]Thanks to Julianto, who pointed out that CLS compliant and CLS non-compliant attributes can possibly be inherited from a base Type.

7.3 Acquired Marking

Z: 7.22

$$
\begin{aligned}
&\mathit{CompliantTopLevelNamedType} == \\
&\quad \{t : \mathit{TopLevelNamedType} \mid \\
&\qquad (t \in \mathit{ExplicitlyCompliantTopLevelNamedType}) \lor \\
&\qquad (t \in \mathit{UnmarkedTopLevelNamedType} \land \\
&\qquad\quad \mathit{ContainedBy}(t) \in \mathit{CompliantAssembly})\}
\end{aligned}
$$

Z: 7.23

$$
\mathit{UnmarkedNamedTypeNestedIn} : \mathit{NamedType} \leftrightarrow \mathit{NamedType}
$$

$$
\begin{aligned}
&\forall t_1, t_2 : \mathit{NamedType} \bullet \\
&\quad t_1 \mapsto t_2 \in \mathit{UnmarkedNamedTypeNestedIn} \Leftrightarrow \\
&\quad (t_1 \underline{\mathit{DirectlyNestedIn}}\ t_2 \land t_1 \in \mathit{UnmarkedNamedType})
\end{aligned}
$$

Z: 7.24

$$
\begin{aligned}
&\mathit{CompliantNamedType} == \\
&\quad \mathrm{dom}(\mathit{UnmarkedNamedTypeNestedIn}^{*} \rhd \\
&\qquad (\mathit{CompliantTopLevelNamedType} \cup \\
&\qquad \mathit{ExplicitlyCompliantNestedNamedType}))
\end{aligned}
$$

Z: 7.25

$$
\mathit{NonCompliantNamedType} == \mathit{NamedType} \setminus \mathit{CompliantNamedType}
$$

Z: 7.26

$$
\begin{aligned}
&\mathit{CompliantMember} == \{m : \mathit{UnmarkedMember} \mid \\
&\quad \mathit{DefinedBy}(m) \in \mathit{CompliantNamedType}\}
\end{aligned}
$$

Z: 7.27

$$
\mathit{NonCompliantMember} == \mathit{MEMBER} \setminus \mathit{CompliantMember}
$$

Z: 7.28

$$
\begin{aligned}
&\mathit{NonCompliantElementaryType} == \\
&\quad (\mathrm{ran}(\mathit{NonCompliantNamedType} \lhd \mathit{Elementary})) \\
&\quad \cup \\
&\quad \mathit{BoxedValueType}
\end{aligned}
$$

Z: 7.29

$$
\begin{aligned}
&\mathit{CompliantElementaryType} == \mathit{ElementaryType} \setminus \\
&\quad \mathit{NonCompliantElementaryType}
\end{aligned}
$$

Z: 7.30

$$NonCompliantType == \\ \quad \mathrm{ran}(NonCompliantElementaryType \lhd Builds^*) \\ \quad \cup \\ \quad \mathrm{ran}(\mathrm{ran}(NonCompliantMember \lhd FnPtr) \lhd Builds^*) \\ \quad \cup \\ \quad UnmanagedPointerType$$

Z: 7.31

$$CompliantType == Type \setminus NonCompliantType$$

Unmarked named types and members acquire compliance or non-compliance from the parent that contains, nests, or defines them (ECMA I 6.3.1). Top level named types acquire from the assembly they are contained by; nested named types acquire from the named type they are directly nested in; members acquire from the named type they are defined by.

A compliant top level named type is an explicitly compliant top level named type, or an unmarked top level named type contained by a compliant assembly (ECMA I 6.3.1).

UnmarkedNamedTypeNestedIn is a partial function relating each unmarked nested named type to the named type it is directly nested in. $UnmarkedNamedTypeNestedIn^*$, the reflexive transitive closure of *Unmarked-NamedTypeNestedIn*, relates each (top level and nested) named type to itself and to zero or more other named types of which at most one is explicitly compliant or explicitly non-compliant. $UnmarkedNamedTypeNestedIn^*$ partitions *NamedType* into subsets: each subset contains zero or more unmarked nested named types and either one top level named type or one explicitly compliant nested named type or one explicitly non-compliant nested named type. A compliant named type is a (top level or nested) named type related via $UnmarkedNamedTypeNestedIn^*$ to either a compliant top level named type or an explicitly compliant nested named type (Figure 7.1).

A compliant member is an unmarked member defined by a compliant named type (ECMA I 6.3.1).

A non-compliant elementary type is a boxed value type or an elementary type related via the *Atomic* total injection to a non-compliant named type. CLS Rule 3 states that boxed value types are non-compliant; we must state this constraint outside the CLS rules, since otherwise Rule 3 would be a logical contradiction. Note that void type is a compliant elementary type.

A non-compliant type is a non-compliant elementary type, or a type built ultimately from one or more non-compliant elementary types, or a function pointer type built from a non-compliant member, or a type built ultimately from one or more non-compliant function pointer types, or an unmanaged pointer type. CLS Rule 17 states that unmanaged pointer types

are non-compliant; we must state this constraint outside the CLS rules, since otherwise Rule 17 would be a logical contradiction.

Although ECMA I 6.1 states that *CLS conformance is a characteristic of types*, ECMA does not explicitly define CLS compliance on general types, for example, on managed pointer types or function pointer types. Nevertheless, several CLS rules, for example Rule 11, require a specification of CLS compliance on a general type. Our model defines CLS compliance on any type. See Appendix E.9.1 for further discussion.

We have declared *UnmarkedNamedTypeNestedIn* to be a partial function, and we assert that it is a defined constant, but we have not proven that the associated predicate constitutes a definition of a partial function.

7.4 Compliant and Non-Compliant Named Types

Z: 7.32

$$\{C_J, C_V, C_E, C_A, C_D, C_S, C_U8, C_I16, C_I32, C_I64, C_I, C_B, C_F32, C_F64, C_C\} \subset \mathit{CompliantNamedType}$$

Z: 7.33

$$\{C_R, C_I8, C_U16, C_U32, C_U64, C_U\} \subset \mathit{NonCompliantNamedType}$$

Z: 7.34

$$\mathit{BuiltinClsValueType} == \mathit{BuiltinValueType} \cap \mathit{CompliantNamedType}$$

Z: 7.35

$$\mathit{BuiltinClsIntegerType} == \mathit{BuiltinClsValueType} \cap \mathit{IntegerType}$$

We state the compliance status of each system named type, which is specified in the ECMA Library, except C_R, which does not appear to be defined in the ECMA Library. Rule 14 states that C_R is non-compliant.

C_J is a compliant named type (ECMA Library and ECMA I 7.9.9). Rule 23 reiterates that *System.Object is CLS-compliant.* We state this constraint outside the CLS rules; if the constraint were stated only as a CLS rule, then an implementation defining C_J as a non-compliant named type would conform to our model, since the preamble would inhibit the application of the rule to C_J.

ECMA does not specify whether each global named type is compliant. In accordance with our policy of minimizing the number of constraints (Section 1.6), we do not constrain each global named type to be compliant. The global named type in some assemblies may contain non-compliant members, for example, static fields and methods (Rule 36).

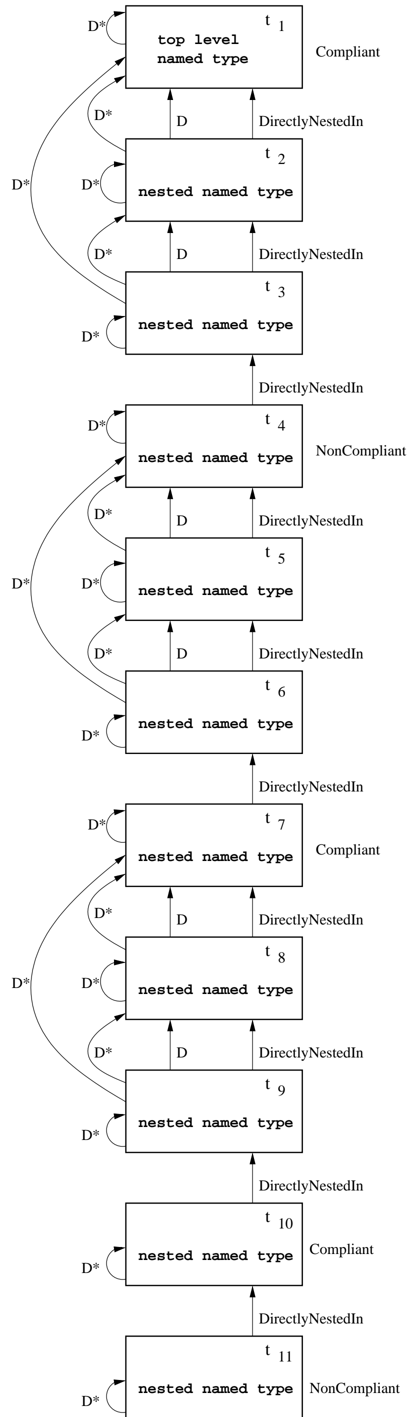

Figure 7.1: Compliant Nested Named Types

Chapter 8

Exposure

Referencibility of an entity is determined by: the relationship between the referencing entity and the referenced entity; and the visibility and accessibility assigned to the referenced entity. ECMA specifies the concept of *security*, which also determines the referencibility. Security is outside the scope of our model. We model the assignment of visibility and accessibility to entities but, except for the concept of *exposure*, we do not model the referencibility conditions, that is the conditions under which a reference is permitted.

Exposure is a subset of the referencibility conditions; exposure is applicable when the referenced entity and the referencing entity are in different assemblies. Exposure determines whether a type can be referenced outside the assembly it is contained by, or a member can be referenced outside the assembly containing the type it is defined by. Although ECMA makes extensive use of the concept of exposure, especially in the CLS rules, it does not explicitly define the term. The exposure of a type or member is determined by a combination of visibility and accessibility.

According to our interpretation of ECMA, a member or nested named type may have greater accessibility than the accessibility of the nested named type it is defined by or nested in (its parent). However, this does not necessarily mean it is more "referencible" than its parent, because the referencibility conditions, which except for the exposure subset we do not model, may specify that it is only referencible when its parent is referencible. Our model defines exposure so that each entity is exposed only when its parent is exposed. (Appendix E.3)

8.1 Visibility

Z: 8.1
Visibility ::= *AssemblyVisibility* | *PublicVisibility*

Z: 8.2

$$VisibilityGreater : Visibility \leftrightarrow Visibility$$

$$VisibilityGreater = \{PublicVisibility \mapsto AssemblyVisibility\}$$

Z: 8.3

$$HasExplicitVisibility : TopLevelNamedType \rightarrow Visibility$$

Z: 8.4

$$HasVisibility : NamedType \rightarrow Visibility$$

$$HasVisibility = HasExplicitVisibility \cup ((NestedIn \rhd TopLevelNamedType) \fatsemi HasExplicitVisibility)$$

Z: 8.5

$$HasElementaryTypeVisibility : ElementaryType \rightarrow Visibility$$

$$HasElementaryTypeVisibility = (Elementary^{\sim} \fatsemi HasVisibility) \cup \{Void \mapsto PublicVisibility\}$$

Z: 8.6

$$HasVis : Type \leftrightarrow Visibility$$

$$HasTypeVisibility : Type \rightarrow Visibility$$

$$HasVis = (Builds^{*})^{\sim} \fatsemi HasElementaryTypeVisibility$$

$$\forall t : Type; \ v : Visibility \bullet t \mapsto v \in HasTypeVisibility \Leftrightarrow t \mapsto v \in HasVis \land \neg (\exists v_1 : Visibility \bullet t \mapsto v_1 \in HasVis \land v \ \underline{VisibilityGreater} \ v_1)$$

Undefined: *HasExplicitVisibility*

A visibility is either assembly visibility or public visibility. *VisibilityGreater* is a total order on *Visibility* such that public visibility is greater than assembly visibility.

Each top level named type has one explicit visibility: either assembly visibility or public visibility (ECMA I 7.5.3.1, I 7.9.5, I 7.9.6.1, II 8.1). The explicit visibility of top level named types is given; it is determined by language syntax and the assembly configuration. According to ECMA I 7.5.3.1 *while a type may be marked to allow it to be exported from the assembly, it is the configuration of the assembly that decides whether the type name is made available.* The given values of *HasExplicitVisibility* are determined after the assembly configuration has made its decision. Language syntax may allow the visibility specifier of a top level named type to be omitted. In this case, the visibility defaults according to the language rules, possibly modified by the assembly configuration, but the top level named type is still said to have an explicit visibility.

Each named type has a visibility: the visibility of a top level named type is the same as its explicit visibility; the visibility of a nested named type is the same as the explicit visibility of the top level named type it is (ultimately) nested in (ECMA I 7.5.3.1, I 7.5.3.4, I 7.11.5, II 8.1).

ECMA does not explicitly define visibility on general types, for example, on managed pointer types or function pointer types. Nevertheless, several CLS rules, for example Rule 12, require a specification of visibility on any type. Our model defines visibility on each type. See Appendix E.9.2 for further discussion.

Each elementary type has the same visibility as the named type it is related to via the *Elementary* total injection; void type has public visibility; each non-elementary type has the same visibility as the "minimum" visibility of the elementary types it is ultimately built from.

We have declared *HasVisibility*, *HasElementaryTypeVisibility*, and *HasTypeVisibility* to be total functions, and we assert that they are defined constants, but we have not proven that the associated predicates constitute definitions of total functions.

8.2 Accessibility

Z: 8.7

Accessibility ::=
  CompilerControlledAccessibility |
  PublicAccessibility |
  FamilyOrAssemblyAccessibility |
  AssemblyAccessibility |
  FamilyAccessibility |
  FamilyAndAssemblyAccessibility |
  PrivateAccessibility

Z: 8.8

$$AGE : Accessibility \leftrightarrow Accessibility$$

$$AccessibilityGreaterOrEqual : Accessibility \leftrightarrow Accessibility$$

$$\begin{aligned} AGE = \{ & PublicAccessibility \mapsto \\ & \quad FamilyOrAssemblyAccessibility, \\ & FamilyOrAssemblyAccessibility \mapsto AssemblyAccessibility, \\ & FamilyOrAssemblyAccessibility \mapsto FamilyAccessibility, \\ & AssemblyAccessibility \mapsto \\ & \quad FamilyAndAssemblyAccessibility, \\ & FamilyAccessibility \mapsto FamilyAndAssemblyAccessibility, \\ & FamilyAndAssemblyAccessibility \mapsto PrivateAccessibility\} \end{aligned}$$

$$AccessibilityGreaterOrEqual = AGE^*$$

Z: 8.9

$$HasMemberAccessibility : MEMBER \rightarrow Accessibility$$

Z: 8.10

$$\mathrm{ran}(Interface \lhd (DefinedBy^\sim \fatsemi HasMemberAccessibility)) = \\ \{PublicAccessibility\}$$

Z: 8.11

$$HasNestedNamedTypeAccessibility : NestedNamedType \rightarrow \\ Accessibility$$

Z: 8.12

$$HasElementaryTypeAccessibility : Type \nrightarrow Accessibility$$

$$HasElementaryTypeAccessibility = \\ (Elementary^\sim \fatsemi HasNestedNamedTypeAccessibility)$$

Z: 8.13

$$HasAcc : Type \leftrightarrow Accessibility$$

$$HasTypeAccessibility : Type \leftrightarrow Accessibility$$

$$HasAcc = (Builds^*)^\sim \fatsemi HasElementaryTypeAccessibility$$

$$\begin{aligned} & \forall t : Type;\ a : Accessibility \bullet t \mapsto a \in HasTypeAccessibility \Leftrightarrow \\ & \quad t \mapsto a \in HasAcc \land \\ & \quad \neg (\exists a_1 : Accessibility \bullet \\ & \qquad t \mapsto a_1 \in HasAcc \land \\ & \qquad\quad a \ \underline{AccessibilityGreaterOrEqual}\ a_1) \end{aligned}$$

Undefined: *HasMemberAccessibility*, *HasNestedNamedTypeAccess−ibility*

An accessibility is a compiler-controlled-accessibility, or a public-accessibility, or a family-or-assembly-accessibility, or an assembly-accessibility, or a family-accessibility, or a family-and-assembly-accessibility, or a private-accessibility. *AccessibilityGreaterOrEqual* is a partial order on *Accessibility*. Note that no ordering is defined between compiler-controlled-accessibility and any other accessibility; nor between assembly-accessibility and family-accessibility.

Each member has an accessibility (ECMA I 7.5.3.2). Each nested named type has an accessibility (ECMA I 7.5.3.4). The accessibility of members and nested named types is given; it is determined by language syntax. See Appendix E.3 for a discussion of the accessibility of members defined by a nested named type, and nested named types nested in a nested named type. Each member defined by an interface has public accessibility (ECMA I 7.5.3.2, I 7.9.4).

ECMA does not explicitly define accessibility on general types, for example, on managed pointer types or function pointer types. Nevertheless, several CLS rules, for example Rule 12, require a specification of accessibility on some types other than elementary types. Our model defines accessibility on each elementary type related via the *Elementary* total injection to a nested named type, and on each type built ultimately from one or more such elementary types. See Appendix E.9.3 for further discussion.

Each elementary type has the same accessibility as the nested named type it is related to via the *Elementary* total injection, if any; if it is not related to a nested named type, then it does not have an accessibility; accessibility is not defined on void type; each non-elementary type has the same accessibility as the "minimum" accessibility of the elementary types it is ultimately built from, if any; if none of the elementary types it is built from has an accessibility, then it also does not have an accessibility; if some of the elementary types it is built from do not have an accessibility, then those elementary types do not participate in the determination of the "minimum" accessibilty of the remaining elementary types. Note that *HasTypeAccessibility* is not a function, because a non-elementary type built ultimately from multiple elementary types may have more than one accessibility; this occurs, for example, when one elementary type has assembly accessibility and the other has family accessibility; since these accessibilities are unrelated by the *AccessibilityGreaterOrEqual* partial order, neither is greater than or equal to the other and both qualify as "minimum".

We have declared *HasElementaryTypeAccessibility* to be a partial function, and we assert that it is a defined constant, but we have not proven that the associated predicate constitutes a definition of a partial function.

8.3 Type and Member Exposure

Z: 8.14

$$ExposedTopLevelNamedType == \{t : TopLevelNamedType \mid t \ \underline{HasVisibility} \ PublicVisibility\}$$

Z: 8.15

$$ExposedNestedNamedType == \{t : NestedNamedType \mid t \ \underline{HasVisibility} \ PublicVisibility \land (\exists a : Accessibility \bullet t \ \underline{HasNestedNamedTypeAccessibility} \ a \land a \ \underline{AccessibilityGreaterOrEqual} \ FamilyAccessibility)\}$$

Z: 8.16

$$ExposedNamedType == ExposedTopLevelNamedType \cup ExposedNestedNamedType$$

Z: 8.17

$$NonExposedType == \{t : Type \mid t \ \underline{HasTypeVisibility} \ AssemblyVisibility \lor (\exists a : Accessibility \bullet t \ \underline{HasTypeAccessibility} \ a \land AssemblyAccessibility \ \underline{AccessibilityGreaterOrEqual} \ a)\}$$

Z: 8.18

$$ExposedType == Type \setminus NonExposedType$$

Z: 8.19

$$ExposedMember == \{m : MEMBER \mid DefinedBy(m) \in ExposedNamedType \land (\exists a : Accessibility \bullet m \ \underline{HasMemberAccessibility} \ a \land a \ \underline{AccessibilityGreaterOrEqual} \ FamilyAccessibility)\}$$

Z: 8.20

$$ExposedMethod == ExposedMember \cap Method$$

Z: 8.21

$$ExposedField == ExposedMember \cap Field$$

Z: 8.22

$$ExposedEvent == ExposedMember \cap Event$$

Z: 8.23
$ExposedProperty == ExposedMember \cap Property$

Z: 8.24
$ExposedCompliantNamedType ==$
$\quad ExposedNamedType \cap CompliantNamedType$

Z: 8.25
$ExposedCompliantType ==$
$\quad ExposedType \cap CompliantType$

Z: 8.26
$ExposedCompliantMember ==$
$\quad ExposedMember \cap CompliantMember$

Z: 8.27
$SystemNamedType \subset ExposedTopLevelNamedType$

An exposed top level named type is a top level named type that has public visibility. An exposed nested named type is a nested named type that has public visibility and at least family accessibility. A non-exposed type is a type that has assembly visibility or has an accessibility less than or equal to assembly accessibility. Note that some types do not have an accessibility and some types have more than one accessibility (Section 8.2). Note that void type is an exposed type. An exposed member is one that has at least family accessibility and is defined by an exposed named type. Each system named type is an exposed top level named type (ECMA Library).

Chapter 9

CLS Rules

The CLS rules are collected in ECMA I 10. Our model does not admit the possibility that a CLS rule may be broken. It is a logical contradiction for an element of *ExposedCompliantNamedType*, *ExposedCompliantType*, or *ExposedCompliantMember* to break a CLS rule. See Appendix B.5 for further discussion.

Preamble

> *These rules apply only to "externally visible" items – types that are visible outside of their own assembly and members of those types that have public, family, or family-or-assembly accessibility. Furthermore, items may be explicitly marked as CLS-compliant or not using the System.CLSCompliantAttribute. The CLS rules apply only to items that are marked as CLS-compliant.*

We interpret this to mean: the CLS rules apply only to exposed items that are marked as CLS-compliant or acquire CLS-compliance as described in Section 7.3. That is, these rules apply only to the sets *ExposedCompliantNamedType*, *ExposedCompliantType*, and *ExposedCompliantMember*. See Appendix E.4 for further discussion.

Rule 1

> *CLS rules apply only to those parts of a type that are accessible or visible outside of the defining assembly.*

Rule 1 repeats part of the Preamble. There is nothing to state.

Rule 2

Members of non-CLS compliant types shall not be marked CLS-compliant.

Z: 9.1
$$Cls02 == \{m : ExposedCompliantMember \mid DefinedBy(m) \in NonCompliantNamedType\}$$

Theorem 9.1
$Cls02 = \varnothing$

Rule 2 cannot be "broken".[1] The definition of *CompliantMember* requires that the named type that every compliant member is defined by should also be compliant (Section 7.3). This is consistent with ECMA I 6.3.1, which, according to our interpretation (Appendix D (I 6.3.1)), does not permit a member (of either a compliant or non-compliant named type) to be explicitly marked compliant. If source code contains a member explicitly marked compliant, then the source code is in error; this is not a CLS error, as the requirement specified by ECMA I 6.3.1 precedes the CLS rule. If a compiler generates a named type from such source code, then the named type does not conform to our model; and any assembly containing the named type does not conform.

Rule 3

The CLS does not include boxed value types.

Z: 9.2
$$Cls03 == ExposedCompliantType \cap BoxedValueType$$

Theorem 9.2
$Cls03 = \varnothing$

A value type may be an explicitly compliant named type, or an explicitly non-compliant named type (Section 7.1), or it may acquire compliance or non-compliance from the assembly it is contained by or from the named type it is nested in (Section 7.3). Each value type is associated with one boxed value type via the *Atomic* total injection, and one unboxed value type via the *Ubvt* total injection (Section 4.1, Figure 4.3). Rule 3 states that each

[1] Thanks to Adnan Bader, who pointed out that CLS Rule 2 is a tautology.

boxed value type is non-compliant, regardless of the compliance of the value type it is related to.

Rule 3 cannnot be "broken". In the definition of *NonCompliantElementaryType* (Section 7.3), we constrained each boxed value type to be a non-compliant elementary type. Therefore, we need not state $Cls03 = \varnothing$ as an axiom, since it is provable as a theorem. If we had not constrained each boxed value type to be a non-compliant elementary type, and we had stated $Cls03 = \varnothing$ as an axiom, then our model would not be satisfiable, as Rule 3 would be a logical contradiction; for example, if v were a value type such that v is an exposed compliant named type, then $Atomic(v)$ would be both an exposed compliant type and a boxed value type; but Rule 3 states that no type can be both an exposed compliant type and a boxed value type.

Rule 4

> *Assemblies shall follow Annex 7 of Technical Report 15 of the Unicode Standard 3.0 (ISBN 0-201-61633-5) governing the set of characters permitted to start and be included in identifiers. For CLS purposes, two identifiers are the same if their lowercase mappings (as specified by the Unicode locale-insensitive, 1-1 lowercase mappings) are the same. That is, for two identifiers to be considered different under the CLS they shall differ in more than simply their case. However, in order to override an inherited definition the CLI requires the precise encoding of the original declaration be used.*

Our model does not specify *UNICODE* in sufficient detail to permit a formal statement of Rule 4.

Rule 5

> *All names introduced in a CLS-compliant scope shall be distinct independent of kind, except where the names are identical and resolved via overloading*

For the purposes of Rule 5: *name* means *name of an exposed compliant member*; *scope* means *named type*; *kind* means *method, field, event, property, or nested named type*; *distinct* and *identical* mean *distinct and identical when compared according to Rule 4*. (ECMA I 7.5.2).

We justify this interpretation as follows. Immediately prior to the introduction of Rule 5, ECMA I 7.5.2 explicitly declares *kind* to mean *method, field, event, property, or nested named type*. The same sentence also refers to *signature*. Since *kind* (with the meaning just defined) and *signature* cannot

relate to *assembly*, in this context *scope* does not include *assembly*. *Name* refers to the names only of compliant members; otherwise, frameworks could not achieve compliance by *marking sufficient offending members within the type as not CLS-compliant so that the remaining members do not conflict with one another*. We assume that *names* must also refer only to the names of exposed members.

Our model does not specify *UNICODE* in sufficient detail to permit *distinct* and *identical* names to be identified in accordance with Rule 4.

Rule 6

> *Fields and nested types shall be distinct by identifier comparison alone, even though the CTS allows distinct signatures to be distinguished. Methods, properties, and events that have the same name (by identifier comparison) shall differ by more than just the return type, except as specified in CLS Rule 39.*

Our model does not specify *UNICODE* in sufficient detail to permit *distinct* names to be identified in accordance with Rule 4.

Rule 7

> *The underlying type of an enum shall be a built-in CLS integer type.*

Z: 9.3

$$
\begin{aligned}
&\mathit{Cls}07 == \{t : \mathit{ExposedCompliantNamedType} \mid \\
&\quad (\exists f : \mathit{EnumValueField} \bullet \\
&\qquad \mathit{DefinedBy}(f) = t \land \\
&\qquad \mathit{Atomic}^{\sim}(\mathit{FieldType}(\mathit{HasFieldSignature}(f))) \notin \\
&\qquad\quad \mathit{BuiltinClsIntegerType} \setminus \{C_I\})\} \\
&\mathit{Cls}07 = \varnothing
\end{aligned}
$$

See Appendix D (II 13.3) for a discussion of the allowable underlying type of an enum.

We explicitly exclude C_I as an allowable named type for the enum value field of a compliant enum type. Although C_I is a builtin cls integer type (Section 7.4, ECMA Library "System.IntPtr", and ECMA I 7.2.2), and Rule 7 (ECMA I 7.5.2) states that *the underlying type of an enum shall be a built-in CLS integer type*, ECMA II 21.34 item 33 does not include C_I. We assume ECMA II 21.34 item 33 takes precedence.

Rule 8

> *There are two distinct kinds of enums, indicated by the presence or absence of the System.FlagsAttribute custom attribute. One represents named integer values, the other named bit flags that can be combined to generate an unnamed value. The value of an enum is not limited to the specified values.*

We expect that a CLS rule will specify some restriction applying to CLS compliant assemblies, types, or members. This rule has two parts: an explanation that there are two kinds of enums, depending on FlagsAttribute; and a statement that the value of an enum is not limited to the values of its associated literals. Both parts seem to apply equally to both compliant and non-compliant items. This does not seem to be a CLS rule. There is nothing to state.

Rule 9

> *Literal static fields of an enum shall have the type of the enum itself.*

Z: 9.4

$$
\begin{aligned}
&\mathit{Cls09} == \{t : \mathit{ExposedCompliantNamedType} \mid \\
&\quad (\exists f_1 : \mathit{EnumValueField};\ f_2 : \mathit{EnumLiteralField} \bullet \\
&\qquad \mathit{DefinedBy}(f_1) = t \land \\
&\qquad \mathit{DefinedBy}(f_2) = t \land \\
&\qquad \mathit{HasFieldSignature}(f_1) \neq \mathit{HasFieldSignature}(f_2))\}
\end{aligned}
$$

Theorem 9.3
$\mathit{Cls09} = \varnothing$

This rule, if taken literally, is self-contradictory, regardless of how we model the type system: an enum type has a value field and at least one literal static field; if the literal static field has the type of the enum, then it too has a value field and at least one literal static field; and so on, *ad infinitum.*

We adopt the common sense interpretation: *literal static fields of an enum shall have the type of the value field of the enum.* Alternatively, since ECMA uses the phrase *underlying type* to mean *the type of the value field*: *literal static fields of an enum shall have the underlying type of the enum.*

Rule 9 cannot be "broken", because ECMA II 13.3, outside the CLS rules, specifies that *the static literal fields of an enum ... all of these fields shall have the type of the enum ....*

Rule 10

Accessibility shall not be changed when overriding inherited methods, except when overriding a method inherited from a different assembly with accessibility Family-or-Assembly. In this case the override shall have accessibility family.

Z: 9.5

$$
\begin{array}{l}
\mathit{Cls}10 == \{m_1 : \mathit{ExposedCompliantMember} \mid \\
\quad \exists\, m_2 : \mathit{MEMBER};\ a_1, a_2 : \mathit{ASSEMBLY};\\
\quad p_1, p_2 : \mathit{Accessibility} \bullet \\
\qquad (\mathit{ContainedBy}(\mathit{DefinedBy}(m_1)) = a_1 \land \\
\qquad \mathit{ContainedBy}(\mathit{DefinedBy}(m_2)) = a_2 \land \\
\qquad m_2 \mapsto m_1 \in \mathit{OverridesMember} \land \\
\qquad \mathit{HasMemberAccessibility}(m_1) = p_1 \land \\
\qquad \mathit{HasMemberAccessibility}(m_2) = p_2 \land \\
\qquad (\\
\qquad\quad (a_1 = a_2 \land p_1 \neq p_2) \lor \\
\qquad\quad (a_1 \neq a_2 \land p_1 \neq p_2 \land \\
\qquad\qquad (p_1 \neq \mathit{FamilyOrAssemblyAccessibility} \lor \\
\qquad\qquad\quad (p_1 = \mathit{FamilyOrAssemblyAccessibility} \land \\
\qquad\qquad\quad p_2 \neq \mathit{FamilyAccessibility} \\
\qquad\qquad\quad) \\
\qquad\qquad) \\
\qquad\quad) \\
\qquad) \\
\quad)\} \\
\mathit{Cls}10 = \varnothing
\end{array}
$$

Rule 11

All types appearing in a signature shall be CLS-compliant.

Z: 9.6

$$
\begin{array}{l}
\mathit{Cls}11 == \{m : \mathit{ExposedCompliantMember} \mid \\
\quad (\exists\, t : \mathit{Type} \bullet (\mathit{HasMemberSignature}(m)\ \underline{\mathit{ContainsType}}\ t \land \\
\qquad t \notin \mathit{CompliantType}))\} \\
\mathit{Cls}11 = \varnothing
\end{array}
$$

We assume *type* means all types, for example, it includes managed and unmanaged pointers.

Does the preamble apply to the word *types* in Rule 11? Possibly. It cannot apply to the word *signature*, since the preamble does not mention

signatures. However, it makes more sense if we apply the preamble to *members*, as follows: *all types appearing in the signature of an exposed compliant member shall be CLS-compliant.* Does the preamble apply to both *types* and *members*? It is probably sufficient to apply it only to members, since Rule 12 ensures that, if the members are exposed, so too are the types.

Rule 12

> *The visibility and accessibility of types and members shall be such that types in the signature of any member shall be visible and accessible whenever the member itself is visible and accessible. For example, a public method that is visible outside its assembly shall not have an argument whose type is visible only within the assembly.*

Z: 9.7

$$
\begin{array}{l}
\mathit{Cls}12 == \{ m : \mathit{ExposedCompliantMember} \mid \\
\quad \exists\, t : \mathit{Type} \bullet \\
\quad \mathit{HasMemberSignature}(m) \ \underline{\mathit{ContainsType}}\ t \land \\
\quad (\\
\qquad \neg\, (\mathit{HasTypeVisibility}(t)\ \underline{\mathit{VisibilityGreater}} \\
\qquad\quad \mathit{HasVisibility}(\mathit{DefinedBy}(m)) \lor \\
\qquad \mathit{HasTypeVisibility}(t) = \\
\qquad\quad \mathit{HasVisibility}(\mathit{DefinedBy}(m)) \\
\qquad) \lor \\
\qquad (\mathit{Atomic}^{\sim}(t) \in \mathit{NestedNamedType} \land \\
\qquad \neg\, (\mathit{HasTypeAccessibility}(t)\ \underline{\mathit{AccessibilityGreaterOrEqual}} \\
\qquad\quad \mathit{HasMemberAccessibility}(m)) \\
\qquad) \\
\quad)\} \\
\mathit{Cls}12 = \varnothing
\end{array}
$$

We assume *type* means all types, for example, it includes managed and unmanaged pointers.

See Appendix E.6 for a discussion of this rule. If Rule 12 is not a CLS rule, but applies to all types and members, then it has an impact on the definition of visibility and accessibility on *Type*: it eliminates the need to define visibility and accessibility of a type t as the minimum of the visibilities and accessibilities of the types from which t was built. It does not invalidate the definitions in Sections 8.1 and 8.2, but it provides an opportunity to simplify them.

Rule 13

The value of a literal static is specified through the use of field initialization metadata (see Partition II). A CLS compliant literal must have a value specified in field initialization metadata that is of exactly the same type as the literal (or of the underlying type, if that literal is an enum).

This rule cannot be stated formally within the scope of our model, since we do not model field initialization metadata.

Rule 14

Typed references are not CLS-compliant.

Z: 9.8

$Cls14 == TypedrefType \cap ExposedCompliantNamedType$

$Cls14 = \varnothing$

$TypedrefType == \{C_R\}$ (Section 2.14). In the real-world, it is sufficient if implementations mark C_R as non-compliant. Rule 11 then ensures that C_R cannot be used in the signature of an exposed compliant member. Since C_R is a value type, it is sealed, so no other named type can inherit it.

Rule 15

The varargs constraint is not part of the CLS, and the only calling convention supported by the CLS is the standard managed calling convention.

Z: 9.9

$$\begin{aligned} &Cls15 == \{m : ExposedCompliantMember \mid \\ &\quad \exists\, c : CallingConvention \bullet \\ &\qquad (HasMemberSignature(m)\ \underline{MethodCallingConvention}\ c \wedge \\ &\qquad c = VarargCallingConvention)\} \end{aligned}$$

$Cls15 = \varnothing$

Rule 16

Arrays shall have elements with a CLS-compliant type and all dimensions of the array shall have lower bounds of zero. Only the fact that an item is an array and the element type of the

array shall be required to distinguish between overloads. When overloading is based on two or more array types the element types shall be named types.

Z: 9.10

$$Cls16a == \{t : ExposedCompliantType \mid \exists s : ArrayShape;\ t_1 : NonCompliantType \bullet (s \mapsto t_1)\ \underline{Ary}\ t\}$$

$$Cls16a = \varnothing$$

$$Cls16b == \{t : ExposedCompliantType \mid \exists s : ArrayShape;\ n : \mathbb{N}_1;\ t_1 : Type \bullet (s \mapsto t_1)\ \underline{Ary}\ t \land NthLowerBound(s, n) \neq 0 \}$$

$$Cls16b = \varnothing$$

Rule 17

Unmanaged pointer types are not CLS-compliant.

Z: 9.11

$$Cls17 == ExposedCompliantType \cap UnmanagedPointerType$$

Theorem 9.4

$Cls17 = \varnothing$

Rule 17 cannnot be "broken". In the definition of *NonCompliantType* (Section 7.3), we constrained each unmanaged pointer type to be a non-compliant type. Therefore, we need not state $Cls17 = \varnothing$ as an axiom, since it is provable as a theorem. If we had not constrained each unmanaged pointer type to be a non-compliant type, and we had stated $Cls17 = \varnothing$ as an axiom, then our model would not be satisfiable, as Rule 17 would be a logical contradiction; see Rule 3 for a similar example.

Rule 18

CLS-compliant interfaces shall not require the definition of non-CLS compliant methods in order to implement them.

Z: 9.12

$$Cls18 == \{i : ExposedCompliantNamedType \mid \\ \quad i \in Interface \land \\ \quad (\exists m : NonCompliantMember \bullet m \ \underline{DefinedBy} \ i)\}$$

$$Cls18 = \varnothing$$

There are two ways an interface i can require the definition of a method m in order to implement i: either m is defined by i, or i requires implementation of an interface i_1 and m is defined by i_1. This rule prohibits the first case, where i is an exposed compliant named type and m is a non-compliant member. Rule 20 prohibits the second case, where i is an exposed compliant named type and i_1 is a non-compliant named type. Note that in the second case, if m is non-compliant then, by Rule 18, i_1 is also non-compliant.

Rule 19

CLS-compliant interfaces shall not define static methods, nor shall they define fields.

Z: 9.13

$$Cls19 == \{i : ExposedCompliantNamedType \mid \\ \quad i \in Interface \land \\ \quad (\exists m : MEMBER \bullet \\ \quad\quad m \ \underline{DefinedBy} \ i \land \\ \quad\quad m \in StaticMethod \cup Field \\ \quad)\}$$

$$Cls19 = \varnothing$$

Rule 20

CLS-compliant classes, value types, and interfaces shall not require the implementation of non-CLS-compliant interfaces.

Z: 9.14

$$Cls20 == \{t : ExposedCompliantNamedType \mid \\ \quad \exists i : NonCompliantNamedType \bullet \\ \quad\quad i \in Interface \land \\ \quad\quad t \ \underline{Inherits} \ i\}$$

$$Cls20 = \varnothing$$

Rule 21

An object constructor shall call some class constructor of its base class before any access occurs to inherited instance data. This does not apply to value types, which need not have constructors.

Code generation is outside the scope of our model.

Rule 22

An object constructor shall not be called except as part of the creation of an object, and an object shall not be initialized twice.

Code generation is outside the scope of our model.

Rule 23

System.Object is CLS-compliant. Any other CLS-compliant class shall inherit from a CLS-compliant class.

Z: 9.15

$$
\begin{array}{l}
\mathit{Cls23} == \{t : \mathit{ExposedCompliantNamedType} \mid \\
\quad t \in \mathit{Class} \land \\
\quad (\exists\, t_1 : \mathit{NonCompliantNamedType} \bullet t \ \underline{\mathit{DirectlyInherits}}\ t_1)\} \\
\mathit{Cls23} = \varnothing
\end{array}
$$

We assume that *inherit* means *directly inherit*. Otherwise, the rule is trivial, since every class inherits C_J, which Section 7.4 specifies is CLS compliant.

Rule 24

The methods that implement the getter and setter methods of a property shall be marked SpecialName in the metadata.

Z: 9.16

$$
\begin{array}{l}
\mathit{Cls24} == \{p : \mathit{ExposedCompliantMember} \mid \\
\quad (\exists\, m : \mathit{Method} \bullet \\
\qquad (\\
\qquad\quad m\ \underline{\mathit{SetterOf}}\ p \lor \\
\qquad\quad m\ \underline{\mathit{GetterOf}}\ p \\
\qquad) \land \\
\qquad \neg\, (m\ \underline{\mathit{HasMemberAttribute}}\ \mathit{SpecialNameMemberAttribute}) \\
\quad)\} \\
\mathit{Cls24} = \varnothing
\end{array}
$$

Neither our model nor ECMA specifies that a property and its accessors must be all compliant or all non-compliant (Appendix E.8). Therefore, in accordance with the preamble, we apply Rule 24 to each exposed compliant property and its associated accessors, whether the associated accessors are compliant or non-compliant.

ECMA does not specify that *other* property methods should be marked *SpecialName*. Compare this with Rule 29.

Should this rule be more general than a CLS rule?

Rule 25

> *The accessibility of a property and of its accessors shall be identical.*
>
> **Z: 9.17**

$$\begin{aligned}
&\mathit{Cls}25 == \{p : \mathit{ExposedCompliantMember} \mid \\
&\quad \exists\, m : \mathit{Method} \bullet \\
&\qquad (p \in \mathit{Property} \land \\
&\qquad m\ \underline{\mathit{MethodSemantics}}\ p \land \\
&\qquad \mathit{HasMemberAccessibility}(m) \neq \mathit{HasMemberAccessibility}(p))\} \\
&\mathit{Cls}25 = \varnothing
\end{aligned}$$

Neither our model nor ECMA specifies that a property and its accessors must be all compliant or all non-compliant (Appendix E.8). Therefore, in accordance with the preamble, we apply Rule 25 to each exposed compliant property and its associated accessors, whether the associated accessors are compliant or non-compliant.

Our model, in conformance with ECMA I 7.5.3.2, defines accessibility on all members, including properties (Section 8.2). However, ECMA II 21.31 and II 22.1.13 do not define attribute flags for a property's accessibility, so there does not seem to be any way to represent a property's accessibility in the real world. Therefore we assume that, in the real world, the accessibility of a property is inferred from the accessibility of one of its accessors (Appendix B.6). Which one? If all accessors have the same accessibility it doesn't matter which.

Should this rule be more general than a CLS rule?

Alternative 1: If a property's accessibility can only be deduced from the accessibility of one of its accessors, and if ECMA does not specify which accessor, then it may be sensible to require all accessors, whether compliant or not, to have the same accessibility. If we accept this argument, then Rule 25 is stronger than a CLS rule.

Alternative 2: It may be even more sensible to insert criteria in ECMA to select a property's accessibility when its accessors have different accessibilities. If we accept this argument, then Rule 25 is only a CLS rule.

Alternative 3: Since the only access to a property is via its accessors, the notion of *accessibility of a property* may be irrelevant, in spite of ECMA I 7.11.3. If so, then there may be no compelling reason to either insist that the accessors of non-compliant properties should all have the same accessibility (alternative 1), or specify criteria to select a property's accessibility from its accessors (alternative 2). Suppose a non-compliant property p has a setter s and a getter g such that the accessibility of s is less that the accessibility of g. Then there may exist named types that can get p but cannot set it. This does not seem to be a problem. Similarly, it does not seem to be a problem if there exist named types that can set p but cannot get it, although this may be more questionable. If we accept this argument, then Rule 25 is only a CLS rule.

Alternative 4: If we accept argument 3 in the case of non-compliant properties, then it is only a small step to apply the same argument to compliant properties. If we accept argument 3 in the case of compliant properties, then there is no need for Rule 25.

Alternative 5: If all accessors of a property do not have the same accessibility, might compilers be unable to verify that the getter's return type is the same as the setter's last parameter type? If so, would this apply equally to non-compliant and compliant properties? If "yes", and we accept this argument, then Rule 25 is stronger than a CLS rule.

Alternative 6: Is there some other reason why Rule 25 should be a CLS rule, but not a stronger rule?

Assumption: Rule 25 is a valid CLS rule.

Is a property's "other method" considered to be an accessor? We assume "yes". Does a property's "other method" need to have the same accessibility as the getter and setter methods? We assume "yes".

Rule 26

> *A property and its accessors shall all be static, all be virtual, or all be instance.*
>
> **Z: 9.18**

$$
\begin{aligned}
&\mathit{Cls26} == \{p : \mathit{ExposedCompliantMember} \mid \\
&\quad \exists\, m_1, m_2 : \mathit{Method} \bullet \\
&\qquad p \in \mathit{Property} \land \\
&\qquad m_1\ \underline{\mathit{MethodSemantics}}\ p \land \\
&\qquad m_2\ \underline{\mathit{MethodSemantics}}\ p \land \\
&\qquad (\neg\, (\{m_1, m_2\} \subseteq \mathit{StaticMethod}) \lor \\
&\qquad\quad \neg\, (\{m_1, m_2\} \subseteq \mathit{VirtualMethod}) \lor \\
&\qquad\quad \neg\, (\{m_1, m_2\} \subseteq \mathit{InstanceMethod}) \\
&\qquad)\} \\
&\mathit{Cls26} = \varnothing
\end{aligned}
$$

Neither our model nor ECMA specifies that a property and its accessors must be all compliant or all non-compliant (Appendix E.8). Therefore, in accordance with the preamble, we apply Rule 26 to each exposed compliant property and its associated accessors, whether the associated accessors are compliant or non-compliant.

Neither our model nor ECMA defines static, virtual, or instance on properties. Therefore, we interpret this rule to mean: *The accessors of a property shall all be static, all be virtual, or all be instance.*

Should this rule be more general than a CLS rule? Suppose a non-compliant property has a static get method and an instance set method, which can presumably set a different value for each instance? What is the meaning of the get method?

Rule 27

> *The type of a property shall be the return type of the getter and the type of the last argument of the setter. The types of the parameters of the property shall be the types of the parameters to the getter and the types of all but the final parameter of the setter. All of these types shall be CLS-compliant, and shall not be managed pointers (i.e. shall not be passed by reference).*

Z: 9.19

$$\begin{aligned}&Cls27a == \{p : ExposedCompliantMember \mid (\exists\, m : Method \bullet \\ &\quad m \ \underline{GetterOf}\ p \land \\ &\quad PropertyType(HasPropertySignature(p)) \neq \\ &\quad MethodReturnType(HasMethodSignature(m)))\}\end{aligned}$$

$$Cls27a = \varnothing$$

$$\begin{aligned}&Cls27b == \{p : ExposedCompliantMember \mid (\exists\, m : Method \bullet \\ &\quad m \ \underline{SetterOf}\ p \land \\ &\quad PropertyType(HasPropertySignature(p)) \neq \\ &\quad last(MethodParameterTypes(HasMethodSignature(m))))\}\end{aligned}$$

$$Cls27b = \varnothing$$

$$\begin{aligned}&Cls27c == \{p : ExposedCompliantMember \mid (\exists\, m : Method \bullet \\ &\quad m \ \underline{GetterOf}\ p \land \\ &\quad \#(PropertyParameterTypes(HasPropertySignature(p))) \neq \\ &\quad \#(MethodParameterTypes(HasMethodSignature(m))))\}\end{aligned}$$

$$Cls27c = \varnothing$$

$$
\begin{aligned}
&\mathit{Cls27d} == \{p : \mathit{ExposedCompliantMember} \mid (\exists\, m : \mathit{Method} \bullet \\
&\quad m\ \underline{\mathit{SetterOf}}\ p \land \\
&\quad \#(\mathit{PropertyParameterTypes}(\mathit{HasPropertySignature}(p))) + 1 \neq \\
&\quad \#(\mathit{MethodParameterTypes}(\mathit{HasMethodSignature}(m))))\} \\
&\mathit{Cls27d} = \varnothing
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Cls27e} == \{p : \mathit{ExposedCompliantMember} \mid (\exists\, m : \mathit{Method}; \\
&n : \mathbb{N}_1 \bullet \\
&\quad m\ \underline{\mathit{GetterOf}}\ p \land \\
&\quad (\mathit{PropertyParameterTypes}(\mathit{HasPropertySignature}(p)))(n) \neq \\
&\quad (\mathit{MethodParameterTypes}(\mathit{HasMethodSignature}(m)))(n))\} \\
&\mathit{Cls27e} = \varnothing
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Cls27f} == \{p : \mathit{ExposedCompliantMember} \mid (\exists\, m : \mathit{Method}; \\
&n : \mathbb{N}_1 \bullet \\
&\quad m\ \underline{\mathit{SetterOf}}\ p \land \\
&\quad (\mathit{PropertyParameterTypes}(\mathit{HasPropertySignature}(p)))(n) \neq \\
&\quad (\mathit{MethodParameterTypes}(\mathit{HasMethodSignature}(m)))(n))\} \\
&\mathit{Cls27f} = \varnothing
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Cls27g} == \{p : \mathit{ExposedCompliantMember} \mid (\exists\, n : \mathbb{N}_1 \bullet \\
&\quad (\mathit{PropertyParameterTypes}(\mathit{HasPropertySignature}(p)))(n) \\
&\quad \in \mathit{ManagedPointerType})\} \\
&\mathit{Cls27g} = \varnothing
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Cls27h} == \{p : \mathit{ExposedCompliantMember} \mid (\exists\, n : \mathbb{N}_1 \bullet \\
&\quad (\mathit{PropertyParameterTypes}(\mathit{HasPropertySignature}(p)))(n) \\
&\quad \in \mathit{NonCompliantType})\} \\
&\mathit{Cls27h} = \varnothing
\end{aligned}
$$

Neither our model nor ECMA specifies that a property and its accessors must be all compliant or all non-compliant (Appendix E.8). Therefore, in accordance with the preamble, we apply Rule 27 to each exposed compliant property and its associated accessors, whether the associated accessors are compliant or non-compliant.

Rule 27 implies that each property and its getter must have the same number of parameters; and each property must have one less parameter than its setter. We explicitly state this requirement.

Rule 28

Properties shall adhere to a specific naming pattern. The SpecialName attribute referred to in CLS rule 26 shall be ignored

in appropriate name comparisons and shall adhere to identifier rules.

Z: 9.20

$$Cls28a == \{p : ExposedCompliantMember \mid (\exists\, m : Method \bullet \\ m\ \underline{GetterOf}\ p \land \\ HasMemberName(m) \neq GetName \frown HasMemberName(p))\}$$

$$Cls28a = \varnothing$$

$$Cls28b == \{p : ExposedCompliantMember \mid (\exists\, m : Method \bullet \\ m\ \underline{SetterOf}\ p \land \\ HasMemberName(m) \neq SetName \frown HasMemberName(p))\}$$

$$Cls28b = \varnothing$$

Neither our model nor ECMA specifies that a property and its accessors must be all compliant or all non-compliant (Appendix E.8). Therefore, in accordance with the preamble, we apply Rule 28 to each exposed compliant property and its associated accessors, whether the associated accessors are compliant or non-compliant.

ECMA I 9.4 specifies the *specific naming pattern*, which we paraphrase as: *if n is the name of a property, then the name of its getter shall be get_n and the name of its setter shall be set_n.*

The phrase *... shall adhere to identifier rules* appears to repeat Rule 4. We no not repeat the constraint here.

What does it mean to say: *the SpecialName attribute ... shall be ignored in appropriate name comparisons*? Alternatively, what would be the outcome of a comparision if the SpecialName attribute were not ignored? Nowhere does ECMA seem to provide specific instructions about the way the SpecialName attribute affects name comparisons. In the absence of such instructions, a direction to ignore the SpecialName attribute has little meaning.

Rule 29

The methods that implement an event shall be marked SpecialName in the metadata.

Z: 9.21

$$
\begin{array}{l}
Cls29 == \{ e : ExposedCompliantMember \mid \\
\quad (\exists\, m : Method \bullet \\
\quad\quad (\\
\quad\quad\quad m\ \underline{AddOnOf}\ e \lor \\
\quad\quad\quad m\ \underline{RemoveOnOf}\ e \lor \\
\quad\quad\quad m\ \underline{FireOf}\ e \lor \\
\quad\quad\quad m\ \underline{EventOtherOf}\ e \\
\quad\quad) \land \\
\quad\quad \neg\, (m\ \underline{HasMemberAttribute}\ SpecialNameMemberAttribute) \\
\quad)\} \\
Cls29 = \varnothing
\end{array}
$$

Neither our model nor ECMA specifies that an event and its accessors must be all compliant or all non-compliant (Appendix E.8). Therefore, in accordance with the preamble, we apply Rule 29 to each exposed compliant event and its associated accessors, whether the associated accessors are compliant or non-compliant.

Rule 30

The accessibility of an event and of its accessors shall be identical.

Z: 9.22

$$
\begin{array}{l}
Cls30 == \{ m : ExposedCompliantMember \mid \\
\quad \exists\, e : Event \bullet \\
\quad\quad (m\ \underline{MethodSemantics}\ e \land \\
\quad\quad HasMemberAccessibility(m) \neq HasMemberAccessibility(e))\} \\
Cls30 = \varnothing
\end{array}
$$

Neither our model nor ECMA specifies that an event and its accessors must be all compliant or all non-compliant (Appendix E.8). Therefore, in accordance with the preamble, we apply Rule 30 to each exposed compliant event and its associated accessors, whether the associated accessors are compliant or non-compliant.

See Section 9 for a discussion of the accessibility of properties; similar arguments about events apply to Rule 30.

Rule 31

The add and remove methods for an event shall both either be present or absent.

Z: 9.23

$$
\begin{aligned}
&Cls31 == \{e : ExposedCompliantMember \mid \\
&\quad (\\
&\qquad \exists\, m : Method \bullet m \ \underline{AddOnOf}\ e \land \\
&\qquad \neg\, (\exists\, m : Method \bullet m \ \underline{RemoveOnOf}\ e) \\
&\quad) \lor \\
&\quad (\\
&\qquad \exists\, m : Method \bullet m \ \underline{RemoveOnOf}\ e \land \\
&\qquad \neg\, (\exists\, m : Method \bullet m \ \underline{AddOnOf}\ e) \\
&\quad)\} \\
&Cls31 = \varnothing
\end{aligned}
$$

Neither our model nor ECMA specifies that an event and its accessors must be all compliant or all non-compliant (Appendix E.8). Therefore, in accordance with the preamble, we apply Rule 31 to each exposed compliant event and its associated accessors, whether the associated accessors are compliant or non-compliant.

Rule 32

The add and remove methods for an event shall each take one parameter whose type defines the type of the event and that shall be derived from System.Delegate.

Z: 9.24

$$
\begin{aligned}
&Cls32a == \{e : ExposedCompliantMember \mid (\exists\, m : Method \bullet \\
&\quad (m \ \underline{AddOnOf}\ e \lor m \ \underline{RemoveOnOf}\ e) \land \\
&\quad \#(MethodParameterTypes(HasMethodSignature(m))) \neq 1)\} \\
&Cls32a = \varnothing
\end{aligned}
$$

$$
\begin{aligned}
&Cls32b == \{e : ExposedCompliantMember \mid (\exists\, m : Method \bullet \\
&\quad (m \ \underline{AddOnOf}\ e \lor m \ \underline{RemoveOnOf}\ e) \land \\
&\quad EventType(HasEventSignature(e)) \neq \\
&\qquad MethodParameterTypes(HasMethodSignature(m))(1))\} \\
&Cls32b = \varnothing
\end{aligned}
$$

$$
\begin{aligned}
&Cls32c == \{e : ExposedCompliantMember \mid \\
&\quad \neg\, (Atomic^{\sim}(EventType(HasEventSignature(e))) \\
&\qquad \underline{Inherits}\ C_D)\} \\
&Cls32c = \varnothing
\end{aligned}
$$

Neither our model nor ECMA specifies that an event and its accessors must be all compliant or all non-compliant (Appendix E.8). Therefore, in

accordance with the preamble, we apply Rule 32 to each exposed compliant event and its associated accessors, whether the associated accessors are compliant or non-compliant.

In spite of the phrase *defines the type of the event*, we cannot take Rule 32 to be a definition of event type. (If we did, event type would be defined only for compliant events.) Therefore, we interpret Rule 32 as follows: *the add and remove methods for an event shall each take one parameter whose type is the same as the type of the event and . . . .*

Is this a CLS rule? Can non-compliant events have add and remove methods with other than one parameter? Or with a parameter whose type is not the same as the type of the event? Can a non-compliant event have a type that is not derived from System.Delegate?

Rule 33

> *Events shall adhere to a specific naming pattern. The SpecialName attribute referred to in CLS rule 31 shall be ignored in appropriate name comparisons and shall adhere to identifier rules.*

Z: 9.25

$$\begin{aligned}&\mathit{Cls33a} == \{e : \mathit{ExposedCompliantMember} \mid (\exists\, m : \mathit{Method} \bullet\\&\quad m\ \underline{\mathit{AddOnOf}}\ e \land\\&\quad \mathit{HasMemberName}(m) \neq \mathit{AddName} \frown \mathit{HasMemberName}(e))\}\\&\mathit{Cls33a} = \varnothing\end{aligned}$$

$$\begin{aligned}&\mathit{Cls33b} == \{e : \mathit{ExposedCompliantMember} \mid (\exists\, m : \mathit{Method} \bullet\\&\quad m\ \underline{\mathit{RemoveOnOf}}\ e \land\\&\quad \mathit{HasMemberName}(m) \neq \mathit{RemoveName} \frown \mathit{HasMemberName}(e))\}\\&\mathit{Cls33b} = \varnothing\end{aligned}$$

$$\begin{aligned}&\mathit{Cls33c} == \{e : \mathit{ExposedCompliantMember} \mid (\exists\, m : \mathit{Method} \bullet\\&\quad m\ \underline{\mathit{FireOf}}\ e \land\\&\quad \mathit{HasMemberName}(m) \neq \mathit{RaiseName} \frown \mathit{HasMemberName}(e))\}\\&\mathit{Cls33c} = \varnothing\end{aligned}$$

Neither our model nor ECMA specifies that an event and its accessors must be all compliant or all non-compliant (Appendix E.8). Therefore, in accordance with the preamble, we apply Rule 33 to each exposed compliant event and its associated accessors, whether the associated accessors are compliant or non-compliant.

ECMA I 9.4 specifies the *specific naming pattern*, which we paraphrase as: *if n is the name of an event, then the name of its add method shall be*

add_n; the name of its remove method shall be remove_n; and the name of its fire method shall be raise_n.

The phrase *... shall adhere to identifier rules* appears to repeat Rule 4. We no not repeat the constraint here.

What does it mean to say: *the SpecialName attribute ... shall be ignored in appropriate name comparisons*? Alternatively, what would be the outcome of a comparision if the SpecialName attribute were not ignored? Nowhere does ECMA seem to provide specific instructions about the way the SpecialName attribute affects name comparisons. In the absence of such instructions, a direction to ignore the SpecialName attribute has little meaning.

Rule 34

> *The CLS only allows a subset of the encodings of custom attributes. The only types that shall appear in these encodings are: System.Type, System.String, System.Char, System.Boolean, System.Byte, System.Int16, System.Int32, System.Int64, System.Single, System.Double, and any enumeration type based on a CLS- compliant base integer type.*

This rule cannot be stated formally within the scope of our model, because we do not model custom attributes.

Rule 35

> *The CLS does not allow publicly visible required modifiers (modreq, see Partition II), but does allow optional modifiers (modopt, see Partition II) they do not understand.*

This rule cannot be stated formally within the scope of our model, because we do not model modreq and modopt.

Rule 36

> *Global static fields and methods are not CLS-compliant.*

Z: 9.26

$$
\begin{aligned}
&\mathit{Cls}36 == \{m : \mathit{ExposedCompliantMember} \mid \\
&\quad (\exists\, t : \mathit{GlobalNamedType} \bullet \\
&\qquad m \ \underline{\mathit{DefinedBy}}\ t \land \\
&\qquad m \in \mathit{StaticMethod} \cup \mathit{StaticField} \\
&\quad)\} \\
&\mathit{Cls}36 = \varnothing
\end{aligned}
$$

Rule 37

> *Only properties and methods may be overloaded.*

Rule 6 prohibits overloading of fields. We do not repeat the constraint here.

Rule 38

> *Properties, instance methods, and virtual methods may be overloaded based only on the number and types of their parameters, except the conversion operators named op_Implicit and op_Explicit which may also be overloaded based on their return type.*

This rule cannot be stated formally within the scope of our model, because we do not model op_Implicit and op_Explicit.

Rule 39

> *If either op_Implicit or op_Explicit is overloaded on its return type, an alternate means of providing the coercion shall be provided.*

This rule cannot be stated formally within the scope of our model, because we do not model op_Implicit and op_Explicit; and there is no way to identify *the alternate means of providing the coercion.*

Rule 40

> *Objects that are thrown shall be of type System.Exception or inherit from it. Nonetheless, CLS compliant methods are not required to block the propagation of other types of exceptions.*

This rule cannot be stated formally within the scope of our model, because we do not model *throw.*

Rule 41

> *Attributes shall be of type System.Attribute, or inherit from it.*

This rule cannot be stated formally within the scope of our model, because we do not model attributes.

Part III

Appendices

Appendix A

Index of Constants

This Appendix indexes the constants, identifies the formal undefined constants, and indicates how the value of each constant is determined. See Section 1.2 for further information.

Term	*Introduced*	*Defined*	*in terms of*
AbstractClass	Z2.78	Yes	*NonAbstractClass*, *Class*
AbstractMethod	Z3.32	Yes	*Method*, et.al.
AbstractMethodAttribute	Z3.20	Yes	Z free-type
AbstractNamedType	Z2.22	Yes	*NamedType*, *NonAbstractNamedType*
AbstractValueType	Z2.87	Yes	*C_E*
Accessibility	Z8.7	Yes	Z free-type
AccessibilityGreaterOrEqual	Z8.8	Yes	*PublicAccessibility*, et.al.
AddName	Z2.5	Informal	Unicode characters
AddOnOf	Z3.13	No	language
ArrayElementInstance	Z4.63	Yes	*InstanceOf*, *ArrayElementType*
ArrayElementType	Z4.61	Yes	*CompoundComponentType*
ArrayInstance	Z4.62	Yes	*ObjectInstanceOf*, *ArrayType*
ArrayInstanceOf	Z4.67	Yes	*ArrayInstance*, et.al.
ArrayShape	Z4.53	Yes	*Dimension*
ArrayType	Z4.60	Yes	*AtomicNamedType*, *Atomic*, *Inherits*, *C_A*
Ary	Z4.64	No	given
AryInst	Z4.65	No	given
ASSEMBLY	Z2.35	No	given
AssemblyAccessibility	Z8.7	Yes	Z free-type
AssemblyVisibility	Z8.1	Yes	Z free-type
Atomic	Z4.4	No	given
AtomicNamedType	Z4.5	Yes	*Atomic*
BooleanType	Z2.83	Yes	*C_B*
Box	Z4.72	Yes	*UbvtInst*, *ObjectInst*
BoxedValueInstance	Z4.69	Yes	*Value*, *ObjectInst*
BoxedValueInstanceOf	Z4.70	Yes	*ObjectInstanceOf*, *BoxedValueType*
BoxedValueType	Z4.68	Yes	*ValueType*, *Atomic*
BoxingType	Z4.71	Yes	*Ubvt*, *Atomic*
Builds	Z6.9	Yes	*DirectlyBuilds*
BuiltinClsIntegerType	Z7.35	Yes	*BuiltinClsValueType*, *IntegerType*
BuiltinClsValueType	Z7.34	Yes	*BuiltinValueType*, *CompliantNamedType*
BuiltinReferenceType	Z2.92	Yes	*C_J*, *C_S*
BuiltinValueType	Z2.88	Yes	*IntegerType*, et.al.
C_A	Z2.50	Yes	*SystemArrayName*
CallingConvention	Z5.1	Yes	Z free-type
C_B	Z2.63	Yes	*SystemBooleanName*
C_C	Z2.66	Yes	*SystemCharName*
CCallingConvention	Z5.1	Yes	Z free-type
C_D	Z2.51	Yes	*SystemDelegateName*

Term	*Introduced*	*Defined*	*in terms of*
C_E	Z2.48	Yes	*SystemEnumName*
C_F32	Z2.64	Yes	*SystemSingleName*
C_F64	Z2.65	Yes	*SystemDoubleName*
CharType	Z2.85	Yes	C_C
C_I	Z2.61	Yes	*SystemIntPtrName*
C_I16	Z2.55	Yes	*SystemInt16Name*
C_I32	Z2.57	Yes	*SystemInt32Name*
C_I64	Z2.59	Yes	*SystemInt64Name*
C_I8	Z2.53	Yes	*SystemSByteName*
C_J	Z2.46	Yes	*SystemObjectName*
Class	Z2.68	Yes	*NamedType*, *Inherits*, et.al.
ClsAttribute	Z7.1	Yes	Z free-type
CompilerControlledAccessibility	Z8.7	Yes	Z free-type
CompliantAssembly	Z7.20	Yes	*ExplicitlyCompliantAssembly*
CompliantAttribute	Z7.1	Yes	Z free-type
CompliantElementaryType	Z7.29	Yes	*ElementaryType*, *NonCompliantElementaryType*
CompliantMember	Z7.26	Yes	*UnmarkedMember*, *DefinedBy*, *CompliantNamedType*
CompliantNamedType	Z7.24	Yes	*UnmarkedNamedTypeNestedIn*, et.al.
CompliantTopLevelNamedType	Z7.22	Yes	*TopLevelNamedType*, et.al.
CompliantType	Z7.31	Yes	*Type*, *NonCompliantType*
CompoundComponentType	Z4.21	Yes	*PassableType*, et.al.
ContainedBy	Z2.37	No	configuration
ContainsType	Z5.20	Yes	*Signature*, et.al.
CONVENTIONALCLASS	Z2.7	No	source code
CONVENTIONALOBJECT	Z2.12	No	given
ConventionalObjectIdentity	Z2.15	No	given
ConventionalObjectState	Z2.17	No	given
C_R	Z2.49	Yes	*SystemTypedReferenceName*
C_S	Z2.52	Yes	*SystemStringName*
C_U	Z2.62	Yes	*SystemUIntPtrName*
C_U16	Z2.56	Yes	*SystemUInt16Name*
C_U32	Z2.58	Yes	*SystemUInt32Name*
C_U64	Z2.60	Yes	*SystemUInt64Name*
C_U8	Z2.54	Yes	*SystemByteName*
C_V	Z2.47	Yes	*SystemValueTypeName*
DefaultCallingConvention	Z5.1	Yes	Z free-type
DefinedBy	Z3.3	No	language
DelegateType	Z4.23	Yes	C_D, et.al.
Dimension	Z4.51	Yes	*LowerBound*, *UpperBound*
DirectlyBuilds	Z6.8	Yes	*Type*, et.al.
DirectlyInherits	Z2.9	No	language

Term	*Introduced*	*Defined*	*in terms of*
DirectlyNestedIn	Z2.24	No	language
Elementary	Z4.17	Yes	*Atomic*, *Ubvt*, *ObjRef*
ElementaryType	Z4.18	Yes	*Void*, *Elementary*
EnumLiteralField	Z6.2	Yes	*Field*, *EnumType*, et.al.
EnumType	Z2.90	Yes	*DirectlyInherits*, *C_E*
EnumValueField	Z6.1	Yes	*Field*, *EnumType*, et.al.
Event	Z3.2	No	language
EventOtherOf	Z3.16	No	language
EventSignature	Z5.6	Yes	*EvSig*
EventType	Z5.17	Yes	*Signature*, et.al.
EvSig	Z5.3	No	language
ExactInstanceOf	Z2.16	No	given
ExplicitlyCompliantAssembly	Z7.5	Yes	*HasAssemblyClsAttribute*, et.al.
ExplicitlyCompliantNamedType	Z7.8	Yes	*HasNamedTypeClsAttribute*, et.al.
ExplicitlyCompliantNestedNamed−Type	Z7.10	Yes	*ExplicitlyCompliantNamedType*, *NestedNamedType*
ExplicitlyCompliantTopLevel−NamedType	Z7.9	Yes	*ExplicitlyCompliantNamedType*, *TopLevelNamedType*
ExplicitlyNonCompliantAssembly	Z7.6	Yes	*HasAssemblyClsAttribute*, et.al.
ExplicitlyNonCompliantMember	Z7.17	Yes	*HasMemberClsAttribute*, et.al.
ExplicitlyNonCompliantNamedType	Z7.11	Yes	*HasNamedTypeClsAttribute*, *NonCompliantAttribute*
ExplicitlyNonCompliantNested−NamedType	Z7.13	Yes	*ExplicitlyNonCompliantNamedType*, *NestedNamedType*
ExplicitlyNonCompliantTopLevel−NamedType	Z7.12	Yes	*ExplicitlyNonCompliantNamedType*, *TopLevelNamedType*
ExportedFrom	Z2.38	No	configuration
ExposedCompliantMember	Z8.26	Yes	*ExposedMember*, *CompliantMember*
ExposedCompliantNamedType	Z8.24	Yes	*ExposedNamedType*, *CompliantNamedType*
ExposedCompliantType	Z8.25	Yes	*ExposedType*, *CompliantType*
ExposedEvent	Z8.22	Yes	*ExposedMember*, *Event*
ExposedField	Z8.21	Yes	*ExposedMember*, *Field*
ExposedMember	Z8.19	Yes	*MEMBER*, et.al.
ExposedMethod	Z8.20	Yes	*ExposedMember*, *Method*
ExposedNamedType	Z8.16	Yes	*ExposedTopLevelNamedType*, *ExposedNestedNamedType*
ExposedNestedNamedType	Z8.15	Yes	*NestedNamedType*, et.al.
ExposedProperty	Z8.23	Yes	*ExposedMember*, *Property*
ExposedTopLevelNamedType	Z8.14	Yes	*TopLevelNamedType*, *PublicVisibility*, et.al.
ExposedType	Z8.18	Yes	*Type*, *NonExposedType*
FamilyAccessibility	Z8.7	Yes	Z free-type

Term	*Introduced*	*Defined*	*in terms of*
FamilyAndAssemblyAccessibility	Z8.7	Yes	Z free-type
FamilyOrAssemblyAccessibility	Z8.7	Yes	Z free-type
FastcallCallingConvention	Z5.1	Yes	Z free-type
Field	Z3.2	No	language
FieldAttribute	Z3.21	Yes	Z free-type
FieldSignature	Z5.5	Yes	*FldSig*
FieldType	Z5.16	Yes	*Signature*, et.al.
FinalMethodAttribute	Z3.20	Yes	Z free-type
FireOf	Z3.15	No	language
FldSig	Z5.3	No	language
FloatingType	Z2.84	Yes	C_F32, C_F64
FnPtr	Z4.14	No	given
FnPtrInst	Z4.37	No	given
FunctionPointerInstance	Z4.38	Yes	*FnPtrInst*
FunctionPointerInstanceOf	Z4.47	Yes	*FnPtrInst*, et.al.
FunctionPointerType	Z4.15	Yes	*FnPtr*
GetName	Z2.5	Informal	Unicode characters
GetterOf	Z3.11	No	language
GlobalName	Z2.5	Informal	Unicode characters
GlobalNamedType	Z2.30	Yes	*GlobalName*, et.al.
HasAssemblyClsAttribute	Z7.2	No	configuration
HasAssemblyName	Z2.36	No	configuration
HasConventionalClassName	Z2.8	No	language
HasElementaryTypeAccessibility	Z8.12	Yes	*HasNestedNamedTypeAccessibility*, *Elementary*
HasElementaryTypeVisibility	Z8.5	Yes	*HasVisibility*, et.al.
HasEventName	Z3.6	No	language
HasEventSignature	Z5.10	No	language
HasExplicitVisibility	Z8.3	No	language
HasFieldAttribute	Z3.24	No	language
HasFieldName	Z3.5	No	language
HasFieldSignature	Z5.9	No	language
HasMemberAccessibility	Z8.9	No	language
HasMemberAttribute	Z3.22	No	language
HasMemberClsAttribute	Z7.4	No	language
HasMemberName	Z3.9	Yes	*HasMethodName*, et.al.
HasMemberSignature	Z5.12	Yes	*HasMethodSignature*, et.al.
HasMethodAttribute	Z3.23	No	language
HasMethodName	Z3.4	No	language
HasMethodSignature	Z5.8	No	language
HasNamedTypeClsAttribute	Z7.3	No	language
HasNamedTypeName	Z2.20	Yes	*HasConventionalClassName*
HasNestedNamedTypeAccessibility	Z8.11	No	language

Term	*Introduced*	*Defined*	*in terms of*
HasPropertyName	Z3.7	No	language
HasPropertySignature	Z5.11	No	language
HasRank	Z4.55	Yes	*ArrayShape*, *Rank*
HasTypeAccessibility	Z8.13	Yes	*Builds*, *Accessibility*
HasTypeVisibility	Z8.6	Yes	*Builds*, *Visibility*, et.al.
HasVisibility	Z8.4	Yes	*HasExplicitVisibility*, et.al.
HideBySignatureMethodAttribute	Z3.20	Yes	Z free-type
Hides	Z6.14	Yes	*MEMBER*, et.al.
IDENTITY	Z2.13	No	given
Implements	Z2.94	Yes	*Inherits*, *Interface*
ImplementsMethod	Z6.15	Yes	*Method*, et.al.
ImportedInto	Z2.39	No	configuration
Inherits	Z2.10	Yes	*DirectlyInherits*
Instance	Z4.24	No	given
InstanceConstructorName	Z2.5	Informal	Unicode characters
InstanceField	Z3.38	Yes	*Field*, et.al.
InstanceMethod	Z3.36	Yes	*Method*, et.al.
InstanceOf	Z4.41	Yes	*ObjectInstanceOf*, et.al.
IntegerType	Z2.82	Yes	*C_I8*, et.al.
Interface	Z2.70	Yes	*NamedType*, *Inherits*, et.al.
LiteralFieldAttribute	Z3.21	Yes	Z free-type
LowerBound	Z4.49	Yes	$\mathbb{Z}$
ManagedPointerInstance	Z4.34	Yes	*ManPtrInst*
ManagedPointerInstanceOf	Z4.45	Yes	*ManPtrInst*, et.al.
ManagedPointerType	Z4.11	Yes	*ManPtr*
ManPtr	Z4.10	No	given
ManPtrInst	Z4.33	No	given
MEMBER	Z3.1	No	language
MemberAttribute	Z3.19	Yes	Z free-type
Method	Z3.2	No	language
MethodAttribute	Z3.20	yes	Z free-type
MethodCallingConvention	Z5.13	Yes	*Signature*, et.al.
MethodParameterTypes	Z5.15	Yes	*Signature*, et.al.
MethodReturnType	Z5.14	Yes	*Signature*, et.al.
MethodSemantics	Z3.17	Yes	*SetterOf*, et.al.
MethodSignature	Z5.4	Yes	*MethSig*
MethSig	Z5.3	No	language
Name	Z2.2	Yes	*UNICODE*
NamedType	Z2.19	Yes	*CONVENTIONAL_CLASS*
NestedIn	Z2.25	Yes	*DirectlyNestedIn*
NestedNamedType	Z2.27	Yes	*DirectlyNestedIn*
NewslotMethodAttribute	Z3.20	Yes	Z free-type
NonAbstractClass	Z2.77	Yes	*NonAbstractNamedType*, *Class*

Term	*Introduced*	*Defined*	*in terms of*
NonAbstractMethod	Z3.33	Yes	*Method*, et.al.
NonAbstractNamedType	Z2.21	Yes	*ExactInstanceOf*
NonCompliantAssembly	Z7.21	Yes	*ASSEMBLY*, *CompliantAssembly*
NonCompliantAttribute	Z7.1	Yes	Z free-type
NonCompliantElementaryType	Z7.28	Yes	*NonCompliantNamedType*, *Elementary*, *BoxedValueType*
NonCompliantMember	Z7.27	Yes	*MEMBER*, *CompliantMember*
NonCompliantNamedType	Z7.25	Yes	*NamedType*, *CompliantNamedType*
NonCompliantType	Z7.30	Yes	*NonCompliantElementaryType*, et.al.
NonExposedType	Z8.17	Yes	*Type*, et.al.
NthLowerBound	Z4.56	Yes	*ArrayShape*, et.al.
NthProductSize	Z4.59	Yes	*ArrayShape*, et.al.
NthSize	Z4.58	Yes	*ArrayShape*, et.al.
NthUpperBound	Z4.57	Yes	*ArrayShape*, et.al.
Null	Z4.25	Yes	Z free-type
NullInstance	Z4.26	Yes	*Null*
NullInstanceOf	Z4.40	Yes	*NullInstance*, et.al.
Object	Z2.79	Yes	*ExactInstanceOf*, *Class*
ObjectInst	Z4.27	No	given
ObjectInstance	Z4.28	Yes	*ObjectInst*
ObjectInstanceOf	Z4.42	Yes	*ObjectInst*, et.al.
ObjectReference	Z2.18	Yes	*IDENTITY*
ObjectReferenceInstance	Z4.32	Yes	*ObjRefInst*
ObjectReferenceInstanceOf	Z4.44	Yes	*ObjRefInst*, et.al.
ObjectReferenceType	Z4.9	Yes	*ObjRef*
ObjRef	Z4.8	No	given
ObjRefInst	Z4.31	No	given
OverridesField	Z6.17	Yes	*Field*, et.al.
OverridesMember	Z6.18	Yes	*OverridesMethod*, *OverridesField*
OverridesMethod	Z6.16	Yes	*Method*, et.al.
ParameterTypes	Z5.2	Yes	*PassableType*
PassableType	Z4.20	Yes	*UnboxedValueType*, et.al.
PointerType	Z4.19	Yes	*ManagedPointerType*, *UnmanagedPointerType*, *FunctionPointerType*
PrivateAccessibility	Z8.7	Yes	Z free-type
Property	Z3.2	No	language
PropertyOtherOf	Z3.12	No	language
PropertyParameterTypes	Z5.19	Yes	*Signature*, et.al.
PropertySignature	Z5.7	Yes	*PropSig*
PropertyType	Z5.18	Yes	*Signature*, et.al.
PropSig	Z5.3	No	language

Term	*Introduced*	*Defined*	*in terms of*
PublicAccessibility	Z8.7	Yes	Z free-type
PublicVisibility	Z8.1	Yes	Z free-type
RaiseName	Z2.5	Informal	Unicode characters
Rank	Z4.52	Yes	$\mathbb{N}_1$
ReferenceType	Z2.81	Yes	*Class*, *Interface*
RemoveName	Z2.5	Informal	Unicode characters
RemoveOnOf	Z3.14	No	language
RequiresImplementationOf	Z2.93	Yes	*Inherits*, *Interface*
ReturnableType	Z4.22	Yes	*PassableType*, et.al.
SealedNamedType	Z2.23	Yes	*NamedType*, *DirectlyInherits*
SetName	Z2.5	Informal	Unicode characters
SetterOf	Z3.10	No	language
Signature	Z5.3	No	language
SpecialName	Z2.6	Yes	*GlobalName*, et.al.
SpecialNameMemberAttribute	Z3.19	Yes	Z free-type
STATE	Z2.14	No	given
StaticField	Z3.37	Yes	*Field*, et.al.
StaticFieldAttribute	Z3.21	Yes	Z free-type
StaticMethod	Z3.34	Yes	*Method*, et.al.
StaticMethodAttribute	Z3.20	Yes	Z free-type
StdcallCallingConvention	Z5.1	Yes	Z free-type
SystemArrayName	Z2.3	Informal	Unicode characters
SystemBooleanName	Z2.3	Informal	Unicode characters
SystemByteName	Z2.3	Informal	Unicode characters
SystemCharName	Z2.3	Informal	Unicode characters
SystemDelegateName	Z2.3	Informal	Unicode characters
SystemDoubleName	Z2.3	Informal	Unicode characters
SystemEnumName	Z2.3	Informal	Unicode characters
SystemInt16Name	Z2.3	Informal	Unicode characters
SystemInt32Name	Z2.3	Informal	Unicode characters
SystemInt64Name	Z2.3	Informal	Unicode characters
SystemIntPtrName	Z2.3	Informal	Unicode characters
SystemName	Z2.4	Yes	*SystemObjectName*, et.al.
SystemNamedType	Z2.67	Yes	*C_J*, et.al.
SystemObjectName	Z2.3	Informal	Unicode characters
SystemSByteName	Z2.3	Informal	Unicode characters
SystemSingleName	Z2.3	Informal	Unicode characters
SystemStringName	Z2.3	Informal	Unicode characters
SystemTypedReferenceName	Z2.3	Informal	Unicode characters
SystemUInt16Name	Z2.3	Informal	Unicode characters
SystemUInt32Name	Z2.3	Informal	Unicode characters
SystemUInt64Name	Z2.3	Informal	Unicode characters
SystemUIntPtrName	Z2.3	Informal	Unicode characters

Term	*Introduced*	*Defined*	*in terms of*
SystemValueTypeName	Z2.3	Informal	Unicode characters
ThiscallCallingConvention	Z5.1	Yes	Z free-type
TopLevelNamedType	Z2.28	Yes	*NamedType*, *NestedNamedType*
Type	Z4.1	No	given
TypeConstructorName	Z2.5	Informal	Unicode characters
TypedrefType	Z2.86	Yes	*C_R*
Ubvt	Z4.6	No	given
UbvtInst	Z4.29	No	given
UnboxedValueInstance	Z4.30	Yes	*UbvtInst*
UnboxedValueInstanceOf	Z4.43	Yes	*UbvtInst*, et.al.
UnboxedValueType	Z4.7	Yes	*Ubvt*
UNICODE	Z2.1	No	Unicode standard
UnmanagedPointerInstance	Z4.36	Yes	*UnmanPtrInst*
UnmanagedPointerInstanceOf	Z4.46	Yes	*UnmanPtrInst*, et.al.
UnmanagedPointerType	Z4.13	Yes	*UnmanPtr*
UnmanPtr	Z4.12	No	given
UnmanPtrInst	Z4.35	No	given
UnmarkedAssembly	Z7.7	Yes	*ASSEMBLY* , et.al.
UnmarkedMember	Z7.18	Yes	*MEMBER*, *ExplicitlyNonCompliantMember*
UnmarkedNamedType	Z7.14	Yes	*NamedType*, et.al.
UnmarkedNamedTypeNestedIn	Z7.23	Yes	*NamedType*, et.al.
UnmarkedNestedNamedType	Z7.16	Yes	*UnmarkedNamedType*, *NestedNamedType*
UnmarkedTopLevelNamedType	Z7.15	Yes	*UnmarkedNamedType*, *TopLevelNamedType*
UpperBound	Z4.50	Yes	$\mathbb{Z}$
UserDefinedOrdinaryValueType	Z2.91	Yes	*UserDefinedValueType*, *EnumType*
UserDefinedValueType	Z2.89	Yes	*ValueType*, *BuiltinValueType*, *AbstractValueType*
Value	Z2.80	Yes	*ExactInstanceOf* , *ValueType*
ValueType	Z2.69	Yes	*NamedType*, *Inherits*, et.al.
VarargCallingConvention	Z5.1	Yes	Z free-type
VirtualMethod	Z3.35	Yes	*Method*, et.al.
VirtualMethodAttribute	Z3.20	Yes	Z free-type
Visibility	Z8.1	Yes	Z free-type
VisibilityGreater	Z8.2	Yes	*PublicVisibility*, *AssemblyVisibility*
Void	Z4.2	Yes	Z free-type
VoidType	Z4.3	Yes	*Void*

Appendix B

Real-World Representation

This Appendix informally discusses issues involved in mapping our model to the real world. It suggests guidelines, but it is not prescriptive.

B.1 Conformance

There is more than one way to map our model to the real world; any implementation satisfying the constraints conforms to our model. See Section 1.3 for further discussion. It may be desirable to deem an implementation to be conformant even if it does not satisfy certain axioms; see Appendix B.5 for an example. A conformance statement may specify relaxed conditions. Our model does not prescribe a conformance statement.

B.2 Types in Assemblies

If multiple assemblies contain identically named and identically defined named types, possibly constructed from the same source files, then they are different named types. Our model is consistent with ECMA I 8.6 *A type defined identically in two different assemblies is considered two different types.*

Since every named type is contained by one assembly (Section 2.8), it follows that a class that exists as a piece of source code, but has not yet been incorporated into an assembly, does not exist in the universe of discourse.

B.3 Types and Instances

This Section suggests a possible representation of named types, types, objects, values, identities, instances, and related subsets. Many entities are represented in several places and have different representations. For example, named types are represented by source code in some programming

language; they are also represented by TypeDef rows, internal compiler data structures, tables in virtual execution systems, and elsewhere. The representations described in this Section may not be the only real-world representations.

NamedType *NamedType* contains every named type in the universe of discourse (Section 2.3). In the real world, named types generally originate as class, value type, and interface descriptions in source code. A named type also has other representations, as indicated above, including a row in a TypeDef table in some assembly. The TypeDef table is outside the scope of our model; ECMA defines the content and the logical layout of its rows. Some named types, for example, arrays (derived from C_A), need not originate as source code and need not be represented as TypeDef rows; they may be represented only in the virtual execution system. Since our model is abstract, some universes of discourse may have an infinity of named types. At any point in time only some elements of *NamedType* are represented in the real world.

Class This is a special case of a named type (Section 2.10).

ValueType This is a special case of a named type (Section 2.10).

Interface This is a special case of a named type (Section 2.10). Its TypeDef row contains a flag designating it as an interface.

SealedNamedType This is a special case of a named type (Section 2.5). Its TypeDef row contains a flag designating it as a sealed named type.

AbstractNamedType This is a special case of a named type (Section 2.4). Its TypeDef row contains a flag designating it as an abstract named type.

NonAbstractNamedType This is a special case of a named type (Section 2.4).

Type Types are represented in compilers, virtual execution systems, and elsewhere. We do not offer any representation guidelines.

ArrayType This is a special case of a type (Section 4.4.2).

Object *Object* contains every object that could potentially exist (Section 2.12). Objects are not represented in the real world. However, through the *ObjectInst* total injection, each object is associated with a distinct object instance (Figure 4.1). Object instances are represented in the real world.

Value *Value* contains every value that could potentially exist (Section 2.12). Values are not represented in the real world. However, through the *ObjectInst* total injection, each value is associated with a distinct boxed value instance (a special case of an object instance), and through the *UbvtInst* total injection each value is associated with a distinct unboxed value instance (Figure 4.3). Object instances and unboxed value instances are represented in the real world.

ObjectInstance In the real world, representations of object instances are created and destroyed by a creation and destruction mechanism. Those object instances represented in the real world at some point in time are represented by a collection of contiguous bits in a garbage collected heap. Each object instance is represented in, at most, one place at a time in the heap. Each place in the heap contains, at most, one object instance representation at a time. Object instance representations do not overlap in the heap. Each object instance representation contains, among other things, a collection of field locations, that is, the object state. Although we do not model the progression of time, the bits in a particular object instance representation change over time; after the change, the same object instance is represented in a different state. The garbage collector may move an object instance representation; after the move, the same object instance is represented.

BoxedValueInstance This is a special case of an object instance (Section 4.5, Figure 4.3).

ArrayInstance This is a special case of an object instance (Section 4.4.2). Each array instance representation contains, among other things, a sequence of array element locations.

UnboxedValueInstance Those unboxed value instances that are represented in the real world at some point in time are represented by a collection of contiguous bits, in a location such as a local variable, a field location, a parameter location, an evaluation stack location, or elsewhere. The same unboxed value instance may be represented in multiple locations at the same time. Each location may contain, at most, the representation of one unboxed value instance at a time. Our model does not model locations. Although we do not model the progression of time or locations, the bits in a particular location change over time; after the change, the location contains the representation of a different unboxed value instance.

IDENTITY Identities are not represented in the real world. However, through the *ObjRefInst* total injection, each identity is associated with a distinct object reference instance (Figure 4.2). Object reference instances are represented in the real world.

ObjectReference This is a synonym for *IDENTITY*.

ObjectReferenceInstance The *ObjRefInst* total injection, *ConventionalObjectIdentity* bijection, and *ObjectInst* total injection relate each object reference instance to a distinct object instance (Section 4.2, Figure 4.2). Note that each value has one object reference, which relates via the *ObjectInst* total injection to the corresponding boxed value instance (Figure 4.3). For each object instance represented in the real world, the corresponding object reference instance is also represented in at least one location. Each object reference instance represented in the real world is represented as the address of the corresponding object instance in a garbage collected heap. Each object reference instance represented in the real world is in some location such as a local variable, a field location, a parameter location, an evaluation stack location, or elsewhere. The same object reference instance may be represented in multiple locations at the same time. If the garbage collector moves an object instance representation, it changes the representations of the corresponding object reference instance. If an object reference instance ceases to be represented in at least one location in the real world, then the corresponding object instance also ceases to be represented, and the garbage collector may reclaim its memory.

ManagedPointerInstance The *ManPtrInst* total injection relates each managed pointer instance p to a distinct instance i (Section 4.2, Figures 4.5 and 4.6). Each managed pointer instance p represented in the real world is represented as the address of the corresponding instance i. Each managed pointer instance represented in the real world is in some location such as a local variable, a parameter location, an evaluation stack location, or elsewhere. The same managed pointer instance may be represented in multiple locations at the same time.

UnmanagedPointerInstance The *UnmanPtrInst* total injection relates each non-null unmanaged pointer instance p to a distinct instance i (Section 4.2, Figure 4.7). Each non-null unmanaged pointer instance p represented in the real world is represented as the address of the corresponding instance i. Each unmanaged pointer instance represented in the real world is in some location such as a local variable, a field location, a parameter location, an evaluation stack location, or elsewhere. The same unmanaged pointer instance may be represented in multiple locations at the same time.

FunctionPointerInstance The *FnPtrInst* total injection relates each non-null function pointer instance p to the implementation of a distinct non-abstract method m (Section 4.2, Figure 4.8). Each non-null function pointer instance p represented in the real world is represented as

the address of the corresponding method m. Each function pointer instance represented in the real world is in some location such as a local variable, a field location, a parameter location, an evaluation stack location, or elsewhere. The same function pointer instance may be represented in multiple locations at the same time. Each function pointer instance p is related via the method m to a method signature. We do not offer any representation guidelines for the relationship.

NullInstance Null instance is an instance of any type except a managed pointer type or a void type (Section 4.3). Null instance is represented as a collection of zero bits. A null instance represented in the real world is in some location such as a local variable, a field location, a parameter location, an evaluation stack location, or elsewhere. Null instance may be represented in multiple locations at the same time.

B.4 Locations

Although we do not model *location*, we suggest a possible real-world representation of field locations and array element locations.

In this Section our informal terminology distinguishes between *field* (a *member* defined by a *named type*) and *field location* (a contiguous collection of bits embedded in the representation of one object instance or one unboxed value instance).

Field Location Each field location l is a contiguous collection of bits embedded in the representation of one object instance o or one unboxed value instance v; l is associated with one field f; f has a signature s (Section 5.3); s is associated via the *FieldType* function with one compound component type t (Section 5.4); t is an atomic named type, or an unboxed value type, or an unmanaged pointer type, or a function pointer type (Section 4.1).

If t is an atomic named type, then l contains the representation of one object reference instance that references an object instance that is an instance of t.

If t is an unboxed value type, then l contains the representation of one unboxed value instance that is an instance of t.

If t is an unmanaged pointer type, then l contains the representation of one unmanaged pointer instance that is an instance of t.

If t is a function pointer type, then l contains the representation of one function pointer instance that is an instance of t.

Array Element Location Each array element location l is a contiguous collection of bits embedded in the representation of one array instance

a; l is associated with one array element instance i; i is associated via the *InstanceOf* relation with one array element type t (Section 4.4.2, Figure 4.4); t is an atomic named type, or an unboxed value type, or an unmanaged pointer type, or a function pointer type (Sections 4.4.2, 4.1).

If t is an atomic named type, then l contains the representation of one object reference instance that references an object instance that is an instance of t.

If t is an unboxed value type, then l contains the representation of one unboxed value instance that is an instance of t.

If t is an unmanaged pointer type, then l contains the representation of one unmanaged pointer instance that is an instance of t.

If t is a function pointer type, then l contains the representation of one function pointer instance that is an instance of t.

B.5 CLS Compliance in the Real World

Our model does not admit the possibility that a member or a named type marked as CLS compliant breaks a CLS rule (Section 9). However, in the real world it may be desirable for a compiler to generate a valid assembly, perhaps accompanied by a warning message, when a member or a named type marked as CLS compliant breaks a CLS rule; and it may be desirable for the virtual execution system to allow such an assembly to execute; and for tools to process it, within the limitations imposed by the error. (However, see Appendix E.4 for a discussion of the feasibility of performing accurate CLS compliance checking at compile time.)

Therefore, a conformance statement may permit an implementation to treat broken CLS rules in any way the conformance statement sees fit. Our model does not prescribe a conformance statement.

B.6 Accessibility of Properties

In the real world, the accessibility of a property may be inferred from the accessibility of one of its accessors. See Rule 25 for further discussion.

Appendix C

Proof of Theorems

Theorem: 2.1
$\mathit{GlobalNamedType} \subset \mathit{SealedNamedType}$

Proof

1. We need to show:
   (a) each element of *GlobalNamedType* is an element of *SealedNamed−Type*; and
   (b) there is an element of *SealedNamedType* not in *GlobalNamedType*.
2. Proof of 1(a) by contradiction.
3. Assume there is an element t of *GlobalNamedType* not in *SealedNamed−Type*.
4. From Z2.30: $t \in \mathit{NamedType}$.
5. From Z2.32: $t \notin \operatorname{ran} \mathit{DirectlyInherits}$.
6. From 4, 5, and Z2.23: $t \in \mathit{SealedNamedType}$.
7. 3 leads to a contradiction.
8. From 7: 1(a) is proven.
9. $\mathit{C_I8} \in \mathit{SealedNamedType}$ and $\mathit{C_I8} \notin \mathit{GlobalNamedType}$.
10. From 9: 1(b) is proven.

QED

Theorem: 2.2
$\langle \mathit{Class}, \mathit{ValueType}, \mathit{Interface} \rangle$ partitions $\mathit{NamedType}$

Proof

1. We need to show:

    (a) *Class* and *ValueType* are disjoint; and
    (b) *ValueType* and *Interface* are disjoint; and
    (c) *Interface* and *Class* are disjoint; and
    (d) *NamedType* is the union of *Class*, *ValueType*, and *Interface*.

2. Proof of 1(a) by contradiction.

3. Assume there is an element t of *NamedType* such that $t \in Class$ and $t \in ValueType$.

4. From 3 and Z2.68:

    (a) $t = C_E$; or
    (b) $t \in GlobalNamedType$; or
    (c) $t = C_J \wedge \neg\,(t\ \underline{Inherits}\ C_V)$; or
    (d) $t\ \underline{Inherits}\ C_J \wedge \neg\,(t\ \underline{Inherits}\ C_V)$.

5. (a) From 3 and Z2.69: $t \neq C_E$, so 4(a) does not hold.
    (b) From 3, Z2.31, and Z2.69: $\neg\,(t \in GlobalNamedType)$, so 4(b) does not hold.
    (c) From 3 and Z2.69: $t\ \underline{Inherits}\ C_V$, so 4(c) and 4(d) do not hold.

6. 3 leads to a contradiction, so 1(a) is proven.

7. Proof of 1(b) by contradiction.

8. Assume there is an element t of *NamedType* such that $t \in ValueType$ and $t \in Interface$.

9. From 8, Z2.69, Z2.47, and Z2.10: $t\ \underline{Inherits}\ C_J$.

10. From 8 and Z2.70: $\neg\,(t\ \underline{Inherits}\ C_J)$.

11. 8 leads to a contradiction, so 1(b) is proven.

12. Proof of 1(c) by contradiction.

13. Assume there is an element t of *NamedType* such that $t \in Interface$ and $t \in Class$.

14. From 13 and Z2.68:

    (a) $t = C_E$; or
    (b) $t \in GlobalNamedType$; or

(c) $t = C_J \wedge \neg\,(t\ \underline{Inherits}\ C_V)$; or
(d) $t\ \underline{Inherits}\ C_J \wedge \neg\,(t\ \underline{Inherits}\ C_V)$.

15. From 13, Z2.48, Z2.47, Z2.10, and Z2.70:

    (a) $C_E\ \underline{Inherits}\ C_J$ and $\neg\,(t\ \underline{Inherits}\ C_J)$, so 14(a) does not hold.
    (b) From 13, Z2.70: $t \notin \mathit{GlobalNamedType}$, so 14(b) does not hold.
    (c) From 13, Z2.70: $\neg\,(t = C_J)$, so 14(c) does not hold.
    (d) From 13, Z2.70: $\neg\,(t\ \underline{Inherits}\ C_J)$, so 14(d) does not hold.

16. 13 leads to a contradiction, so 1(c) is proven.

17. Proof of 1(d) by contradiction.

18. Assume there is an element t of *NamedType* such that t is not an element of any of *Class*, *ValueType*, *Interface*.

19. We examine the inheritance graph:

    (a) Either $t\ \underline{Inherits}\ C_J$ or $\neg\,(t\ \underline{Inherits}\ C_J)$.
    (b) If $t\ \underline{Inherits}\ C_J$, then either $t\ \underline{Inherits}\ C_V$ or $\neg\,(t\ \underline{Inherits}\ C_V)$.
    (c) If $t\ \underline{Inherits}\ C_J$ and $t\ \underline{Inherits}\ C_V$, then either $t = C_E$ or $t \neq C_E$.
    (d) If $t\ \underline{Inherits}\ C_J$ and $t\ \underline{Inherits}\ C_V$ and $t = C_E$, then $t \in \mathit{Class}$.
    (e) If $t\ \underline{Inherits}\ C_J$ and $t\ \underline{Inherits}\ C_V$ and $t \neq C_E$, then $t \in \mathit{ValueType}$.
    (f) If $t\ \underline{Inherits}\ C_J$ and $\neg\,(t\ \underline{Inherits}\ C_V)$, then $t \in \mathit{Class}$.
    (g) If $\neg\,(t\ \underline{Inherits}\ C_J)$, then either $t = C_J$ or $t \neq C_J$.
    (h) If $\neg\,(t\ \underline{Inherits}\ C_J)$ and $t = C_J$, then $t \in \mathit{Class}$.
    (i) If $\neg\,(t\ \underline{Inherits}\ C_J)$ and $t \neq C_J$, then either $t \in \mathit{GlobalNamed{-}Type}$ or $t \notin \mathit{GlobalNamedType}$.
    (j) If $\neg\,(t\ \underline{Inherits}\ C_J)$ and $t \neq C_J$ and $t \in \mathit{GlobalNamedType}$, then $t \in \mathit{Class}$.
    (k) If $\neg\,(t\ \underline{Inherits}\ C_J)$ and $t \neq C_J$ and $t \notin \mathit{GlobalNamedType}$, then $t \in \mathit{Interface}$.

20. There is no other possibility. 18 leads to a contradiction.

21. From Z2.68, Z2.69, and Z2.70: every element of *Class*, *ValueType*, and *Interface* is an element of *NamedType*.

22. From 20 and 21: 1(d) is proven.

QED

Theorem: 2.3
$\langle Object, Value \rangle$ partitions *CONVENTIONAL_OBJECT*

Proof

1. We need to show:
   (a) *Object* and *Value* are disjoint; and
   (b) *CONVENTIONAL_OBJECT* is the union of *Object* and *Value*.
2. Proof of 1(a) by contradiction.
3. Assume there is an element c of *CONVENTIONAL_OBJECT* such that $c \in Object$ and $c \in Value$.
4. From 3 and Z2.79: $c \in \mathrm{dom}(ExactInstanceOf \rhd Class)$.
5. From 4: $ExactInstanceOf(c) \in Class$.
6. From 3 and Z2.80: $c \in \mathrm{dom}(ExactInstanceOf \rhd ValueType)$.
7. From 6: $ExactInstanceOf(c) \in ValueType$.
8. From Z2.16: *ExactInstanceOf* is a function.
9. From Theorem 2.2: *Class* and *ValueType* are disjoint.
10. 5, 7, 8, and 9 are contradictory.
11. 3 leads to a contradiction, so 1(a) is proven.
12. Proof of 1(b) by contradiction.
13. Assume there is an element c of *CONVENTIONAL_OBJECT* that c is not an element of either *Object* or *Value*.
14. From Z2.16 and Z2.19: $ExactInstanceOf(c) \in NamedType$.
15. From Theorem 2.2:
    (a) $ExactInstanceOf(c) \in Class$, or
    (b) $ExactInstanceOf(c) \in ValueType$, or
    (c) $ExactInstanceOf(c) \in Interface$.
16. From 13, Z2.79, Z2.80, and 15: $ExactInstanceOf(c) \in Interface$.

17. From Z2.21: $ExactInstanceOf(c) \in NonAbstractNamedType$.

18. From Z2.22: $ExactInstanceOf(c) \notin AbstractNamedType$.

19. From 16 and 18: $ExactInstanceOf(c)$ is an element of *Interface* and not an element of *AbstractNamedType*.

20. 19 contradicts Z2.74.

21. 13 leads to a contradiction.

22. From Z2.16, Z2.79, and Z2.80: every element of *Object* and *Value* is an element of $CONVENTIONAL_OBJECT$.

23. From 21 and 22: 1(b) is proven.

QED

Theorem: 2.4

$IntegerType \subset ValueType$

Proof

1. We need to show that each of C_I8, C_U8, etc is an element of *ValueType* (Z2.82).

2. From Z2.53 through Z2.62: each of C_I8, C_U8, etc inherits C_V and none is equal to C_E.

3. From 2 and Z2.69: each of C_I8, C_U8, etc is an element of *ValueType*.

QED

Theorem: 2.5

$BooleanType \subset ValueType$

Proof

1. We need to show that C_B is an element of *ValueType* (Z2.83).

2. From Z2.63: C_B inherits C_V and $C_B \neq C_E$.

3. From 2 and Z2.69: C_B is an element of *ValueType*.

QED

Theorem: 2.6

$FloatingType \subset ValueType$

Proof

1. We need to show that each of C_F32 and C_F64 is an element of $ValueType$ (Z2.84).
2. From Z2.64 and Z2.65: each of C_F32 and C_F64 inherits C_V and neither is equal to C_E.
3. From 2 and Z2.69: each of C_F32 and C_F64 is an element of $ValueType$.

QED

Theorem: 2.7
$CharType \subset ValueType$

Proof

1. We need to show that C_C is an element of $ValueType$ (Z2.85).
2. From Z2.66: C_C inherits C_V and $C_C \neq C_E$.
3. From 2 and Z2.69: C_C is an element of $ValueType$.

QED

Theorem: 2.8
$TypedrefType \subset ValueType$

Proof

1. We need to show that C_R is an element of $ValueType$ (Z2.83).
2. From Z2.49: C_R inherits C_V and $C_R \neq C_E$.
3. From 2 and Z2.69: C_R is an element of $ValueType$.

QED

Theorem: 2.9
$AbstractValueType \subset Class$

Proof

1. We need to show that C_E is an element of $Class$ (Z2.87).
2. From Z2.68: C_E is an element of $Class$.

QED

Theorem: 2.10
$BuiltinValueType \subset ValueType$

Proof

1. We need to show that each of *IntegerType*, *BooleanType*, *FloatingType*, *CharType*, and *TypedrefType* is a subset of *ValueType* (Z2.88).
2. From Theorems 2.4 through 2.8: each of *IntegerType*, *BooleanType*, *FloatingType*, *CharType*, and *TypedrefType* is a subset of *ValueType*.

QED

Theorem: 2.11

$EnumType \subset ValueType$

Proof

1. We need to show:
   (a) each element of *EnumType* is an element of *ValueType*; and
   (b) there is an element of *ValueType* not in *EnumType*.
2. From Z2.90: $EnumType == \mathrm{dom}(DirectlyInherits \rhd \{C_E\})$.
3. From 2: each element of *EnumType* directly inherits C_E.
4. From 3, Z2.48, and Z2.10: each element of *EnumType* inherits C_V.
5. From 3: no element of *EnumType* equals C_E.
6. From 4, 5, and Z2.69: each element of *EnumType* is an element of *ValueType*.
7. From 6: 1(a) is proven.
8. From Z2.53: $C_I8 \in ValueType$ and $C_I8 \notin EnumType$.
9. From 8: 1(b) is proven.

QED

Theorem: 2.12

$EnumType \subset UserDefinedValueType$

Proof

1. We need to show:
   (a) each element of *EnumType* is an element of *UserDefinedValueType*; and
   (b) there is an element of *UserDefinedValueType* not in *EnumType*.

2. From Z2.89: $\mathit{UserDefinedValueType} == \mathit{ValueType} \setminus \mathit{BuiltinValueType} \setminus \mathit{AbstractValueType}$.

3. From Z2.11: each element of $\mathit{EnumType}$ is an element of $\mathit{ValueType}$.

4. From Z2.88, Z2.82 through Z2.86, and Z2.53 through Z2.66: no element of $\mathit{BuiltinValueType}$ is an element of $\mathit{EnumType}$.

5. From Z2.87 and Z2.48: no element of $\mathit{AbstractValueType}$ is an element of $\mathit{EnumType}$.

6. From 2, 3, 4, 5: 1(a) is proven.

7. From ECMA Library, 2.69, and 2: the value type whose name is "`System.DateTime`" is in $\mathit{UserDefinedValueType}$ and not in $\mathit{EnumType}$.

8. From 7: 1(b) is proven.

QED

Theorem: 2.13

$\mathit{BuiltinReferenceType} \subset \mathit{Class}$

Proof

1. We need to show:

   (a) each element of $\mathit{BuiltinReferenceType}$ is an element of Class; and

   (b) there is an element of Class not in $\mathit{BuiltinReferenceType}$.

2. From Z2.92: $\mathit{BuiltinReferenceType} == \{C_J, C_S\}$.

3. From Z2.68: $C_J \in \mathit{Class}$.

4. From Z2.52 and Z2.68: $C_S \in \mathit{Class}$.

5. From 2 and 3: 1(a) is proven.

6. From Z2.47 and 2: $C_V \in \mathit{Class}$ and $C_V \notin \mathit{BuiltinReferenceType}$.

7. From 6: 1(b) is proven.

QED

Theorem: 2.14

$\mathit{disjoint}\langle \mathit{AbstractValueType}, \mathit{BuiltinValueType}, \mathit{UserDefinedValueType} \rangle$

Proof

1. We need to show:

   (a) $AbstractValueType$ and $BuiltinValueType$ are disjoint; and
   (b) $BuiltinValueType$ and $UserDefinedValueType$ are disjoint; and
   (c) $UserDefinedValueType$ and $AbstractValueType$ are disjoint.

2. From Z2.87: $AbstractValueType == \{C_E\}$.

3. From Z2.88, Z2.82, Z2.83, Z2.84, Z2.85, and Z2.86: $C_E \notin Builtin-ValueType$.

4. From 2 and 3: 1(a) is proven.

5. From Z2.89: 1(b) is proven.

6. From Z2.89: 1(c) is proven.

QED

Theorem: 2.15
$\neg \exists c : Class;\ v : ValueType \bullet c \mapsto v \in Inherits$

Proof

1. Proven from Z2.23 and Z2.72.

QED

Theorem: 2.16
$\neg \exists i : Interface;\ c : Class \bullet i \mapsto c \in Inherits$

Proof

1. Proof by contradiction.

2. Assume there are elements i in $Interface$ and c in $Class$ such that $i\ \underline{Inherits}\ c$.

3. From 2 and Z2.68:

   (a) $c = C_E$; or
   (b) $c \in GlobalNamedType$; or
   (c) $c = C_J \wedge \neg\,(c\ \underline{Inherits}\ C_V)$; or
   (d) $c\ \underline{Inherits}\ C_J \wedge \neg\,(c\ \underline{Inherits}\ C_V)$.

4. From 2, Z2.48, Z2.47, and Z2.10: If 3(a) holds, then $i\ \underline{Inherits}\ C_J$, which contradicts Z2.70.

5. From 2 and Z2.32: If 3(b) holds, we have a contradiction.

6. From 2: If 3(c) holds, then i $\underline{Inherits}$ C_J, which contradicts Z2.70.
7. From 2 and Z2.10: If 3(d) holds, then i $\underline{Inherits}$ C_J, which contradicts Z2.70.
8. 2 leads to a contradiction.

QED

Theorem: 2.17

$\neg\ \exists\, i : \mathit{Interface};\ v : \mathit{ValueType} \bullet i \mapsto v \in \mathit{Inherits}$

Proof

1. Proof by contradiction.
2. Assume there are elements i in *Interface* and v in *ValueType* such that i $\underline{Inherits}$ v.
3. From 2 and Z2.69: v $\underline{Inherits}$ C_V.
4. From 2, 3, Z2.47, and Z2.10: i $\underline{Inherits}$ C_J, which contradicts Z2.70.
5. 2 leads to a contradiction.

QED

Theorem: 2.18

$\neg\ \exists\, v : \mathit{ValueType};\ c : \mathit{Class} \bullet$
$(c \neq C_J) \wedge c \neq C_V) \wedge (c \neq C_E) \wedge (v \mapsto c \in \mathit{Inherits})$

Proof

1. Proof by contradiction.
2. Assume there are elements v in *ValueType* and c in *Class* such that v $\underline{Inherits}$ c and $c \neq C_J$ and $c \neq C_V$ and $c \neq C_E$.
3. From 2 and Z2.68:
   (a) $c = C_E$; or
   (b) $c \in \mathit{GlobalNamedType}$; or
   (c) $c = C_J \wedge \neg\,(c\ \underline{Inherits}\ C_V)$; or
   (d) $c\ \underline{Inherits}\ C_J \wedge \neg\,(c\ \underline{Inherits}\ C_V)$.
4. From 2: If 3(a) holds, then $c \neq C_E$ is a contradiction.
5. From 2 and Z2.32: If 3(b) holds, we have a contradiction.

6. From 2: If 3(c) holds, then $c \neq C_J$ is a contradiction.
7. From 2 and Z2.10: If 3(d) holds, then we can show that Z2.71 is a contradiction.
8. 2 leads to a contradiction.

QED

Theorem: 2.19
$\neg \exists v_1, v_2 : \mathit{ValueType} \bullet v_1 \mapsto v_2 \in \mathit{Inherits}$

Proof

1. Proven from Z2.23 and Z2.72.

QED

Theorem: 3.1
$\mathrm{dom}\, \mathit{HasMemberName} = \mathit{MEMBER}$

Proof

1. We need to show:
   (a) each element of *MEMBER* occurs as the first component of at least one element of *HasMemberName*; and
   (b) each element of *HasMemberName* has an element of *MEMBER* as its first component.
2. From Z3.8: each element of *MEMBER* is an element of *Method*, *Field*, *Event*, or *Property*.
3. From Z3.4: each element of *Method* occurs as the first component of at least one element of *HasMethodName*. Similarly for *Field*, *Event*, and *Property*.
4. From 2, 3 and Z3.9: each element of *MEMBER* occurs as the first component of at least one element of *HasMemberName*.
5. From 4: 1(a) is proven.
6. From Z3.9, Z3.4, Z3.5, Z3.6, Z3.7: each element of *HasMemberName* has an element of *Method*, *Field*, *Event*, or *Property* as its first component.
7. From Z3.8 and 6: 1(b) is proven.

QED

Theorem: 3.2

$\forall\, m_1, m_2 : \mathit{MEMBER} \mid m_1 = m_2 \bullet$
$\quad \mathit{HasMemberName}(m_1) = \mathit{HasMemberName}(m_2)$

Proof

1. From Z3.9: for each element n of *HasMemberName*, $n \in \mathit{HasMethod-Name}$, or $n \in \mathit{HasFieldName}$, or $n \in \mathit{HasEventName}$, or $n \in \mathit{Has-PropertyName}$.
2. From Z3.4 through Z3.7: each of *HasMethodName*, *HasFieldName*, *HasEventName*, and *HasPropertyName* is a function.

QED

Theorem: 3.3

$\forall\, m : \mathit{Method} \mid$
$\quad m\ \underline{\mathit{HasMethodAttribute}}\ \mathit{StaticMethodAttribute} \bullet$
$\quad \neg\,(m\ \underline{\mathit{HasMethodAttribute}}\ \mathit{FinalMethodAttribute})$

Proof

1. From Z3.25: each method having a static method attribute does not have a virtual method attribute.
2. From Z3.28: each method having a final method attribute has a virtual method attribute.
3. From 1 and 2: no method having a static method attribute has a final method attribute.

QED

Theorem: 3.4

$\forall\, m : \mathit{Method} \mid$
$\quad m\ \underline{\mathit{HasMethodAttribute}}\ \mathit{StaticMethodAttribute} \bullet$
$\quad \neg\,(m\ \underline{\mathit{HasMethodAttribute}}\ \mathit{NewslotMethodAttribute})$

Proof

1. From Z3.25: each method having a static method attribute does not have a virtual method attribute.
2. From Z3.29: each method having a newslot method attribute has a virtual method attribute.

3. From 1 and 2: no method having a static method attribute has a newslot method attribute.

QED

Theorem: 3.5
$\langle \mathit{StaticMethod}, \mathit{VirtualMethod}, \mathit{InstanceMethod} \rangle$ partitions Method

Proof

1. We need to show:
   (a) *StaticMethod* and *VirtualMethod* are disjoint; and
   (b) *VirtualMethod* and *InstanceMethod* are disjoint; and
   (c) *InstanceMethod* and *StaticMethod* are disjoint; and
   (d) *Method* is the union of *StaticMethod*, *VirtualMethod*, and *InstanceMethod*.
2. From Z3.25, Z3.34, and Z3.35: 1(a) is proven.
3. From Z3.36: 1(b) is proven.
4. From Z3.36: 1(c) is proven.
5. From Z3.34, Z3.35, and Z3.36: each element of *StaticMethod*, *Virtual–Method*, *InstanceMethod* is an element of *Method*.
6. From Z3.36: each element of *Method* is an element of *StaticMethod*, or *VirtualMethod*, or *InstanceMethod*.
7. From 5 and 6: 1(d) is proven.

QED

Theorem: 4.1
$\mathrm{dom}\, \mathit{Elementary} = \mathit{NamedType}$

Proof

1. We need to show:
   (a) each element of *NamedType* occurs as the first component of at least one element of *Elementary*; and
   (b) each element of *Elementary* has an element of *NamedType* as its first component.
2. From Z4.4: *Atomic* is a total injection from *NamedType* to *Type*.

3. From Z4.17 and 2: each element of *NamedType* occurs as the first component of at least one element of *Atomic*.

4. From 3: 1(a) is proven.

5. From Z4.17: each element of *Elementary* has an element of dom *Atomic*, or dom *Ubvt*, or dom *ObjRef* as its first component.

6. From Z4.4, Z4.6, and Z4.8: $\operatorname{dom} Atomic = NamedType$, and $\operatorname{dom} Ubvt \subset NamedType$, and $\operatorname{dom} ObjRef \subset NamedType$.

7. From 5 and 6: each element of *Elementary* has an element of *NamedType* as its first component.

8. From 7: 1(b) is proven.

QED

Theorem: 4.2

$\forall e_1, e_2 : Elementary \mid first(e_1) \neq first(e_2) \bullet$
$\quad second(e_1) \neq second(e_2)$

Proof

1. From Z4.17: $e_1 \in Atomic$, or $e_1 \in Ubvt$, or $e_1 \in ObjRef$.

2. If $e_1 \in Atomic$, then either $e_2 \in Atomic$ or $e_2 \notin Atomic$.

3. If $e_1 \in Atomic$ and $e_2 \in Atomic$, then $second(e_1) \neq second(e_2)$, because *Atomic* is injective.

4. If $e_1 \in Atomic$ and $e_2 \notin Atomic$, then $second(e_1) \neq second(e_2)$, because the ranges of *Atomic*, *Ubvt*, and *ObjRef* are pairwise disjoint.

5. Similar arguments apply if e_1 is an element of *Ubvt* or *ObjRef*.

QED

Theorem: 4.3

$BoxedValueType \subset AtomicNamedType$

Proof

1. We need to show:

   (a) each element of *BoxedValueType* is an element of *AtomicNamedType*; and

   (b) there is an element of *AtomicNamedType* not in *BoxedValueType*.

2. From Z4.5 and Z4.68: 1(a) is proven.

3. $Atomic(C_J)$ is an element of *AtomicNamedType* and not an element of *BoxedValueType*.

4. From 3: 1(b) is proven.

QED

Theorem: 4.4
$BoxedValueInstance \subset ObjectInstance$

Proof

1. We need to show:

   (a) each element of *BoxedValueInstance* is an element of *ObjectInstance*; and

   (b) there is an element of *ObjectInstance* not in *BoxedValueInstance*.

2. From Z4.28 and Z4.69: 1(a) is proven.

3. Any exact instance of C_S is an element of *ObjectInstance* and not an element of *BoxedValueInstance*.

4. From 3: 1(b) is proven.

QED

Theorem: 4.5
$\text{dom}\, BoxedValueInstanceOf = BoxedValueInstance$

Proof

1. We need to show:

   (a) each element of *BoxedValueInstance* occurs as the first component of at least one element of *BoxedValueInstanceOf*; and

   (b) each element of *BoxedValueInstanceOf* has an element of *Boxed−ValueInstance* as its first component.

2. Proof of 1(a) by contradiction.

3. Assume there is an element i of *BoxedValueInstance* such that i is not the first component of at least one element of *BoxedValueInstanceOf*.

4. From 3 and Theorem 4.4: $i \in ObjectInstance$.

5. From 4, Z4.28, and Z4.27: there is one element v of $Object \cup Value$ such that $v\ \underline{ObjectInst}\ i$.
6. From 3, 5, and Z4.69: $v \in Value$.
7. From 6 and Z2.16: there is one element d of $NamedType$ such that $v\ \underline{ExactInstanceOf}\ d$.
8. From 6, 7 and Z2.80: $d \in ValueType$.
9. From 8, Z4.5, and Z4.4: there is one element t of $AtomicNamedType$ such that $d\ \underline{Atomic}\ t$.
10. From 8, 9 and Z4.68: $t \in BoxedValueType$.
11. From 5 and Z4.27: $i\ \underline{ObjectInst^{\sim}}\ v$.
12. From Z4.42, 11, 7, 9: $i\ \underline{ObjectInstanceOf}\ t$.
13. From 12, 10, Z4.70: $i\ \underline{BoxedValueInstanceOf}\ t$.
14. 13 contradicts 3.
15. From 14: 1(a) is proven.
16. A similar argument proves 1(b).

QED

Theorem: 4.6
$\mathrm{ran}\ BoxingType = BoxedValueType$

Proof

1. We need to show:
   (a) each element of $BoxedValueType$ occurs as the second component of at least one element of $BoxingType$; and
   (b) each element of $BoxingType$ has an element of $BoxedValueType$ as its second component.
2. Proof of 1(a) by contradiction.
3. Assume there is an element t of $BoxedValueType$ such that t is not the second component of at least one element of $BoxingType$.
4. From 3 and Theorem 4.3: $t \in AtomicNamedType$.
5. From 4, Z4.5, and Z4.4: there is one element d of $NamedType$ such that $d\ \underline{Atomic}\ t$.

6. From 5, 3, and Z4.68: $d \in \mathit{ValueType}$.
7. From 6, Z4.6, and Z4.7: there is one element u of $\mathit{UnboxedValueType}$ such that $d \ \underline{\mathit{Ubvt}}\ u$.
8. From 7 and Z4.6: $u \ \underline{\mathit{Ubvt}^{\sim}}\ d$.
9. From 8, 5, and Z4.71: $u \ \underline{\mathit{BoxingType}}\ t$.
10. 9 contradicts 3.
11. From 10: 1(a) is proven.
12. A similar argument proves 1(b).

QED

Theorem: 4.7
$\operatorname{ran} \mathit{Box} = \mathit{BoxedValueInstance}$

Proof

1. The proof is similar to the proof of Theorem 4.6.

QED

Theorem: 6.1
$\forall i : \mathit{ImplementsMethod} \bullet \mathit{second}(i) \in \mathit{VirtualMethod}$

Proof

1. From Z6.15: if $i \in \mathit{ImplementsMethod}$, then $\mathit{second}(i) \in \mathit{AbstractMethod}$.
2. From 1, Z3.27, Z3.32, Z3.35: if $i \in \mathit{ImplementsMethod}$, then $\mathit{second}(i) \in \mathit{VirtualMethod}$.

QED

Theorem: 9.1
$\mathit{Cls}02 = \varnothing$

Proof

1. Proof by contradiction.
2. Assume there is an element m of $\mathit{Cls}02$.
3. From 2 and Z9.1: $m \in \mathit{ExposedCompliantMember}$ and $\mathit{DefinedBy}(m) \in \mathit{NonCompliantNamedType}$.
4. From 3 and Z8.26: $m \in \mathit{CompliantMember}$.

5. From 4 and Z7.26: $DefinedBy(m) \in CompliantNamedType$.
6. From 5 and Z7.25: $\neg\ (DefinedBy(m) \in NonCompliantNamedType)$.
7. 6 contradicts 3.
8. 2 leads to a contradiction.

QED

Theorem: 9.2
$Cls03 = \varnothing$

Proof

1. Proof by contradiction.
2. Assume there is an element m of $Cls03$.
3. From 2 and Z9.2: $m \in ExposedCompliantType$ and $m \in BoxedValueType$.
4. 3 contradicts Z7.28.

QED

Theorem: 9.3
$Cls09 = \varnothing$

Proof

1. Proof by contradiction.
2. Assume there is an element t of $Cls09$.
3. From 2 and Z9.4: $(\exists\, v : EnumValueField;\ l : EnumLiteralField \bullet$
   $DefinedBy(v) = t \land$
   $DefinedBy(l) = t \land$
   $HasFieldSignature(v) \neq HasFieldSignature(l))$.
4. 3 contradicts Z6.7.

QED

Theorem: 9.4
$Cls17 = \varnothing$

Proof

1. Proof by contradiction.
2. Assume there is an element t of $Cls17$.

3. From 2 and Z9.11: $m \in \mathit{ExposedCompliantType}$ and $m \in \mathit{Unmanaged-PointerType}$.

4. 3 contradicts Z7.30.

QED

Appendix D

ECMA References

This Appendix discusses the relationship between this document and selected sections of ECMA.

I 6.3.1: Marking Items as CLS-Compliant

> *... This allows any item (assembly, type, or type member) to be explicitly marked as CLS-compliant or not.*
>
> *...*
>
> *By default, a type inherits the CLS-compliance attribute of its enclosing type (for nested types) or acquires the value attached to its assembly (for top-level types). It may be marked as either CLS-compliant or not CLS-Compliant by attaching the "`System.CLSCompliantAttribute`" attribute.*
>
> *By default, other members (methods, fields, properties, and events) inherit the CLS-compliance their type. They may be marked as not CLS-compliant by attaching the attribute "`System.CLSCompliantAttribute(false)`".*

This explicitly states that: types may be marked as CLS-compliant or not CLS-Compliant; and members may be marked as not CLS-compliant. We assume that the omission of "CLS-compliant" from the second item is deliberate, and that it implies that members can never be marked as CLS-compliant, in spite of the previous statement that "This allows any item ... to be explicitly marked as CLS-compliant or not."

I 7.5: Naming

> *Names are given to entities of the type system so that they can be referred to by other parts of the type system or by the implementations of the types. Types, fields, methods, properties and*

> *events have names. With respect to the type system values, locals, and parameters do not have names. An entity of the type system is given a single name, e.g. there is only one name for a type.*

Some types do not have names; for example, we assume that a function pointer type does not have a name (ECMA I 7.9: *implicit types need not have user-supplied names*). It might be possible to deem that a function pointer type has a name like *pointer to method taking Int32 and String and returning Boolean*, but we do not do so. Also, the virtual execution system may manufacture an arbitrary name.

I 7.5.2: Assemblies and Scoping

> *Generally, names are not unique. Names are collected into groupings called scopes. Within a scope, a name may refer to multiple entities so long as they are of different kinds (methods, fields, nested types, properties and events) or have different signatures.*
>
> *...*
>
> *A named entity has its name in exactly one scope. Hence, to identify a named entity, both a scope and a name need to be supplied. The scope is said to qualify the name. Types provide a scope for the names in the type; hence types qualify the names in the type. For example, consider a compound type Point that has a field named x. The name "field x" by itself does not uniquely identify the named field, but the qualified name "field x in type Point" does.*
>
> *Since types are named, the names of types are also grouped into scopes. To fully identify a type, the type name shall be qualified by the scope that includes the type name. Type names are scoped by the assembly that contains the implementation of the type. ...*
>
> *The CTS supports an enum ... an alternate name for an existing type.*

Type names are scoped by the assembly that contains the implementation of the type suggests that the assembly scopes all type names. However, the same Section earlier suggests that the names of nested types are scoped by the type that nests them, and ECMA I 7.5.3.4 confirms this interpretation. In our model, the name of a nested named type is scoped by the named type it is directly nested in.

See Appendix E.1 for a discussion of the importation of non-uniquely named types into an assembly.

An enum cannot be an alternate name for an existing type: for example, an enum has a value field, and an enum directly inherits C_E; if an emum were an alternate name for an existing type t, then t would also have a value field and inherit C_E.

I 7.5.3.4: Nested Types

> *A type (called a nested type) can be a member of an enclosing type. ... A nested type is part of the enclosing type ... The names of nested types are scoped by their enclosing type, not their assembly. ... There is no requirement that the names of nested types be unique within an assembly.*

We assume that the terminology *type* does not refer to every type. For example, we assume that a pointer type cannot be an enclosing type. We assume that an enclosing type can only be a named type, as declared in Section 2.3.

We assume that the terminology *enclosing* means "lexical enclosure in the source language"; it follows that there can be at most one enclosing named type, and the nesting graph must be acyclic. Note, however, that ECMA II 21.29, by using the word *typical*, allows the possibility of languages that define nesting by some mechanism other than lexical enclosure. ECMA II 21.29 (informative text) confirms that there can be at most one enclosing named type. It does not confirm that the nesting graph must be acyclic.

ECMA II 11 explicitly states that an interface may be nested in a class, value type, or interface. ECMA II 9.6 reiterates that an interface may be nested in a class or value type. ECMA II 9.6 prohibits a class or value type from being nested in an interface. We assume that a class and value type can be nested in a class or value type.

I 7.9.1: Array Types

> *An array type shall be defined by specifying the element type of the array, the rank (...) of the array, and the upper and lower bounds of each dimension of the array. ... The bounds ... shall be signed integers. While the actual bounds for each dimension are known at runtime, the signature may specify the informaton that is known at compile time: no bounds, a lower bound, or both an upper and lower bound.*
>
> *...*
>
> *Values of an array type are objects; hence an array type is a kind of object type.*

> ...
>
> *In general, array types are only distinguished by the type of their elements and their rank.*

This states that an array type includes full shape information: rank, upper bounds, lower bounds. It also states that array types are distinguished only by their element type and rank. (See Appendix D (II 13.2) for another apparently contradictory statement.) It also states that full shape information is known at runtime, but only partial information may be known at compile time. It seems to be saying, less clearly, that it is acceptable to perform only partial type checking at compile time. This should not be a surprise since it is known that, for one reason or another, full type checking at compile time may not be possible or practicable, unless the type checker is overly restrictive.

An array element cannot be: a typed reference (ECMA I 7.6.1.3); a managed pointer (ECMA II 13.4.2). ECMA does not prohibit an array element from being any other type.

I 7.9.2: Unmanaged Pointer Types

> *An unmanaged pointer type (also known simply as a "pointer type") ...*

ECMA I 7 (Figure 1) suggests that "pointer" is a generalization of managed pointer, unmanaged pointer, and function pointer. Our model adopts the connotation suggested by Figure 1.

I 7.9.6.2: Concreteness

> *An object type may be marked as abstract by the object type definition. An object type that is not marked abstract is by definition concrete. Only object types may be declared as abstract. ...*
>
> *It is an error to attempt to create an instance of an abstract object type ... .*

Although this paragraph appears to be restricted to a discussion of source code markings, it seems to constitute ECMA's definition of *abstract* and *concrete*. Therefore, it suggests that *abstract* and *concrete* are defined on classes and value types only. Our model differs from ECMA: it defines *abstract* on classes, value types, and interfaces; it subsequently constrains each interface to be an abstract named type and each value type to be a non-abstract named type (Section 2.10). We argue that, in spite of the apparent definition, ECMA actually defines *abstract* and *concrete* on classes, value

types, and interfaces: ECMA II 21.34 attaches a *type attributes flag* to each row in TypeDef, including rows that represent interfaces. ECMA II 21.34 item 23 specifies that *an Interface shall have Flags.Abstract = 1.*

From the viewpoint of a compiler, the definition of abstract object type appears to be circular: an abstract object type can be recognized, because it is marked; when an abstract object type is recognized, the compiler is required to mark it. See Appendix D (II 9.1.3) for a similar example. The solution is to acknowledge that *abstract object type* is defined outside the scope of ECMA. Languages identify an abstract object type by a keyword, or the absence of at least one method implementation, or some other syntax construct.

Our model takes an approach based only on the existence of exact instances (Section 2.4). Both approaches are functionally equivalent, since ECMA prohibits an abstract object type from having any exact instance, and we assume compilers and other software artefacts enforce this prohibition, and *Object* contains all objects that could potentially exist.

I 7.9.9: Object Type Inheritance

> *...*
>
> *An object type declares it shall not be used as a base type (be inherited from) by declaring that it is a sealed type.*

Although this paragraph appears to be restricted to a discussion of source code markings, it seems to constitute ECMA's definition of *sealed*. Therefore, it suggests that *sealed* is defined on classes and value types only. Our model differs from ECMA: it defines *sealed* on classes, value types, and interfaces; it subsequently prohibits each interface from being a sealed named type, and requires each value type to be a sealed named type (Section 2.10). We argue that, in spite of the apparent definition, ECMA actually defines *sealed* on classes, value types, and interfaces: ECMA II 21.34 attaches a *type attributes flag* to each row in TypeDef, including rows that represent interfaces. ECMA II 21.34 item 35 specifies that *a ValueType shall be sealed*. Since ECMA I 7.9.9 does not define *sealed* on interfaces, this implies that the sealed flag, although it always exists, is zero for TypeDef rows representing interfaces. ECMA II 21.34 item 27 (*if Flags.Interface == 1 then Flags.Sealed shall be zero*) confirms that ECMA defines *sealed* on interfaces.

I 7.11.3: PropertyDefinitions

> *... While all of the attributes of a member may be applied to a property (accessibility, static, etc.) these are not enforced by the CTS. ...*

Since the Property meta-data table (ECMA II 21.31, II 22.1.13) does not define flags for all the attributes of a member, how can we apply all the member attributes to a property? Language syntax may apply member attributes to a property, but there does not seem to be any way to represent them in meta-data.

I 11.1.1.2: Managed Pointer Types: O and &

> *The O datatype represents an object reference ... .*
>
> *The & datatype (managed pointer) is similar to the O type, but points to the interior of an object. That is, a managed pointer is allowed to point to a field within an object or an element within an array ... .*
>
> *...*
>
> *In order to allow managed pointers to be used more flexibly, they are also permitted to point to areas that aren't under the control of the CLI garbage collector, such as the evaluation stack, static variables, and unmanaged memory. ...*

This specifies *where* a managed pointer can point, but not *what* it can point to, although *what* is obviously restricted by the types that can be stored in *where*. ECMA II 13.4.2 specifies both *where* a managed pointer can point and *what* it can point to.

II 6: Assemblies, Manifests and Modules

> *While some programming languages introduce the concept of a namespace, there is no support in the CLI for this concept. Type names are always specified by their full name relative to the assembly in which they are defined.*

Although ECMA II 6 specifies that there is no concept of *namespace*, ECMA II 21.14, II 21.34, and II 21.35 use the term *namespace* extensively. We adopt the interpretation in ECMA II 21.14: *namespace* is a convention allowing a named type name to be split into two pieces at an included ".". Our model does not recognize the notion of namespace and does not model any means of splitting a named type name.

II 9.1.3: Type Semantics Attributes

> *The type semantic attributes specify whether an interface, class, or value type shall be defined. The interface attribute specifies*

an interface. If this attribute is not present and the definition extends (directly or indirectly) System.ValueType a value type shall be defined. Otherwise, a class shall be defined.

This specifies how to determine whether a given named type is a class, a value type, or an interface: if the interface attribute is set, then the named type is an interface; else if the named type inherits C_V then it is a value type; else it is a class.

According to ECMA II 9.1.3, C_E is a value type. This causes contradictions in ECMA II 12 (*Value types shall be sealed, ...*); and ECMA II 21.34 (*Regular Value Types (i.e., excluding Enums - see later) are defined as deriving directly from System.ValueType. Regular Value Types cannot be derived to a depth of more than one.*); and possibly elsewhere. Therefore our model explicitly introduces C_E as a class (Section 2.10).

From the viewpoint of a compiler, the definition of interface is circular: an interface can be recognized, because the interface attribute is set; when an interface is recognized, the compiler is required to set the interface attribute. The solution is to acknowledge that *interface* is defined outside the scope of ECMA. Languages identify an interface by a keyword. Since language syntax is outside the scope of ECMA, *interface* is an undefined term from ECMA's viewpoint. Once a compiler has set the interface attribute other tools may use it, but the definition of interface should not rely on the interface attribute.

In our model the inheritance graph is undefined, and class, value type, and interface are defined in terms of it (Section 2.10). This approach is equivalent to ECMA, since compilers recognize an interface from a language keyword, and they prohibit an interface from inheriting C_J. Our model introduces *DirectlyInherits* as a relation from *CONVENTIONAL_CLASS* to itself (Section 2.2.1), but it does not provide any means to determine whether a particular pair of conventional classes is an element of the relation. That is, the elements of *DirectlyInherits* are taken to be given: they are determined by language syntax outside the scope of both our model and ECMA.

II 13.2: Arrays

The type of an array (other than a vector) shall be determined by the type of its elements and the number of dimensions.

This clearly states that an array type does not include full shape information; it includes only rank. See Appendix D (I 7.9.1) for an apparently contradictory statement. Our model assumes that array type includes full shape information.

II 13.3: Enums

> *... The symbols of an enum are represented by an underlying type: one of bool, char, ..., float32, float64.*

ECMA I 7.5.2 (*The underlying type shall be a built-in integer type*) and ECMA II 21.34 item 33 (*shall be at least one instance field, of integral type*) contradict ECMA II 13.3. We assume ECMA I 7.5.2 and II 21.34 take precedence.

II 13.4.2: Managed Pointers

ECMA II 13.4.2 specifies (in informative text) at least three kinds of information about managed pointers: what types they can point to (e.g., object references and value types, but not managed pointers); where they can point (e.g., a local variable or a field location of an object, but presumably not to any place not explicitly mentioned); where they can be stored (e.g., local variables and arguments, but not static variables). See also Appendix D (I 11.1.1.2).

The following Sections summarize the information gleaned from ECMA. The language and terminology are informal and closely reflect ECMA's usage, rather than the usage of our model.

Where It Can Point

A managed pointer can point to any of the following locations: a field location of an object (ECMA I 7.2.4, I 11.1.1.2, II 13.4.2); a field location of a value type (ECMA I 7.6.1.3, II 13.4.2); an array element or one position past the end of an array (ECMA I 7.6.1.3, I 11.1.1.2, II 13.4.2); a local variable (ECMA I 7.6.1.3, II 13.4.2); a parameter location (ECMA I 7.6.1.3, II 13.4.2).

What It Can Point To

A managed pointer can point to an instance of: a value type (ECMA I 7.2.4, II 13.4.2, II 14.2); an object reference type (ECMA II 13.4.2).

It cannot point to an instance of: a managed pointer type(ECMA II 7.1, II 13.4.2); a typeref (ECMA II 7.1); a null instance (ECMA II 13.4.2).

ECMA does not explicitly restrict a managed pointer from pointing to an instance of an unmanaged pointer type or a function pointer type; however, we assume it intended to.

Where It Can Be Stored

A managed pointer can be stored in: a return value (ECMA II 13.4.2); a local variable (ECMA II 13.4.2); an argument (parameter?) (ECMA II 13.4.2).

It cannot be stored in: a field location (ECMA II 13.4.2); an array element (ECMA II 13.4.2); a static variable (ECMA II 13.4.2).

II 21.26: MethodSemantics

> *For each property, there shall be a setter, or a getter, or both. [CLS]*

This CLS requirement is not listed in the Collected CLS Rules.

II 21.34: TypeDef

> *Interfaces do not inherit from one another, however, they specify zero or more other interfaces which shall be implemented.*
>
> *Note that any type shall be one, and only one, of:*
>
> - *Class (Flags.Interface = 0, and derives ultimately from System.Object)*
> - *Interface (Flags.Interface = 1)*
> - *Value type, derived ultimately from System.ValueType*
>
> *For any given type, there are two separate, and quite distinct inheritance chains of pointers to other types (the pointers are actually implemented as indexes into metadata tables). The two chains are:*
>
> - *Extension chain — defined via the Extends column of the TypeDef table. Typically, a derived Class extends a base Class (always one, and only one, base Class).*
> - *Interface chains — defined via the InterfaceImpl table. Typically, a Class implements zero, one or more Interfaces.*

According to the diagram, an enum cannot implement an interface; nor can C_V and C_E.

II 21.34 Item 33: TypeDef

> *The Name string of the instance field shall be "value__"; it shall marked RTSpecialName; its type shall be one of ... : ELEMENT_TYPE_U1, ELEMENT_TYPE_I2, ELEMENT_TYPE_I4, ELEMENT_TYPE_I8.*
>
> *[There] shall be no other members (ie, apart from any static literals, and the one instance field called "value__") [CLS]*

This CLS requirement is not listed in the Collected CLS Rules.

II 22.2.12: Type

ECMA I 7 (Common Type System) does not define void type. However, ECMA II 22.2.11 and II 22.2.12 require a definition of void type, since an unmanaged pointer may be a pointer to a void type, and a return type may be a void type.

Appendix E

Unresolved Questions

E.1 Importing Non-uniquely Named Types

Can an assembly contain a named type whose name is n, and import another named type whose name is also n? The options are:

- Implementations are required to permit the importation of non-uniquely named types.
- Implementations are required to prohibit the importation of non-uniquely named types.
- Implementations are allowed to permit or prohibit the importation of non-uniquely named types as they see fit.

ECMA I 7.5.2 seems to specify the first option, as follows. Since a named entity has its name in one scope, a named type name must be in the scope of the assembly that contains the named type, not in the scope of the assemblies that import the named type. Hence, ECMA allows an assembly to import named types whose names are the same as the names of other named types contained by or imported into the assembly. According to ECMA I 7.5.2, named types with non-unique names can be identified by qualifying them with the name of the assembly they are contained by. Further, ECMA nowhere mentions prohibiting the importation of named types with non-unique names.

However, the Microsoft Visual Studio implementation of .NET seems to prohibit the importation of named types whose names are the same as the names of other named types contained by or imported into the assembly. According to both our model and ECMA, a namespace name, if it exists, is not an assembly qualifier; it is part of a named type name (Section 2.2.1, Appendix D (II 6)).

Our model, like ECMA, does not prohibit the importation of named types with non-unique names. Since their importation is allowed, developers

may construct assemblies that import them, and therefore implementations are required to permit their importation. Is this correct?

E.2 Inheriting CLS Compliant and CLS Non-Compliant Attributes

ECMA I 6.3.1 describes a mechanism by which assemblies, named types, and members can have a custom attribute marking them CLS compliant or CLS non-compliant; or can acquire a marking from the entity that contains, nests, or defines them. Top level named types acquire from the assembly they are contained by; nested named types acquire from the named type they are directly nested in; members acquire from the named type they are defined by. In addition, ECMA allows classes, interfaces, and overridden members to inherit custom attributes via the inheritance mechanism.

The *Inherited* property of *System.AttributeUsageAttribute* may be used to inhibit the inheritance of a custom attribute by derived classes, interfaces, and overridden members. Inheritance of the CLS compliant attribute is permitted. Therefore a named type or member could acquire conflicting CLS compliant attributes via multiple paths. ECMA does not specify which path takes precedence. However, our model assumes that ECMA intended to inhibit its inheritance.

Is it correct to assume that the inheritance of a CLS compliant attribute should be inhibited?

E.3 Increasing Accessibility in Nested Named Types

Does ECMA prohibit a nested named type from having an accessibility greater than the accessibility of the nested named type it is nested in? Does ECMA prohibit a member from having an accessibility greater than the accessibility of the nested named type it is defined by?

Our model interprets ECMA as follows. ECMA does not prohibit a nested named type from having an accessibility greater than the accessibility of the nested named type it is nested in. ECMA does not prohibit a member from having an accessibility greater than the accessibility of the nested named type it is defined by. ECMA I 7.5.3.4 “Nested Types” says: *A nested type ... has an accessibility as would any other member of the enclosing type.* ECMA I 7.5.3.2 “Accessibility of Members” says: *In general, a member of a type can have any one of these accessibility rules assigned to it. There are two exceptions, however: (1) Members defined by an interface shall be public. (2) When a type defines a virtual method that overrides an inherited definition, ...* . The access rules in ECMA I 7.5.3 “Visibility, Accessibility, and Security” do not appear to prohibit access to such a member or nested named type: *Access to a member of a type is permitted only if all*

three of the following conditions are met: The type is visible. The member is accessible. All relevant security demands ... have been granted. Note that the rule for accessing a member does not require that its type be accessible; only that it be visible. For example, a member having at least family accessibility and defined by a nested named type having public visibility but less than family accessibility is exposed outside its assembly.

Are these interpretations correct?

E.4 Applicability of CLS Rules Only to Exposed Items

ECMA I 6.3 and ECMA I 10 specify that CLS rules apply only to items that are visible outside their defining assembly and have public, family, or family-or-assembly accessibility. That is, they do not apply to items that are marked CLS compliant but are not exposed.

Does this make sense? It would make sense if there were a rule stating that a non-exposed item cannot be marked compliant or non-compliant; (however, see below). However, consider the following. It is obviously desirable that compilers should be able to perform compliance checking. Therefore, all factors affecting CLS compliance should be known at compile time. Visibility affects exposure, which affects CLS compliance. According to ECMA I 7.5.3.1, "it is the configuration of the assembly that decides whether the type name is made available." Since visibility is determined after compilation, not all factors affecting CLS compliance are known at compile time. For the same reason, it would not make sense to state that a non-exposed item cannot be marked compliant or non-compliant, as suggested above.

In the above discussion we distinguish between compile-time visibility (as marked in the named type's source code) and configuration-time visibility (as specified by the assembly configurator). We assume that configuration-time visibility cannot be greater than compile-time visibility. In our model, exposure is determined by the configuration-time visibility.

Either of the following options would make sense.

- Specify that CLS compliance depends only on compile-time visibility. Then, assuming visibility cannot be increased at configuration time, CLS errors (or warnings) detected by the compiler would be conservative.

- Remove visibility and possibly also accessibility from the statement of applicability of CLS rules. That is, CLS rules would apply to marked or acquiring CLS compliant, regardless of their visibility (and possibly accessibility). It would be valid to mark a named type as CLS compliant even if it is not visible. Later, its visibility may change without

any requirement to check whether it or its members break any CLS rule. Then, the CLS complaint attribute becomes an indicator that the programmer uses to request the compiler to verify CLS compliance of an item.

E.5 Hide by Signature

ECMA I 7.10.4 says *If the member is marked hide by name-and-signature then only a member of the same kind with exactly the same name and type (for fields) or method signature (for methods) is hidden in the derived class.*

ECMA II 8.3 says *hide-by-name-and-sig [means] that the introduction of a name in a given type hides any inherited member of the same kind but with precisely the same type (for fields) or signature (for methods, properties, and events).*

However, ECMA II 22.1.4, II 22.1.4, and II 22.1.13 do not provide a flag to mark fields, events, and properties as *hide by name-and-signature.* Further, ECMA II 21.24, item 23 says *If any of ... HideBySig are set in Flags, then Flags.Virtual shall also be set.* Since *virtual* is not defined on fields, events, and properties, it is difficult to reconcile this requirement with the assumption that *HideBySig* is defined on fields, events, and properties.

We assume that *HideBySig* is not defined on fields, events, and properties. Is this correct?

E.6 Is Rule 12 a CLS Rule?

Surely Rule 12 applies to all members and types, whether or not CLS?

E.7 Can an Unmanaged Pointer Point to Heap?

Can an unmanaged pointer point to a location in the garbage collected heap? Can it point to an object instance? Can it point to a field location of an object instance? Can it point to an unboxed value instance stored in a field location of an object instance? Can it point to an object reference instance stored in a field location of an object instance? Can it point to a managed pointer instance stored in a field location of an object instance? Can it point to an unmanaged pointer instance stored in a field of an object instance? Can it point to a function pointer instance stored in a field location of an object instance? Can it point to an array instance stored in a field location of an object instance?

ECMA II 13.4 *Pointers may contain the address of a field (of an object or value type) or an element of an array.* This refers to both managed and unmanaged pointers. It permits a pointer to point to a field location of an object, and hence into the garbage collected heap. Is this conclusion correct?

ECMA I 11.3.2.1 says *A managed pointer (type &) may be explicitly converted to an unmanaged pointer (type native unsigned int), although this is not verifiable and may produce a runtime exception.* Since a managed pointer could point to an object instance or the interior of an object instance, it follows that an unmanaged pointer also can. Is this conclusion correct?

ECMA II 7.1.2: *If unmanaged pointers are used to dereference managed objects, these objects shall be pinned.* This seems conclusive: "yes".

Our model assumes the answer to all above questions is "yes". Is this correct?

E.8 CLS Compliance of Accessors

Must a property and its accessors all be CLS compliant or all non-compliant? ECMA does not specify. We do not specify any constraints relating the CLS compliance of properties and their accessors; or events and their accessors.

E.9 Compliance, Visibility, and Accessibility of General Types

ECMA does not explicitly define compliance, visibility, and accessibility on general types, for example, on managed pointer types or function pointer types. Nevertheless, several CLS rules, for example Rules 11 and 12, require a specification of compliance, visibility, and accessibility on a general type.

Each elementary type has a compliance, visibility, and accessibility derived from its related named type (Sections 7.3, 8.1, 8.2).

(Note that Rule 12 constrains types in the signature of an [exposed compliant] member to be visible and accessible whenever the member itself is visible and accessible, but there is no similar constraint on non-exposed or non-compliant members.)

The following sections discuss the issues involved in defining the compliance, visibility, and accessibility of any type based on the compliance, visibility, and accessibility of the elementary types it is built from.

Are our assumptions correct?

E.9.1 Compliance

We assume that a non-compliant type is a non-compliant elementary type, or a type built ultimately from one or more non-compliant elementary types, or a function pointer type built from a non-compliant member, or a type built ultimately from one or more non-compliant function pointer types (Section 7.3).

E.9.2 Visibility

A non-elementary type t may be built (ultimately) from multiple elementary types t_1, t_2, ... For example, t may be a function pointer type to a method taking a parameter of type t_1 and returning type t_2. t_1, t_2, ... may have different visibilities. ECMA provides little guidance to suggest how to define the visibility of t. Nevertheless, CLS Rule 12 requires a definition of visibility on a general type. We adopt the obvious choice: we define t's visibility as the "minimum" visibility of t_1, t_2, ... (Section 8.1).

E.9.3 Accessibility

A non-elementary type t may be built (ultimately) from multiple elementary types t_1, t_2, ... t_1, t_2, ... may have different accessibilities. ECMA provides little guidance to suggest how to define the accessibility of t. Nevertheless, CLS Rule 12 requires a definition of accessibility on a general type. We adopt the obvious choice: we define t's accessibility as the "minimum" accessibility of t_1, t_2, ... (Section 8.2).

Bibliography

[1] The Unicode Consortium. *The Unicode Standard Version 3.0*. Addison-Wesley, Reading, MA, 2000.

[2] Wendy Johnston. *A Type Checker for Object-Z*. SVRC, University of Queensland, Qld, Australia, 1996.

[3] Alf Smith. On Recursive Free Types in Z. In J.E. Nicholls, editor, *Z User Workshop*, pages 3–39. Springer-Verlag, 1991.

[4] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Hertfordshire, UK, second edition, 1992.