Towards an Object-Oriented Refinement Calculus

Jamie Shield

October 16, 2001

Contents

Abstract

This thesis is a step towards a programming method which supports a calculational style and is useful for the production of large software systems.

Object-orientation is a popular programming paradigm for the development of large scale systems. It is based on the idea that software should be a set of communicating entities, of which many entities are models of real objects.

The refinement calculus is a notation and set of rules that supports a calculational approach to the formal development of software.

It is envisaged that when the object-oriented paradigm and refinement calculus are fused, the resulting method would provide a development path for large scale, provably correct software.

Several attempts have been made to construct an object-oriented refinement calculus. This thesis discusses several of these and summarises the remainder. The approaches include the modular reasoning work of Utting and Robinson [UR92], the type theoretic approaches of Mikhajlova and Sekerinski [MS97] and the higher-order approach of Calvanti and Naumann [CN99b].

The thesis utilises a typed object calculus as the infrastructure of an alternative approach to the development of an object-oriented refinement calculus. Object calculi are a notation and set of rules that are used to model the concepts of object-orientation. They have been used as the basis of several object-oriented languages.

Both a semantics of object values and a semantics of references for an object-oriented refinement calculus are presented within the thesis. Additionally, both object-based and class-based object-oriented refinement calculi are supported.

A class refinement relation is introduced that supports the concept of modular reasoning. The subclass relation is restricted so that instances of subclasses are behaviourally compatible. A subclass may introduce additional attributes, yet these must adhere to the invariant and history properties of the superclass.

The development of programs that contain references can be tedious. A novel development method, termed coalescing, is introduced to allow unnecessary aliasing to be temporarily removed. Another novel technique is introduced to allow the development of reference semantics programs using a simpler value semantics.

Examples are presented which illustrate the capabilities of program development using an object-oriented refinement calculus. One example presents the application of the iterative, incremental development process of the object-oriented paradigm within the refinement calculus. Another example illustrates the formal use of design patterns.

List of Figures

Chapter 1

Introduction

This thesis is a step towards a programming method in which designs and programs can be calculated from specifications and which supports the production of large software systems. The development of provably correct programs can be reliably accomplished for small programs via such methods as the "Refinement Calculus" [Mor94, MV94, BvW98]. The refinement calculus is a wide spectrum language with a set of correctness preserving rules. The language is termed wide spectrum as it extends Dijkstra's [Dij76] guarded command language (GCL) with a non-executable 'specification' statement.

A program development in the refinement calculus is initiated by writing a specification statement which dictates the assumptions and desired effect of the program. Using rules provided by the refinement calculus, these specifications are systematically converted (or refined), with the aid of developer intuition, into the executable subset of the wide spectrum language.

Unfortunately, the scalability of the refinement calculus is limited. Various efforts have been made to develop extensions to overcome the scalability constraints using approaches traditionally used to modularise (informal) program development: modules [Mor94], ADTs [Ban97], and object-orientation [BMvW97, CN00, MS97, UR92]. This thesis focuses primarily upon the object-oriented approach, although the other approaches are not independent and many results of this thesis, especially the work on references, are pertinent to other approaches.

Object-orientation is a programming paradigm centered around the idea that software should be a set of communicating objects, with each object encapsulating data plus the methods for manipulating that data. Many of these objects are abstractions of real objects. Object-oriented development, when performed correctly, results in significant increases in code reuse and reliability. A thorough review of the object-oriented approach is covered by Meyer [Mey97].

The motivation for fusing the object-oriented paradigm and the refinement calculus is that it is envisaged that the resulting method would provide a development path for the calculation of large scale, provably correct software. Lano and Haughton [LH94] acknowledge that the application of formal specifications to large practical systems requires

powerful system-structuring mechanisms: "a need which object orientation addresses admirably." They suggest that an object-oriented language is not sufficient for the successful application of object-oriented techniques and that a disciplined approach is also required. The amalgamation of a refinement calculus with object-orientation would produce such a disciplined method.

Despite previous work, a practical object-oriented refinement calculus does not yet exist. Researchers have, so far, been primarily concerned with the modelling of the core subtyping/dynamic dispatch features, with little attention being paid to the population of such calculi with practicalities. This thesis summarises the approaches previously considered, and presents an alternative approach, building on an object calculus which handles much of the required object-oriented infrastructure.

Object calculi have been extensively studied by Abadi and Cardelli [AC96] in an effort to bridge the gap between existing types and those required for object-oriented languages[1]. Object calculi are analogous to the λ-calculus [Bav81] except that they manipulate objects rather than functions. The study of object calculi has been instrumental in separating the concepts of inheritance and subtyping [CHC94]. The development of future object-oriented languages should benefit markedly from the utilisation of the key concepts of object calculi in their semantics.

This thesis uses an object calculus as the infrastructure of both an object-based [SLU88] and class-based refinement calculus. This basis provides an abstraction of objects. Current formulations of object-oriented refinement calculi expend significant effort dealing with the object-oriented semantics, rather than concentrating on the aspects of refinement. Goldsack and Kent [GK96, §2.3] identify that "the problems of combining object-orientation and formal specification are due to the programming language origin of many of the concepts of object-orientation." For instance, object-oriented refinement has a close relationship with type compatibility [LH94, p78] [GK96, §2.4]. That is, if class *A* is a refinement of class *B*, then it should be possible to use *A* as an instance of *B*. As will be seen in Chapter 4, the direct support of object-oriented fundamentals in the object calculus (such as the determination of types and subtypes) allows for an abstract, concise expression of refinement relations. This indicates that an abstraction of objects simplifies the development of an object-oriented refinement calculus. Using an object calculus as an abstraction of objects must thus be a step in the right direction.

Goldsack and Kent [GK96] suggest that object-oriented structuring mechanisms can complicate reasoning about specifications. For instance, polymorphism makes it difficult to control which version of a method is actually executed, and the use of inheritance means that a method definition is fragmented across many classes. The integration of the object-oriented programming paradigm with the refinement calculus negates these concerns. Given a specification of a class *A*, for class *B* to be a subclass of *A*, each method

[1] Palsberg and Schwartzbach [PS94, s2.6] discuss this gap and argue that its cause is due to the origins of type theory as a discipline of logic.

in the subclass must be shown to be an incremental 'improvement', or refinement, of the corresponding method in the superclass. Consequently, it is irrelevant as to which method is invoked as all method variations conform to the original specification.

This thesis illustrates the use of several new development techniques. In particular, Chapter 6 introduces several techniques for easing the development of programs with a reference semantics. Additionally, the use of design patterns [GHJV96, LBG96] in the role of guiding formal, provable program transformations is illustrated in Chapter 7.

This thesis is organised as follows. In Chapter 2 the preliminary concepts of object-orientation, object calculi and refinement calculi are introduced. Chapter 3 presents a summary of the existing approaches to object-oriented refinement calculi. In Chapter 4 a wide-spectrum, object-oriented language is developed. The notions of refinement and data-refinement are presented in the language. To the author's knowledge, this is the first presentation of a refinement relation that deals with the refinement of heterogeneously typed statements. Such refinements are useful when replacing an object with a sub-object within a client program. Heterogeneously typed refinement allows the client to use the new attributes of the sub-object. A reference semantics and associated language constructs are also provided.

Chapter 5 introduces the notion of object-refinement—essentially an application of statement refinement to the method components of an object. In general, without restriction, the substitution of an object o in a program with an object-refinement is not a refinement as predicates such as $o = c$ for some constant c are not upheld by the object-refinement. To achieve the desired object-refinement monotonicity in statements, predicates are restricted to those monotonic in object-refinement. This constraint has minor practical significance. The object-refinement relation is then generalised to an object-data-refinement relation. Finally, class-based extensions of these notions are provided.

Specialised concepts and techniques for developing reference semantics programs are presented in Chapter 6. In particular, a novel technique which supports the development of reference semantics programs using a simpler value semantics is introduced. Also a novel technique that removes unnecessary aliasing is presented.

Chapter 7 uses several novel techniques and examples of the use of an object-oriented refinement calculus.

Appendix A provides a reference of additional laws, definitions and theorems. Appendix B presents the proofs of the theorems listed in the thesis. Appendix C provides a glossary for the symbol syntax. Finally, Appendix D provides concise definitions of the terminology used. It also acts as an index.

Chapter 2

Background

There is a diversity of terminologies used for object-oriented concepts. This chapter establishes the terminology and background concepts used within this thesis. This includes several core aspects of object-orientation and how these aspects can be grounded in a theory, namely the object calculus. Finally, the foundations of refinement calculi are presented. Figure 2.1 illustrates how the preliminary material presented in this chapter is used in subsequent chapters.

2.1 Object-Orientation

A review of object-oriented concepts is presented in this section. Meyer [Mey97] and Abadi and Cardelli [AC96] present introductions to the object-oriented programming paradigm.

The complexity of the problems that computers solve is rising. This increasing complexity drives the development of new generations of languages as it becomes cumbersome to solve new problems with current languages. With each new language generation the supported language constructs become more abstract. The transition from languages that support abstract data structures through to object-oriented languages can be viewed in this context. Module-oriented languages offer encapsulation through abstract data structures (ADS). ADSs contain data and methods for manipulating this data. When an ADS is formulated, a single instance (or copy) is constructed. To acquire another copy another ADS must be formulated. For example, to model a car with four wheels, four ADSs would be required to model the wheels. The recognition of this inefficiency motivated the formation of the abstract data type (*ADT*). An ADT contains data type information and methods for manipulating instances of the data type. With the addition of the concepts of inheritance, polymorphism and dynamic dispatch, object-oriented languages were introduced.

Object-Orientation "Object-oriented software construction is the building of software systems as structured collections of possibly partial abstract data type implementations"

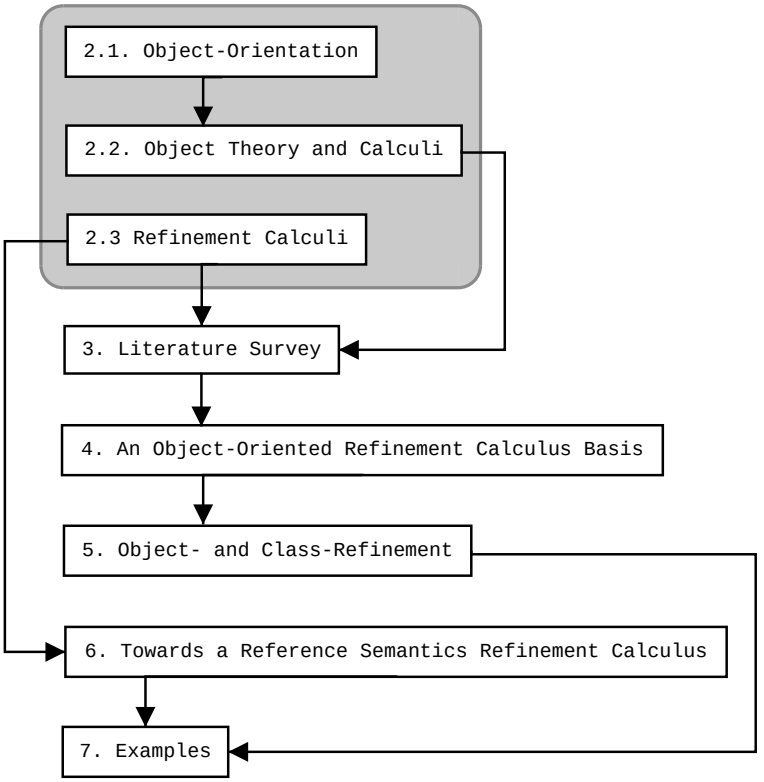

Figure 2.1: Reading Map

[Mey97, p147]. This 'working' definition, however, concentrates on the language of object-orientation. The techniques for developing object-oriented programs, however, are just as, if not more, important than the language. A proficient programmer can develop modularised code in a non-module-oriented language: the language merely serves to encourage good programming practice. Work on object-oriented programming reasoning notations and methods (such as UML [JBR99, PJ99] and Fusion [CAB+94]), heuristics [Rie96], and design patterns [GHJV96, Pre95] highlight the desire to capture and standardise object-oriented programming techniques. An object-oriented refinement calculus could be regarded in a similar vein: the formalisation of object-oriented techniques.

Objects If object-orientation deals with implementations of ADTs, then objects are instantiations of ADTs. The term *fields* is used to denote the data portions of an object and the term *methods* is used to denote the procedural portions. *Attributes* refers to the combination of both. An attribute's *host* is the object in which it is contained.

Classes A *class* can be viewed as a set of object instances or alternatively, as a template or blue-print, describing the construction of each object of the class. In addition, it can be considered a type, like an ADT, providing a service through a list of suitably formulated operations [GK96].

Subclasses A *subclass* is a class constructed incrementally using another class. A class *C* can be extended with new attributes, or existing attributes can be overwritten to form a subclass (*D*). Class *C* is termed the *superclass* of *D*.

Inheritance is the sharing of attributes between a class and its subclass. Multiple inheritance occurs when a subclass inherits attributes from more than one class. Single inheritance occurs when a subclass inherits attributes from only one class.

Polymorphism Given classes *C* and *D* where *D* is a subclass of *C*, then an instance of class *D* (*d*) may be used as an instance of class *C*. This is referred to as (subtyping) *polymorphism*.

Subtyping Traditionally, the polymorphic reuse of objects is restricted to situations of subclassing. There are occasions, however, when it is desirable to use an object (*d*) as an instance of a differing class (*C*) even though its class (*D*) may not be a subclass (of *C*). *Syntactic subtyping* is introduced for this reason. It allows *polymorphic* reuse of objects regardless of whether their respective classes exist in a subclass relationship. This increase in flexibility is part of the approach known as inheritance-is-not-subtyping [CHC94].

An object's (and class') type is determined by a list of the names of its attributes and their types (types of its fields and the parameters of its methods). One possible definition of the subtyping relationship is that subtypes may introduce new attributes while maintaining the old. With this definition, a subtype has at least the attributes of its supertype. For object types γ and δ, if δ has at least the attributes of γ, then δ is a subtype of the supertype γ:

$$\delta \preceq \gamma$$

Consequently, given a class *C* with type γ, and another class *D* of type δ, then code written to use an instance of class *C* would also operate on an instance of class *D*. The effect of the code may be substantially different despite this syntactic compliance.

Subsumption Given types γ and δ where $\delta \preceq \gamma$, and an object *d* of type δ, that is $d : \delta$, then the recognition that *d* is also of type γ is known as *subsumption*.

$$d : \delta \wedge \delta \preceq \gamma \Rrightarrow d : \gamma$$

Dynamic Dispatch Given class *C* with method *m* and subclass *D*, when method *m* is invoked on an instance *d* of class *D* (and by subsumption, also an instance of *C*), it is *D*'s version of the method *m* that is executed, not *C*'s. This is termed *dynamic dispatch*. It has several synonyms including dynamic binding and runtime method discrimination.

Subclassing-is-not-Subtyping As discussed above, allowing subtyping in the absence of subclassing provides more opportunities for reuse of objects. The alternative should also be considered: subclassing without subtyping. The following example, adapted from one of Abadi and Cardelli's [AC96, p31], shows that subclassing does not necessarily induce subtyping.

Example 2.1 (Subclassing-is-not-Subtyping) The class *maxClass* has a field *n* of type $\mathbb{Z}$ and a method *max* which takes another instance of class *maxClass*, and returns the object with the largest *n* field. The syntax used in this example is informal.

class *maxClass* **is**
  var $n : \mathbb{Z} := 0$;
  method $max(other$: **Self** $)$: **Self** **is**
    if (**self**.$n > other.n$) **then return self else return** *other* **end**
  end
end

The subclass *minMaxClass* includes an additional method for returning the object with the minimum *n*.

subclass *minMaxClass* **of** *maxClass* **is**
  method $min(other$: **Self** $)$: **Self** **is**
    if (**self**.$n < other.n$) **then return self else return** *other* **end**
  end
end

The object type of the first class *maxClass* is:

ObjectType *MaxType* **is**
  var $n : \mathbb{Z}$;
  method $max(other$: *MaxType*$)$: *MaxType*
end

Correspondingly, the object type of class *minMaxClass* is

ObjectType *MinMaxType* **is**
  var $n : \mathbb{Z}$;
  method $max(other$: *MinMaxType*$)$: *MinMaxType*
  method $min(other$: *MinMaxType*$)$: *MinMaxType*
end

Under the subtyping definition these types are not related as the method parameter types are not equivalent[1].

[1] This constraint is relaxed later.

Now consider a subclass of *minMaxClass*, namely *newMinMaxClass*. It overrides the *max* method to use the minimum method.

subclass *newMinMaxClass* **of** *minMaxClass* **is**
    override $max(other : \textbf{Self}) : \textbf{Self}$ **is**
        if $(other.min(\textbf{self}) = other)$ **then return self else return** *other* **end**
    end
end

An instance *mm* of *newMinMaxClass* has the type *newMinMaxType*.

ObjectType *newMinMaxType* **is**
    var $n : \mathbb{Z}$;
    method $max(other : newMinMaxType) : newMinMaxType$;
    method $min(other : newMinMaxType) : newMinMaxType$
end

Since the label *newMinMaxType* acts like a bounded quantifier it can be replaced with another label, e.g., *MinMaxType*. *MinMaxType* and *newMinMaxType* are therefore the same type. Consequently the instance *mm* of *newMinMaxClass* has type *MinMaxType*.

$mm : MinMaxType$

This is referred to as *structural type equivalence*. *Name type equivalence* also requires the names to be equivalent.

Assuming the argument that subclassing implies subtyping means that subtyping should hold between *MinMaxType* and *MaxType*.

$$(minMaxClass \ \textbf{subclass} \ maxClass) \Rrightarrow MinMaxType \preceq MaxType$$

Consequently, through subsumption, $mm : MaxType$.

$$(mm : MinMaxType \wedge MinMaxType \preceq MaxType) \Rrightarrow mm : MaxType$$

Given object instance *m* of *maxClass*, then the call $mm.max(m)$ would produce a dynamic error as *m* does not possess a *min* method.

$mm.max(m)$
$\equiv$ Argument substitution.
if $(m.min(mm) = m)$ **then return** *mm* **else return** *m* **end**

Given this contradiction, it must be deduced that subclassing does not always imply subtyping. The reader is referred to the work of Cook, Hill and Canning [CHC94] for a thorough discussion of this topic.
$\diamond$

Object-based The object-based approach [SLU88] supports the concept of objects but not inheritance [GK96, p7]. Object-based languages are intended to be both simpler and more flexible than class-based languages. They typically use special objects termed *prototypes* as the templates for the creation of object instances. *Cloning* is the term that refers to the process of copying an object.

2.2 Object Theory and Calculi

The λ-*calculus* [Bav81] is a theory of functions which provides the foundations for procedural languages. *Object calculi*, analogously, are used to provide the foundations for object-oriented languages. Object calculi reduce object-oriented notions such as subtyping and inheritance to a few basic concepts. Abadi and Cardelli [AC96] provide a collection of such object-oriented calculi. Their book advances through the calculi by presenting a calculus, showing its usefulness, and using the deficiencies of one calculus to motivate the next, more sophisticated (in its typing mechanisms) calculus. The book also explores the topic of a denotational semantics for the object calculi, and various techniques for defining and axiomatising the 'Self' type of an object. Of interest for this research is their discussion of interpreting objects. They address the question of "Why should we study object calculi, instead of encoding objects in λ-calculi" [AC96, p77,257]. The aim of this thesis is to utilise their advances in object calculi to suggest improvements in the modelling of object-oriented refinement calculi.

From the assortment of calculi presented by Abadi and Cardelli [AC96] a calculus had to be chosen that would most easily integrate with the lattice-theoretic foundation of a refinement calculus. This section presents an introduction to the object calculus features that are used for this integration.

The form of the rules and judgements used by the object calculus is as follows. Each rule has a number of premise judgements above the line and a consequence judgement below the line.

Axiom 2.2 (Rule Name)

$$\frac{\mathit{premise}_1 \quad \mathit{premise}_2}{\mathit{consequence}_1}$$

Judgements Each judgement $E \vdash A$ consists of an environment E for the assertion A. A judgement of the form $E \vdash \diamond$ expresses that E is a well-formed environment. Given a type construction T, the judgement $E \vdash T$ expresses that T is a well-formed type (in environment E). Other forms of judgements used include:

Subtyping:

$$E \vdash A \preceq B$$

which states that A is a subtype of B in environment E,

Value typing:

$$E \vdash a : A$$

which states that a is of type A in environment E, and finally,

Truth:

$$E \vdash p$$

which states that p is true in the environment E, e.g.,

$$E \vdash true$$

Syntactic Definitions Syntactic definitions are provided using the syntax $\widehat{=}$.

Environment Omission When the environment of all premises and the consequence is the same and is evident from the context (or is irrelevant) the environment can be omitted.

$$\frac{\vdash A_1}{\vdash A_2} \mathrel{\widehat{=}} \frac{E \vdash A_1}{E \vdash A_2}$$

Rule Abbreviations When writing rules, the abbreviation *for all* $(i \in 1..n) \bullet E_i \vdash A_i$ (where $n > 0$) stands for n premises of the form $E_1 \vdash A_1 ... E_n \vdash A_n$. A premise of the form $j \in 1..n$, in comparison, denotes that there are n different rules, one for each $1..n$. For example, the rule

$$\frac{\vdash A_j \;\; j \in 1..n}{\vdash B_j}$$

is actually the following set of rules.

$$\frac{\vdash A_1}{\vdash B_1} \cdots \frac{\vdash A_i}{\vdash B_i} \cdots \frac{\vdash A_n}{\vdash B_n}$$

Environments Environments are formed by the union of value-type environments and type environments.

$$Environment \mathrel{\widehat{=}} ValueTypes \cup Subtypes$$

A value-type environment is a relation from labels to types representing the declared type of each label (variable).

$$ValueTypes \mathrel{\widehat{=}} Labels \pfun Types$$

Although the type of variables may not be necessarily unique (due to subtyping) the environment stores the declared, or least type.

A type environment is a relation from types to types representing subtype relationships. The use of a single type T is actually an abbreviation of its relation to the *Top* type, $T \preceq Top$.

$$Subtypes \mathrel{\widehat{=}} Types \leftrightarrow Types$$

Least Type The environment stores the least type (most defined type) of a variable. Hence rules that use the knowledge of a variable's least type are actually constraining the environment. To aid readability, however, a 'lowest type' judgement is introduced. For any environment E:

$$\frac{}{(a : B) \cup E \vdash a :_{\perp} B}$$

Free Variables The term T *nfi* A means that T may not occur free in A.

Substitution Syntactic substitutions are effected using the notation $A[x \backslash y]$ which denotes the substitution of y for x in A.

The object calculus presents constructs termed objects. **Objects differ from records (or functions) in that each component attribute has access to its own host.** Objects are typed and a subtyping relationship is introduced that is similar to the one presented in Section 2.1. The subtyping relationship ($\preceq$) is defined as set inclusion within the object calculus semantics. In this thesis it is treated as a syntactic operator and its properties are presented as given rules.

Object Types Object types have the syntax $Objtype \left\{ \; l_1 \mapsto \beta_1, ..., l_n \mapsto \beta_n \; \right\}$ where $l_1, ..., l_n$ are labels specifying the names of the attributes; $\beta_1, ..., \beta_n$ are the types associated with the attributes. For example, objects of type $Objtype \left\{ \; l \mapsto \mathbb{N} \; \right\}$ have an attribute l of type $\mathbb{N}$. An alternative syntax for object types is $Objtype \left\{ \; i \in 1..n \bullet l_i \mapsto \beta_i \; \right\}$. This object type has attributes $l_1, ..., l_n$ of the respective types $\beta_1, ..., \beta_n$.

The object calculus contains many rules for manipulating object calculus constructs and for determining various properties of those constructs. For example, the rule which can be used to show that such types are well-formed (syntactically and semantically 'valid') is termed **Type Object**. All properties that present well-formed types have the prefix **Type**. Other rule name prefixes are **Eq** for properties that equate values—equality properties; **Eval** for evaluation—properties that reduce or evaluate values that are not in their simplest form; **Val** for properties that present the types of values; **Sub** for subtyping properties; and **Env** for rules that allow well-formed environments to be constructed. The following is the rule for establishing whether an object type is well-formed.

Axiom 2.3 (Type Object) The object type $Objtype \left\{ \; i \in 1..n \bullet l_i \mapsto \beta_i \; \right\}$ is well-formed if the type of each of its attributes is well-formed and the labels are distinct.

$$\frac{\textit{for all } (i \in 1..n) \bullet \vdash \beta_i \quad l_i \text{ distinct}}{\vdash Objtype \left\{ \; i \in 1..n \bullet l_i \mapsto \beta_i \; \right\}}$$

The derivation of such rules is outside the scope of this thesis. Interested readers are referred to the work of Abadi and Cardelli [AC96].

Methods Methods have the syntax $\varsigma(x : X)\ body$ where ς is an object quantifier analogous to the λ quantifier in the lambda calculus. The term x is the self parameter and X is its type. Whenever a method is invoked, x is bound to the host object. The body of the method, *body*, is free to reference x. Consequently the behaviour of a method is dependent upon the context, or host, in which the method is invoked. This, in essence, simulates the behaviour of dynamic dispatch. Methods that do not reference their self parameters, x, are termed fields.

Objects Objects are collections of labels and associated methods. The syntax of objects is $object \left\{\ l_1 \mapsto \varsigma(x : X)\ body_1, ..., l_n \mapsto \varsigma(x : X)\ body_n\ \right\}$. The following is an alternative syntax for objects: $object \left\{\ i \in 1..n \bullet l_i \mapsto \varsigma(x : X)\ body_i\ \right\}$.

All objects have a type. The following rule, **Val Object**, allows the type of an object to be determined. The object type of an instance is determined by the types of the method (or field) bodies. These method types are collected and associated with the appropriate labels within the resulting object type. An alternative view of the following rule is that to show that an object is of a certain type, namely $Objtype \left\{\ i \in 1..n \bullet l_i \mapsto \beta_i\ \right\}$, one must show that the types of its methods (fields) correspond to the relevant types in the object type.

Axiom 2.4 (Val Object) Given $\gamma \equiv Objtype \left\{\ i \in 1..n \bullet l_i \mapsto \beta_i\ \right\}$

$$\frac{for\ all\ (i \in 1..n) \bullet\ E \cup \{x_i \mapsto \gamma\} \vdash body_i : \beta_i}{E \vdash object \left\{\ i \in 1..n \bullet l_i \mapsto \varsigma(x_i : \gamma)\ body_i\ \right\} : \gamma}$$

Subtyping There is a characteristic property of subtyping ($\preceq$) called subsumption. This property is formalised by the rule **Val Subsumption**.

Axiom 2.5 (Val Subsumption) Given an object of a particular type δ, that object also has the type of any supertype of δ.

$$\frac{\vdash d : \delta \quad \vdash \delta \preceq \gamma}{\vdash d : \gamma}$$

An object does not change when it is is subsumed, rather the static information known about the object is reduced. Consequently, subsumption does not alter the behaviour of an object.

Subtyping is a preorder (reflexive and transitive). Basic types, such as the Booleans and the natural numbers, subtype discretely. That is, they only subtype reflexively. Subtyping of object types occurs when the subtype has more attributes.

Axiom 2.6 (Sub Object)

$$\frac{for\ all\ (i \in 1..n+m) \bullet \vdash \beta_i \quad l_i \text{ distinct}}{\vdash Objtype \left\{\ i \in 1..n+m \bullet l_i \mapsto \beta_i\ \right\} \preceq Objtype \left\{\ i \in 1..n \bullet l_i \mapsto \beta_i\ \right\}}$$

The assumption ensures the attributes of the subtype have well-formed types.

Equivalence Equivalence relations typically relate values of the same type. However, due to subtyping and subsumption, objects can take on multiple types. Consequently, there are many equivalence relations, one for each type, or alternatively, the equivalence relation is parameterised by a type. The syntax $c =_\beta d$ denotes that object c is equivalent to d under the type β. A result of the parameterisation of equality is that while two objects may be equivalent under one type, they may not be equal under another.

For object types, two instances (e.g., $\mathit{object}\ \{\ i \in 1..n+p \bullet l_i \mapsto \varsigma(x_i : \gamma)\ b_i\ \}$ and $\mathit{object}\ \{\ i \in 1..n+q \bullet l_i \mapsto \varsigma(x_i : \gamma)\ b'_i\ \}$) are equivalent if, when truncated to the attributes in the type that parameterises the equivalence (γ in the rule below) the remaining attributes of both objects are equivalent.

Axiom 2.7 (Eq Object) Given $\gamma \equiv \mathit{Objtype}\ \{\ i \in 1..n \bullet l_i \mapsto \beta_i\ \}$

$$\frac{\mathit{for\ all}\ (i \in 1..n) \bullet\ E \cup \{x_i \mapsto \gamma\} \vdash b_i =_{\beta_i} b'_i}{E \vdash \mathit{object}\ \{\ i \in 1..n \bullet l_i \mapsto \varsigma(x_i : \gamma)\ b_i\ \} =_\gamma \mathit{object}\ \{\ i \in 1..n \bullet l_i \mapsto \varsigma(x_i : \gamma)\ b'_i\ \}}$$

The assumption forces the equivalence of the attributes under the assumption that the self parameter (x_i) has type γ.

Method Invocation/Field Selection In this object calculus, fields and methods are treated identically. Thus method invocation is also field selection. Method selection is equivalent to 'extracting' the relevant method body and binding the self parameter. So for the object $\mathit{object}\ \{\ l_1 \mapsto \varsigma(x)\ 5, l_2 \mapsto \varsigma(y)\ y\ \}$, selecting field l_1 first binds x to the host object; however since there is no occurrence of x in 5 this binding has no effect. Consequently selecting l_1 returns the object 5. In comparison, when method l_2 is invoked, y is bound to the host object and hence the method returns y which is the host object itself. The name of the property for method invocation or field selection is **Eval Select**.

Axiom 2.8 (Eval Select) Given

$$\gamma \equiv \mathit{Objtype}\ \{\ i \in 1..n \bullet l_i : \beta_i\ \}$$

and

$$c \equiv \mathit{object}\ \{\ i \in 1..n \bullet l_i = \varsigma(x_i : \gamma)\ b_i\ \}$$

then $$\frac{\vdash c : \gamma \quad j \in 1..n}{\vdash c_\odot l_j =_{\beta_j} b_j[x_i \backslash c]}$$

where $b_j[x_i \backslash c]$ stands for the substitution of x_i with c in b_j. The assumption $j \in 1..n$ is used to parameterise the rule, acting as though there are actually n rules, one for each $j \in 1..n$.

For method invocation (and field selection), another relevant rule is **Val Select**. The type of a method invocation (field selection) is the type of the method body associated with the label being selected. For example, for the object type $\mathit{Objtype}\ \{\ l \mapsto \mathbb{N}\ \}$, the type of the method invocation of l is $\mathbb{N}$.

Axiom 2.9 (Val Select)

$$\frac{\vdash c : \mathit{Objtype}\left\{\ i \in 1..n \bullet l_i \mapsto \beta_i\ \right\} \quad j \in 1..n}{\vdash c_{\odot} l_j : \beta_j}$$

The final property relevant to method invocations is **Eq Select**. It is desirable that given two equivalent objects, selecting the same label on each produces two objects that are also equivalent.

Axiom 2.10 (Eq Select) Given $\gamma \equiv \mathit{Objtype}\left\{\ i \in 1..n \bullet l_i : \beta_i\ \right\}$

$$\frac{\vdash c =_{\gamma} c' \quad j \in 1..n}{\vdash c_{\odot} l_j =_{\beta_j} c'_{\odot} l_j}$$

Method Update The evaluation of a method update (field update) is intuitive: updating an object c at label l_j with body b, denoted by $c_{\odot} l_j \Leftarrow \varsigma(x : \gamma)\ b$, returns an object which equals c at all labels, except at l_j which is now b. **Eval Update** is used to evaluate the result of a method update[2].

Axiom 2.11 (Eval Update) Given

$$\gamma \equiv \mathit{Objtype}\left\{\ i \in 1..n \bullet l_i : \beta_i\ \right\}$$

and

$$c \equiv \mathit{object}\left\{\ i \in 1..n \bullet l_i \mapsto \varsigma(x_i : \gamma)\ b_i\ \right\}$$

then

$$\frac{E \vdash c : \gamma \quad E \cup \{x \mapsto \gamma\} \vdash b : \beta_j \quad j \in 1..n}{E \vdash c_{\odot} l_j \Leftarrow \varsigma(x : \gamma)\ b =_{\gamma} \mathit{object}\left\{\ l_j \mapsto \varsigma(x : \gamma)\ b, i \in (1..n) \setminus \{j\} \bullet l_i \mapsto \varsigma(x_i : \gamma)\ b_i\ \right\}}$$

These rules form the core of the object calculus upon which our work is based. Using similar calculi Abadi and Cardelli have succeeded in creating several object-oriented languages by providing translations from the language constructs into the object calculus constructs. These languages have varying levels of functionality; some are almost fully featured object-oriented languages.

[2]Typically the rule is generalised to allow reasoning about objects with a different number of attributes. Here, for readability, the rule is simplified, allowing reasoning only about objects with the same number of attributes.

2.3 Refinement Calculus

The refinement calculus is a notation and set of rules for the calculation of executable programs from specifications [Bac80, Mor87, Mor94, BvW98]. One inspiration for the refinement calculus was Dijkstra's [Dij76] weakest precondition semantics. This work provides a predicate transformer semantics for the guarded command language and also lists several properties, termed healthiness conditions, that programs written in the guarded command language possess. The work on the refinement calculus showed that some of these conditions are overly restrictive and that by weakening them it is possible to include new constructs in the language. The main addition to the language is the specification statement. Adding the specification statement to the guarded command language means that the newly representable abstract programs no longer obey Dijkstra's Law of the excluded miracle—thereby allowing the production of miraculous, non-implementable programs, if the developer is not careful.

The inclusion of specifications in the same language as the implementation or code constructs leads to the term *wide-spectrum language* and distinguishes the refinement calculus from earlier work because derivations of programs can now be carried out within a single semantics.

Dijkstra's weakest precondition semantics presented language constructs as functions: the construct

$$wp(P, post)$$

returns the weakest precondition necessary for the program P to execute and terminate in a state satisfying the postcondition *post*. Specifications can be written in this style:

$$pre \Rightarrow wp(P, post)$$

meaning "if activated in a state for which *pre* holds, the program P must terminate in a state for which *post* holds."

Morgan's version of the refinement calculus mutated such specifications to the following form under the assumption that it will be transformed into P:

$$\left[pre, post \right]$$

This new form is considered an (abstract) statement. Morgan's weakest precondition semantics of this statement is:

$$wp(\left[pre, post \right], R) \;\widehat{=}\; pre \wedge (\forall \vec{v} \bullet post \Rightarrow R)$$

That is, for the specification to terminate in a state satisfying R, in the initial state *pre* must hold and essentially, *post* must be a stronger condition than R. The quantification of the program variables $\vec{v}$ is necessary as there are two states being considered. The predicate is a constraint on the initial state; thus the variables of the final state are required to

be quantified. This specification statement has been both improved and decomposed in various ways by Back and Morgan. For instance, Morgan presents a *framed* specification which constrains the variables that may be altered. The *frame w* identifies the modifiable variables.

$$wp(\vec{w}\!:\!\left[\, pre \;,\; post \,\right], R) \;\widehat{=}\; pre \wedge (\forall \vec{w} \bullet post \Rightarrow R)[\vec{v}_0 \backslash \vec{v}]$$

Morgan allows the convention that zero-subscripted variables in *post* denote the initial state of those variables. Consequently the substitution, after quantification of the 'final' variables, produces a predicate on the initial state.

Given an abstract program including specifications, the goal is to produce an executable program through a process of refinement. The program P refines to the program Q, written $P \sqsubseteq Q$, if for all (postcondition) predicates (R) the weakest precondition of Q with respect to that predicate R, denoted by $wp(Q, R)$, is weaker than (or will be guaranteed by) the corresponding weakest precondition of P:

$$P \sqsubseteq Q \;\widehat{=}\; \forall R \bullet wp(P, R) \Rightarrow wp(Q, R)$$

Alternatively, for any postcondition R, if P achieves R from a particular initial state, then Q will also achieve R from that same state. Informally, Q is "at least as good as" P.

The most important property of the refinement calculus is monotonicity. This allows individual program fragments to be refined and then substituted back into the program with the result being that the entire program is refined. Given programs $F(P)$ and $F(Q)$ then if $P \sqsubseteq Q$ then

$$F(P) \sqsubseteq F(Q)$$

Consequently, programs can be refined *piece-wise*. The statements in the guarded command language are all monotonic with respect to refinement.

Using the refinement relation, along with the weakest precondition semantics of the specification statement and the classical imperative language constructs (as given by Dijkstra), various refinement rules can be provided. Once these rules have been developed, the underlying weakest precondition semantics can be ignored, thereby providing a method that allows the calculation of programs from initial specifications. Typically this process involves some intuition to guide the development process. This process is termed algorithmic refinement, or procedural refinement.

The weakest precondition of a program includes the constraint necessary for that program to terminate. *Weakest liberal preconditions* remove the termination constraint leaving only the weakest precondition to establish the goal if the program terminates. For instance, the weakest liberal precondition semantics of the specification statement is defined as follows.

Definition 2.12 (Wlp Specification) For predicate R in an environment with variables $\vec{v}$:

$$wlp.(\vec{w}\!:\!\left[\, pre \;,\; post \,\right]).R \;\widehat{=}\; pre \Rightarrow (\forall \vec{w} \bullet post \Rightarrow R)[\vec{v}_0 \backslash \vec{v}]$$

◇

For the example specification $a{:}\left[\, a = 1 \;,\; a = a_0 + 1 \,\right]$ the weakest liberal precondition is $a = 1 \Rightarrow (\forall\, \vec{a} \bullet a = a_0 + 1 \Rightarrow a = 2)[\vec{a_0} \backslash \vec{a}]$ which is equivalent to $a = 1 \Rightarrow a = 1$ which is *True*. If the specification terminates it will always achieve $a = 2$.

2.3.1 Introduction to Data-refinement

This section introduces data-refinement, both in general and in the context of the refinement calculus [Bac88, Mor89, BvW90, Mor90, MG90, vW92a, vW92b, GM93]. Also presented is the technique of *simulation* [Mil71] which 'wraps up' the use of data-refinement as an algorithmic refinement so that data-refinements can be used in practice.

A *data type* is a piece of data and a set of operations that have exclusive access to view or modify the piece of data. Informally, *data-refinement* denotes the replacement of a data type within a program to increase efficiency and/or produce implementable code. Typically, an abstract, mathematically clear and concise data type (termed here a specification) is replaced with a more concrete, implementation-like data type. The replacement of the specification data type with the implementation data type in **client** code is referred to as *simulation.* Data-refinement, then, is a formal, mathematical relationship between program fragments; namely the respective operations of the specification and implementation data types.

Given the data-refinement of a specification by an implementation data type, simulation is the proof of refinement between the specification client code and the implementation client code. The behaviour of the specification client code has been 'simulated' by the implementation client code—even though they work on different state spaces.

Data-refinement Examples There are three types of relationship that may exist between specification and implementation states. Two of these are functional relationships and the third is relational. One of the functional relationships occurs when the specification state is a function of the implementation state. This can be illustrated, for instance, when implementing a set as an array. For simplicity, sequences are used here as a model of arrays. The set

$$\{1, 2, 3\}$$

can be represented as the sequence

$$\langle 1, 2, 3 \rangle$$

or, alternatively, as

$$\langle 3, 1, 2 \rangle$$

Given n as the cardinality of the specification set, there are $n!$ implementation representations of the specification state as a sequence with no duplicates. Morgan [Mor94] provides specialised rules for data-refinements of this type. In this example, the implementation state has extra (positional) information that was included as a side effect of the data-refinement of the set by a sequence. This information is not needed and hence may be ignored.

Another relationship occurs when the implementation state is a function of the specification. An example of this occurs for programs that average a list of numbers [Mor94, p170]. Given a specification data representation of a sequence of numbers, e.g., $\langle 3, 4, 5 \rangle$, data-refinement could be used to change the representation to a pair containing the sum and cardinality of the sequence: $(12, 3)$. The average can be determined using either data structure. For this type of data-refinement, the specification state has more information than is required. The data-refinement removes the unneeded information.

The third, relational, data-refinement relationship is illustrated in the following example.

Example 2.13 (Variable List) A program maintains a list of variable names and their associated string values. As variables are declared they are appended to a sequence. Given the variables *a* and *b* with values *dog* and *cat* respectively,

$$\langle (a, dog), (b, cat) \rangle$$

declaring a new variable *c* with value *horse* generates the sequence:

$$\langle (a, dog), (b, cat), (c, horse) \rangle$$

For efficiency reasons, the designer decided the list should be able to dynamically reconfigure itself such that the most frequently accessed variables are at the head of the list, thereby decreasing access time. To achieve this, the declaration-ordered sequence is data-refined to a sequence of triples ordered on the third triple element—which indicates the number of variable accesses. Given a variable *a*, accessed six-hundred times and *b*, accessed eight-hundred times, the sequences would be:

$$\langle (b, cat, 800), (a, dog, 600) \rangle$$

Declaring a variable *c* would result with:

$$\langle (b, cat, 800), (a, dog, 600), (c, horse, 0) \rangle$$

No functional relationship exists between the specification and implementation data structures. The specification data structure

$$\langle (a, dog), (b, cat) \rangle$$

could be represented by an implementation data structure in which *a* is accessed more:

$$\langle (a, dog, 8), (b, cat, 6) \rangle$$

or alternatively an implementation data structure in which *b* is accessed more.

$$\langle (b, cat, 800), (a, dog, 600) \rangle$$

Conversely, the implementation data structure

$$\langle (b, cat, 800), (a, dog, 600) \rangle$$

could be representative of a specification data structure in which *b* was declared first:

$$\langle (b, cat), (a, dog) \rangle$$

or, by a data structure in which *a* was declared first.

$$\langle (a, dog), (b, cat) \rangle$$

The specification data structure has additional declaration-order information that was stored as a side effect[3]. The implementation data structure loses this information yet gains extra 'access information' that is used for efficiency gains. In this example a functional relationship exists between the specification and implementation data structures once the extra, unneeded information of the specification data structure is removed. That is;

$$f(specification) = g(implementation)$$

Here f is simply the range of the sequences, indicating that the ideal data structure is a set of pairs relating variables with their values. The function g removes the third triple element and the ordering information.

$$g(implementation) = \{el \in \operatorname{ran} implementation \bullet (fst(el), snd(el))\}$$

Here the specification data structure

$$\langle (a, dog), (b, cat) \rangle$$

is mapped through f to

$$\{(a, dog), (b, cat)\}$$

as is the implementation data structure (through g)

$$\langle (b, cat, 8), (a, dog, 6) \rangle$$

[3]This extra information is termed, by de Roever[dREB+98, p8], implementation bias.

◇

One approach to data-refinement, as presented by Morgan et al. [MV94], is to introduce a predicate transformer (*rep*) that provides a link between specification and implementation predicates (as shown in Definition 2.14).

Definition 2.14 (Data-Refinement) The specification $Prog_s$ data-refines ($\preceq_{DR}$) to the implementation $Prog_i$ under *rep* if:

$$rep;\ \ Prog_s \sqsubseteq Prog_i;\ \ rep$$

That is:

$$Prog_s \preceq_{DR} Prog_i \ \widehat{=}\ rep;\ \ Prog_s \sqsubseteq Prog_i;\ \ rep$$

◇

Rules have been developed that allow *rep* to be 'pushed through' each language construct, piece-wise converting them to an appropriate implementation. For example, laws 2.15 and 2.16 are data-refinement rules for specifications and sequential composition respectively.

Law 2.15 (Data-refine Specifications) For the data-refinement of specifications, α denotes the specification variables being removed, β the implementation variables being introduced and γ the variables common to both specification and implementation states.

Morgan and Gardiner [MG90, p95, Corollary 1] introduce a (*validity*) data-refinement rule for specifications in which the postcondition *post* does not contain initial variables. For subset δ of the abstract variables α, i.e., $\delta \subseteq \alpha$, and $rep\ p \ \widehat{=}\ (\exists\, \alpha \bullet AI \wedge p)$ where *AI* is the abstraction invariant:

$$\begin{array}{l}
\delta, \gamma{:}\ [\, pre\ ,\ post\,] \\
\preceq_{DR} \\
\quad |[\, \mathbf{con}\ \alpha \bullet \\
\qquad\quad \beta, \gamma{:}\ [\, AI \wedge pre\ ,\ (\exists\, \delta \bullet AI \wedge post)\,] \\
\quad]|
\end{array}$$

They also show that data-refinement distributes through logical constants [GM91, p80, Lemma 8]. Consequently, the data-refinement rule can be generalised to allow initial variables in *post*.

$$\begin{array}{l}
\delta, \gamma{:}\ [\, pre\ ,\ post\,] \\
\equiv \text{Initial Variable (A.49)} \\
\quad |[\, \mathbf{con}\ D, G \bullet \\
\qquad\quad \delta, \gamma{:}\ [\, pre \wedge D = \delta \wedge G = \gamma\ ,\ post[\delta_0, \gamma_0 \backslash D, G]\,] \\
\quad]| \\
\preceq_{DR} \\
\quad |[\, \mathbf{con}\ D, G, \alpha \bullet \\
\qquad\quad \beta, \gamma{:}\ [\, AI \wedge pre \wedge D = \delta \wedge G = \gamma\ ,\ (\exists\, \delta \bullet AI \wedge (post)[\delta_0, \gamma_0 \backslash D, G])\,] \\
\quad]|
\end{array}$$

This rule also deals with assignments via Theorem A.52.

The following rule is used for the piece-wise data-refinement of sequential compositions, given the data-refinements of each construct.

Law 2.16 (Data-refine Sequential Composition) Given $pt_1 \preceq_{DR} pt_1'$ and $pt_2 \preceq_{DR} pt_2'$

$$pt_1;\ pt_2 \preceq_{DR} pt_1';\ pt_2'$$

Example 2.17 (Simple Data-Refinement: rep-style) For example, a program fragment that initialises a set, subsequently adds an element and finally sets the variable d to the cardinality of the set:

$$s{:}\left[\, s = \{6\} \,\right];\ \ s{:}\left[\, s = s_0 \cup \{3\} \,\right];\ \ d{:}\left[\, d = \#s \,\right]$$

can be data-refined to a program that uses sequences instead:

$$i{:}\left[\, i = \langle 6 \rangle \,\right];\ \ i{:}\left[\, i = i_0 \frown \langle 3 \rangle \,\right];\ \ d{:}\left[\, d = \#(\operatorname{ran} i) \,\right]$$

To calculate the implementation, the specification is prepended with an appropriate *rep*, in this case: $rep\ \phi \ \widehat{=}\ \exists s \bullet \operatorname{ran} i = s \wedge \phi$. This takes a specification predicate and translates it into the corresponding implementation predicate.

Using law 2.15 the following program segment can be deduced by 'pushing' *rep* through the first specification of the original program.

$$i{:}\left[\, i = \langle 6 \rangle \,\right];\ \ rep;\ \ s{:}\left[\, s = s_0 \cup \{3\} \,\right];\ \ d{:}\left[\, d = \#s \,\right]$$

Again *rep* is pushed through the second[4] and third specifications to give

$$i{:}\left[\, i = \langle 6 \rangle \,\right];\ \ i{:}\left[\, i = i_0 \frown \langle 3 \rangle \,\right];\ \ d{:}\left[\, d = \#(\operatorname{ran} i) \,\right];\ \ rep$$

◇

The disadvantage of using such an approach is that the chosen *rep* must be shown to possess the following properties:

Desideratum 2.18 (Universal Join Homomorphism of rep) For any (specification) predicates ϕ_i,

$$rep(\bigvee_i \phi_i) \equiv \bigvee_i (rep(\phi_i))$$

Gardiner and Morgan [GM91] refer to this property as $\vee$-distributivity.

From this property the following two properties hold. They are listed in their own right for ease of reference.

[4] The calculation of the second specification is non-deterministic. An alternative concrete specification would be: $i{:}\left[\, i = \langle 3 \rangle \frown i_0 \,\right]$.

Desideratum 2.19 (Strictness of rep) Strictness (also known as feasibility) denotes that the only way to achieve the impossible is to start with the impossible. Strictness is required for example, to allow **abort** to data-refine to itself.

$$rep(false) \equiv false$$

This is a special case of Desideratum 2.18 where the disjunction is empty: $(\bigvee_i \phi_i) \equiv$ *false*.

Desideratum 2.20 (Monotonicity of rep) *rep* must also be monotonic.

$$\begin{array}{l} \quad p \Rrightarrow q \\ \Rrightarrow \\ \quad rep\ p \Rrightarrow rep\ q \end{array}$$

The proof that join homomorphism implies monotonicity has been completed by Back and von Wright [BvW98, p45].

Finally, the prepending of *rep* to the specification program, and its removal from the implementation program, requires explanation. This is part of the process of simulation.

Simulation An early definition of *simulation* involved defining a relation between two programs which were "said to be realizations of the same algorithm" [Mil71]. This was used by Hoare [Hoa72, HHS87] and provided the basis for data-refinement in the refinement calculi.

More recently, de Roever, Paul and Engelhardt [dREB+98] have summarised data-refinement and simulation techniques used in several areas of specification and refinement, including Z, VDM, and Back's refinement calculus. Morgan and Gardiner [MG90, p485] present a "soundness of data-refinement" theorem which allows simulation in the context of data-refinement.

After calculating the corresponding implementation program fragment, the specification variable block (with initialisation *Init*) can be refined by the implementation variable block (with initialisation *rep Init*), e.g., for the previous example:

$$\begin{array}{l} \quad |[\ \textbf{var}\ s : S \mid Init \bullet \\ \qquad s{:}\left[\, s = \{6\} \,\right];\ \ s{:}\left[\, s = s_0 \cup \{3\} \,\right];\ \ d{:}\left[\, d = \#s \,\right] \\ \quad]| \\ \sqsubseteq \\ \quad |[\ \textbf{var}\ i : I \mid rep\ Init \bullet \\ \qquad i{:}\left[\, i = \langle 6 \rangle \,\right];\ \ i{:}\left[\, i = i_0 \frown \langle 3 \rangle \,\right];\ \ d{:}\left[\, d = \#(\mathrm{ran}\, i) \,\right] \\ \quad]| \end{array}$$

Thus the data-refinement of the specification code forms part of the proof of the 'simulation' of the specification variable block by the implementation variable block.

2.3.2 Lattice-theoretic semantics

Back and von Wright's [BvW98] refinement calculus is founded in lattice and category theory, e.g., the refinement relation is a lattice ordering. This section provides an introduction to lattice theory.

Lattices are built from a number of other well known mathematical abstractions. One such abstraction is the partial order. A partial order is a relation that is reflexive, antisymmetric and transitive. Examples include $\Rightarrow$ on the type $\mathbb{B}$, $\leq$ on the type $\mathbb{N}$, and $\subseteq$ on the powerset of any set S, that is $\mathbb{P}\,S$. An equivalence relation, in contrast, is reflexive, symmetric and transitive. Equality on the type $\mathbb{N}$ is one example of an equivalence relation.

A partial order may have a bottom element. For example, the natural numbers, with $\leq$ has zero as its bottom element. In general, the bottom element is denoted as $\bot$. The statement **abort** is the bottom of the refinement relation ordering in the refinement calculus. This means, amongst other things, that **abort** refines to anything, or alternatively, any program refines **abort**.

Besides the partial order relation, $\leq$, a set may also have other operators between its members. For instance, the *min* function on the natural numbers takes two numbers and returns the greatest number that is less-than-or-equal to both numbers. This is called the meet or greatest lower bound. When not applied to a specific application, the meet is denoted by the symbol $\sqcap$. The meet of booleans is boolean conjunction, the meet of sets (with $\subseteq$) is intersection, the meet of predicates (with $\Rightarrow$) is logical conjunction, and the meet of predicate transformers is demonic nondeterministic choice. The meet of a subset of a partial order's set is the greatest lower bound or greatest element in the set of lower bounds of that subset. For example, the *min* function which is the meet for the *poset* $(\mathbb{N}, \leq)$, when applied to the subset $\{4, 6\}$, examines all lower bounds of the set, i.e., $\{0, 1, 2, 3, 4\}$ and returns the greatest element, i.e., 4.

Joins are similarly defined, except that they return the least upper bound, rather than the greatest lower bound. That is, the least element of the set of upper bounds is returned by the join. For example, $\cup$ is the join for the poset $(\mathbb{P}\,S, \subseteq)$.

Lattices Informally, a lattice is a partial order where the meet and join exists as a member of the lattice for any two elements. Example of lattices include $\Rightarrow$ on the type $\mathbb{B}$ with $\wedge$ as the meet and $\vee$ as the join, or alternatively, $\subseteq$ on the type $\mathbb{P}\,S$ with $\cap$ as the meet and $\cup$ as the join. The term lattice is derived from the lattice-like appearance that the graph of a lattice makes. Figure 2.2 shows nine example lattices. For example, lattice four uses the set $\{a, b, c, d\}$ with the following relationships: $d \leq b$, $d \leq c$, $b \leq a$ and $c \leq a$. Lattice one is the only lattice that is possible with a single element. Lattice two is the only shape that a two member lattice can form. Similarly, lattice three is the only three member lattice shape. Lattices four and five are the four member lattices while the

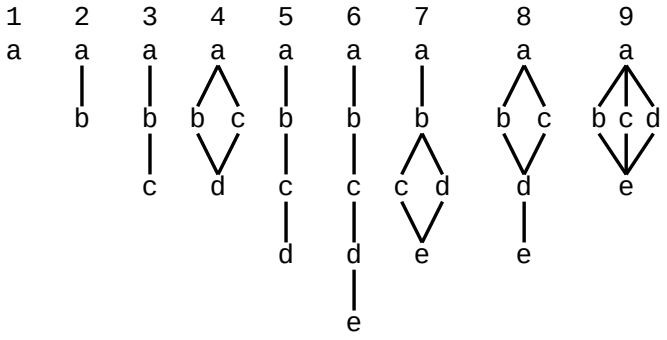

Figure 2.2: Lattice Shapes

remainder are the five member lattices.

Chapter 3

Literature Survey

This chapter provides an introduction to the contentious issues of object-oriented refinement calculi modelling. The description of issues is followed by summaries of several object-oriented refinement calculi.

There have been several attempts at producing an object-oriented refinement calculus, and all vary significantly in both their foundations and features. In particular, the representation of objects varies significantly amongst the calculi. Classical object-oriented languages portray the inclusion of fields and methods in objects as desirable. However, encapsulated methods are difficult to represent due to their recursive nature: methods manipulate the objects of which they are a component. Utting [Utt92] presents this problem in the context of denotational semantics and proves that, in general, there is no solution to the domain equations that allow fields and methods in the same object. Naumann and Calvanti [Nau94b, p471] [CN99a, p1447], however, specialise the general case and show that for certain object-oriented programs, a solution can be found.

When developing a semantics for an object-oriented refinement calculus, the *binary method* problem [BCC+95] is relevant for type-theoretic attempts. Methods which use a value argument of the same type as the method's host (termed the self type) and return a result argument, also of the self type, are called binary methods. Type theoretic researchers have had difficulty typing objects that contain binary methods. Unfortunately, an object-oriented refinement calculus semantics has a need for such objects. An intuitive, abstract representation of objects in an object-oriented refinement calculus is one that contains methods which are predicate transformers that manipulate the host object. That is, the predicate transformers are functions from predicates on a state type involving the host type to predicates on a state type also involving the host type. Consequently these predicate transformers have similar problems to binary methods.

The most common solution to the binary method problem has been to decouple fields and methods. Typically this involves using an object comprised of fields and an external collection of methods that manipulate the object's fields. This however, violates the encapsulation principles of object-orientation—though (through practical restrictions) only in the semantics.

In the context of an object-oriented refinement calculus, the models of object-oriented concepts such as object representation, dynamic dispatch and field update are highly dependent upon each other. There are as many object-oriented refinement calculi as there are object representations.

The modelling of dynamic dispatch falls into one of two approaches, based upon whether the object representation decouples methods. If methods are embedded then dynamic dispatch can be modelled by the execution of the contained method. When methods are not embedded in objects a mechanism must exist for associating them. This association is achieved by the object-oriented refinement calculi reviewed here in the following ways. The approach of Mikhajlova and Sekerinski [MS97] uses an implicit tag whereby object instances are linked to their declaration class which contains a list of methods. Utting [Utt92] provides a function that, in its simplest form, maps object instances, through method names, to the method to be invoked.

A common aspect of object-oriented refinement calculi is their treatment of behavioural conformance of polymorphic objects: given classes C and D where D is a subclass of C, and a program S written for instances of class C, then using subsumed instances of class D instead produces a refinement. This is the fundamental property of (class-based) object-oriented refinement calculi. If this constraint was not applied then it would be possible to form an arbitrary subclass. Instances of this subclass could be subsumed and used as instances of the superclass. The behaviour of such instances would not necessarily conform to that dictated by the superclass specification. By fusing polymorphic reuse with behavioural conformance correct class behaviours can be guaranteed.

The manner in which behavioural conformance of polymorphic objects, also known as *modular reasoning* [Utt92], is implemented varies significantly among object-oriented refinement calculi, and will be examined on a case-by-case basis.

At present most research in the field of object-oriented refinement calculi has concentrated on the modelling of object-oriented concepts with little attention paid to the development of additional refinement techniques that an object-oriented refinement calculus allows.

The remainder of the chapter is a collection of summaries of existing object-oriented refinement calculi. Each summary loosely follows a format consisting of theoretical foundations, object representation, dynamic dispatch, field selection and update, subtyping, class representation, and behavioural conformance.

3.1 Modular Reasoning

Utting and Robinson [UR92] use a lattice-theoretic framework. Object representation is not made explicit. The only assumptions made are that objects are a subset of the value space, and methods are not included.

To model the object's methods and consequently dynamic dispatch, they define a 'late

binding' function

$$\psi : P \rightarrow Val^* \rightarrow Prog$$

that maps procedure names (P) to functions ($Val^* \rightarrow Prog$). The functions map finite sequences of values (Val^*) to program code ($Prog$). The values are used to determine the code to execute. For classical object-oriented languages, the (single) value used would be the host object. However, since multiple values (objects or method arguments) can be used to determine the code to execute, the resulting language is *multiple dispatch.*

For multiple dispatch languages the *dynamic dispatch* mechanism is not dependent upon a single object instance. In this context the decoupling of methods and fields is natural. However, Utting and Robinson indicate that this multiple dispatch mechanism may permit inconsistencies and a loss of object encapsulation as the object(s) used to determine dynamic dispatch may be different from the object which the method manipulates. Consequently, they enforce links between parameterisation and dynamic dispatch. That is, they ensure the values (objects) used to determine the code to execute are the same values passed as parameters to the selected code.

Utting and Robinson constrain a reflexive and transitive subtyping relation with behavioural conformance constraints for polymorphic objects. This *modular reasoning* is achieved by restricting their late binding function (ψ) to be monotonic with respect to the subtyping relation. For objects c and d, where $c \preceq d$, and where both have a method p, then the substitution of c with d in a program is a refinement as the monotonicity of ψ guarantees that

$$\psi\ p\ c \sqsubseteq \psi\ p\ d$$

In comparison, other object-oriented refinement calculi achieve comparable properties by restricting the subclass relation with behavioural conformance constraints. By also restricting subsumption to comply with the subclass relation, similar monotonic properties can be deduced.

Utting and Robinson weaken the monotonicity requirement by using static properties known about arguments passed to the invoked methods. This allows increased flexibility while maintaining modular reasoning. It also means that transitivity is lost. To regain transitivity, a transitive modular reasoning relation is provided, thereby easing proof of modular reasoning for a chain of subtypes. Unfortunately, the subtyping relation does not fully support data-refinement. Utting and Robinson have commenced the development of a more general relation but proof that it maintains the properties of transitive modular reasoning is not complete. Another problem, suggested by Sekerinski [Sek96, p14], is that technical reasons preclude specifications within the methods of Utting and Robinson.

To provide a comparison between the object-oriented refinement calculi presented in this chapter each approach is applied to a simple example specification (and/or refinement) of a bag. The following example presents the refinement of a bag using modular reasoning and the construction of an associated late binding function.

Example 3.1 (Refinement of Bag using Modular Reasoning) Consider a bounded bag object with a *bagdata* field representing the bag information, and a *bound* field containing the maximum number of elements allowed in the bag. The cardinality of the bag is constrained to be not greater than the bound: $\#(a_{\odot}bagdata) \leq a_{\odot}bound$. The # symbol is bag size. The bounded bag object also contains a *put* method which, if possible, inserts a new element into the bag, a *get* method which returns an element from the bag (removing it at the same time) and a *card* method for determining the cardinality of the bag. Such an object is illustrated by the following, informal, syntax where *Bound* is some given constant, and *PUT*, *GET* and *CARD* are the respective method bodies:

object
  private field $bagdata : \mathrm{bag}(ELEMENTS) := [\![\,]\!]$
  private field $bound : \mathbb{N} := Bound$
  method $put($**value** $el : ELEMENTS$**)** $= PUT$
  method $get($**result** $el : ELEMENTS$**)** $= GET$
  method $card($**result** $num : \mathbb{N}$**)** $= CARD$
end

The only constraint Utting and Robinson place on their object representation is that objects *Obj* are a subset of the value space *Val*.

$$Obj \subseteq Val$$

Their examples, however, define objects using partial functions from a set of names *Var* to values[1]. This example uses this same representation.

$$Obj \mathrel{\widehat{=}} Var \pfun Val$$

Subtyping ($\preceq$) is defined within the examples of Utting and Robinson so that objects with a set of attributes are subtypes of those objects with a subset of those attributes. Additionally, the fields must exist in a subtyping relationship.

$$a \preceq b \mathrel{\widehat{=}} a \in Obj \land b \in Obj \land \mathrm{dom}\, b \subseteq \mathrm{dom}\, a \land \forall i : \mathrm{dom}\, b \bullet a_{\odot} i \preceq b_{\odot} i$$

The record field types are said to vary covariantly. *Covariance* is the monotonicity property of record field types according to the subtype relation. If the field types vary antimonotonically, they are said to be *contravariant*. For simplicity, it is assumed that basic, non-object types subtype discretely (only reflexively)[2].

The parameterised predicate *BagType* captures the required typing properties of the fields of a bounded bag.

$$BagType(a) \mathrel{\widehat{=}} a \in Obj \land \mathrm{dom}\, a = \{bound, bagdata\} \land$$
$$a_{\odot}bound \in \mathbb{N} \land a_{\odot}bagdata \in \mathrm{bag}(ELEMENTS)$$

[1] By abusing notations and presenting function application ($a(i)$) using the syntax given in the introduction for method invocation ($a_{\odot}i$), a more consistent presentation is provided here.

[2] When examining Utting and Robinson's work, care is required as the syntax for their subtype relationship $\leq$ is syntactically reversed when compared to the syntax for the subtyping relationship used in this thesis: $a \preceq b \equiv a \geq b$.

These constraints are that a is an object ($a \in Obj$) with fields *bound* and *bagdata* ($\mathrm{dom}\, a = \{bound, bagdata\}$); that *bound* is a natural number and that the bag must consist of elements of the given set *ELEMENTS*. The parameterised predicate *Bag* captures all desired properties of the bounded bag, typing and otherwise.

$$Bag(a) \;\widehat{=}\; BagType(a) \wedge \#(a_\odot bagdata) \leq a_\odot bound$$

The second conjunct constrains the cardinality of the bag to be not greater than the bound.

Unfortunately the predicate *BagType* is not applicable to subtypes of *Bag*. For instance, subtypes may wish to introduce new fields, yet *BagType* constrains the fields to be *bound* and *bagdata* only. Monotonic closure is used to generalise the *BagType* predicate for subtypes.

Definition 3.2 (Monotonic Closure) Monotonic closure is defined as:

$$\overline{P}^u \;\widehat{=}\; \exists u' \bullet P[u \backslash u'] \wedge u \preceq u'$$

where P is a predicate, u is a sequence of distinct variables. $\overline{P}^*$ is the monotonic closure with respect to all names, *Var*.

◇

The only effect of using the monotonic closure of *BagType* ($\overline{BagType(arg)}^*$) is that the predicate constraining a's domain is generalised to a superset of $\{bound, bagdata\}$:

$$\begin{array}{l}\overline{BagType(a)}^* \\ \equiv \\ a \in Obj \wedge \{bound, bagdata\} \subseteq \mathrm{dom}\, a \wedge \\ a_\odot bound \in \mathbb{N} \wedge a_\odot bagdata \in \mathrm{bag}(ELEMENTS)\end{array}$$

This can be seen by the following proof based on a similar one presented by Utting [Utt92, p49]:

$$\begin{array}{l}\overline{BagType(a)}^* \\ \equiv \text{Monotonic Closure (3.2)} \\ \exists a', bound', bagdata' \bullet \\ \quad a' \in Obj \wedge \mathrm{dom}\, a' = \{bound', bagdata'\} \wedge \\ \quad a'_\odot bound' \in \mathbb{N} \wedge a'_\odot bagdata' \in \mathrm{bag}(ELEMENTS') \wedge \\ \quad a \preceq a' \wedge bound \preceq bound' \wedge bagdata \preceq bagdata' \\ \equiv \text{Non-object variables subtype discretely (reflexively).} \\ \exists a' \bullet a' \in Obj \wedge \mathrm{dom}\, a' = \{bound, bagdata\} \wedge \\ \quad a'_\odot bound \in \mathbb{N} \wedge a'_\odot bagdata \in \mathrm{bag}(ELEMENTS) \wedge \\ \quad a \preceq a' \\ \equiv \text{Non-object fields of objects subtype discretely (reflexively).} \\ \exists a' \bullet a' \in Obj \wedge \mathrm{dom}\, a' = \{bound, bagdata\} \wedge \\ \quad a_\odot bound \in \mathbb{N} \wedge a_\odot bagdata \in \mathrm{bag}(ELEMENTS) \wedge \\ \quad a \preceq a'\end{array}$$

$\equiv$ Expand Subtype definition
$\exists\, a' \bullet a' \in Obj \land \operatorname{dom} a' = \{bound, bagdata\} \land$
$\quad a_\odot bound \in \mathbb{N} \land a_\odot bagdata \in \operatorname{bag}(ELEMENTS) \land$
$\quad a \in Obj \land a' \in Obj \land \operatorname{dom} a' \subseteq \operatorname{dom} a \land$
$\quad \forall\, i : \operatorname{dom} a' \bullet a_\odot i \preceq a'_\odot i$
$\equiv$ Predicate Calculus
$\exists\, a' \bullet a' \in Obj \land \operatorname{dom} a' = \{bound, bagdata\} \land$
$\quad a_\odot bound \in \mathbb{N} \land a_\odot bagdata \in \operatorname{bag}(ELEMENTS) \land$
$\quad a \in Obj \land \{bound, bagdata\} \subseteq \operatorname{dom} a \land$
$\quad \forall\, i : \operatorname{dom} a' \bullet a_\odot i \preceq a'_\odot i$
$\equiv$ Existence of a superobject a' of a.
$a \in Obj \land \{bound, bagdata\} \subseteq \operatorname{dom} a \land$
$a_\odot bound \in \mathbb{N} \land a_\odot bagdata \in \operatorname{bag}(ELEMENTS)$

The predicate $BagMC(a)$ is the monotonically closed analogy of the $Bag(a)$ predicate:

$$BagMC(a) \;\widehat{=}\; \overline{BagType(a)}^{*} \land \#(a_\odot bagdata) \leq a_\odot bound$$

This predicate holds for all subobjects of those specified by $Bag(a)$.

The *put* method is used to add an element *el* to the bag contained in the object *host*.

$$host.put_{Bag}(\textbf{value}\ el) \;\widehat{=}\; host\textbf{:}\left[\begin{array}{c} BagMC(host) \land el \in ELEMENTS \land \\ \#(host_\odot bagdata) < host_\odot bound \\ \hline BagMC(host) \land \\ host_\odot bound = host_{0\odot} bound \land \\ host_\odot bagdata = host_{0\odot} bagdata \uplus [\![el]\!] \end{array}\right]$$

There is a precondition on the operation that ensures the cardinality of the bag does not exceed the bound. Since the entire object *host* is in the frame, constraints are required for any fields that are to be kept invariant; hence the postcondition: $host_\odot bound = host_{0\odot} bound$. The final conjunct of the postcondition states that the final value of the *bagdata* field of *host* is the original bag unioned with a singleton bag containing *el*, that is $[\![el]\!]$. The symbol $\uplus$ represents bag union.

The get_{Bag} method uses *host* again as the host object argument, yet uses *el* as a result parameter to return an element from the bag.

$$host.get_{Bag}(\textbf{result}\ el) \;\widehat{=}\; host, el\textbf{:}\left[\begin{array}{c} BagMC(host) \land host_\odot bagdata \neq [\![\,]\!] \\ \hline BagMC(host) \land \\ host_\odot bound = host_{0\odot} bound \land \\ host_{0\odot} bagdata = host_\odot bagdata \uplus [\![el]\!] \end{array}\right]$$

The precondition $host_\odot bagdata \neq [\![\,]\!]$ allows undefined behaviour if the bag is empty. The final conjunct of the postcondition nondeterministically chooses *el* to be an element of *bagdata* and simultaneously removes it.

The $card_{Bag}$ method uses *host* as the host object parameter and *num* as a result parameter for the cardinality of the bag.

$$host.card_{Bag}(\textbf{result}\ num) \;\widehat{=}\; num\textbf{:}\left[\begin{array}{c} BagMC(host) \\ \hline num = \#(host_\odot bagdata) \end{array}\right]$$

Given *Bag* objects and the above three methods, the specification is completed by providing an appropriate late binding function ψ (method name $\rightarrow$ host objects $\rightarrow$ method). In this case the following function is sufficient.

$$\psi \mathrel{\widehat{=}} \left\{ \begin{array}{l} put \mapsto \{Bag \mapsto put_{Bag}\}, \\ get \mapsto \{Bag \mapsto get_{Bag}\}, \\ card \mapsto \{Bag \mapsto card_{Bag}\} \end{array} \right\}$$

Consider a subtype *SubBag* with corresponding methods put_{SubBag}, get_{SubBag} and $card_{SubBag}$. If $put_{Bag} \sqsubseteq put_{SubBag}$, $get_{Bag} \sqsubseteq get_{SubBag}$ and $card_{Bag} \sqsubseteq card_{SubBag}$ then the following ψ function is monotonic (in refinement) with respect to subtyping.

$$\psi \mathrel{\widehat{=}} \left\{ \begin{array}{l} put \mapsto \{Bag \mapsto put_{Bag}, SubBag \mapsto put_{SubBag}\}, \\ get \mapsto \{Bag \mapsto get_{Bag}, SubBag \mapsto get_{SubBag}, \}, \\ card \mapsto \{Bag \mapsto card_{Bag}, SubBag \mapsto card_{SubBag}\} \end{array} \right\}$$

This function models single, dynamic dispatch. Multiple dispatch can be achieved by using multiple objects in the determination of which method to execute.
$\diamond$

Using this model as a basis, the algorithmic refinement laws of the refinement calculus can be used for the development of programs.

3.2 Sums of Products

An alternative approach taken by Mikhajlova and Sekerinski [MS97] extends the typed lambda calculus with *sum types* and *product types* to produce a lattice-theoretic semantics.

Sum types, or *disjoint unions*, are used to combine multiple types while maintaining the distinctiveness of each type. The sum of two types σ and τ is written as $\sigma + \tau$ while σ and τ are termed the *base types*. The summation is undefined if the types are not disjoint.

Sum types are used by Mikhajlova and Sekerinski for modelling subtyping polymorphism and dynamic binding. They achieve polymorphism by using a statement summation operator defined by Back and Butler [BB95]. Building towards the statement summation operator, Back and Butler [BB95] define a predicate summation operator. Given a predicate $p_1 : \mathrm{pred}\,\sigma$, where $\mathrm{pred}\,\sigma$ is a function mapping elements of σ to the Booleans, and $p_2 : \mathrm{pred}\,\tau$, $p_1 + p_2$ acts as p_1 when given a state on σ and as p_2 when given a state on τ. In terms of this operator they define the statement summation operator. Consider a statement S of type **Tran** $\sigma_1\ \sigma_2$ where **Tran** *Pre Post* is a function from a predicate of type pred *Post* to the weakest precondition predicate of type pred *Pre*[3]. Summing S with T : **Tran** $\tau_1\ \tau_2$, written $S + T$: **Tran** $\sigma_1 + \tau_1\ \sigma_2 + \tau_2$, is defined as:

$$(S + T)\ q \mathrel{\widehat{=}} ((S;\ \langle\psi_1\rangle)\ q) + ((T;\ \langle\psi_2\rangle)\ q)$$

[3]The state types of the **Tran** *Pre Post* are syntactically reversed when compared to the *Ptrans Post Pre* syntax provided in Chapter 4.

for $q : \text{pred}\,\sigma_2 + \tau_2$ where $\langle\psi_1\rangle$ and $\langle\psi_2\rangle$ are used to inject states of type σ_2 and τ_2 respectively into the type $\sigma_2 + \tau_2$. When $S + T$ is executed in a state σ_1 it acts as S. When executed in a state τ_1 it acts as T. To effect late binding of an object's method, all versions of the method within a class hierarchy are summed together. The method invoked is then determined by the static type of the state space, which, in this context, is effectively the type of the object. The most important property of the summation operator is monotonicity:

$$S \sqsubseteq S' \wedge T \sqsubseteq T' \Rrightarrow (S + T) \sqsubseteq (S' + T')$$

This ensures that if a class' method is refined, the late binding call which uses the method is also refined.

Mikhajlova and Sekerinski chose to represent their objects using product types. Product types are also known as Cartesian products or tuples.

Example 3.3 (Objects as Products) This example presents the bag example from the previous section using the object representation as provided by Mikhajlova and Sekerinski. To model the bag, two fields are used: a bound of type $\mathbb{N}$ and a bag data field of type $\text{bag}(\mathit{ELEMENTS})$. The object type is the Cartesian product of the fields:

$$\mathbb{N} \times \text{bag}(\mathit{ELEMENTS})$$

An element of this type is $(5, [\![\ ]\!])$. The field labels are not part of the object—this information is maintained in the ordering of the object. Mikhajlova and Sekerinski use the loss of labelling to encapsulate the fields of the object.
◇

A subtyping relation (and thus subsumption) is defined so that an element of a base type is also an element of a sum type that incorporates that base type. For example, given an object $o_1 : O_1$ and a disjoint object type O_2 then via the subtyping relation and subsumption

$$o_1 : (O_1 + O_2)$$

Mikhajlova and Sekerinski use classes to couple an object's type with the methods its elements are to invoke. Classes also incorporate a constructor method which is used to perform initialisation—typically by manipulation of an object's fields. Class definition therefore involves specification of the fields, a constructor method, and the other class methods:

$C =$ **class**
  field $f_1 : F_1, ..., f_m : F_m$
  $C(p : P) = S$
  $\mathit{Meth}_1(i_1 : I_1) : O_1 = \mathit{Body}_1$
  :
  $\mathit{Meth}_n(i_n : I_n) : O_n = \mathit{Body}_n$
end

where $f_1, ..., f_m$ are the fields and $F_1, ..., F_m$ are their respective types, C is the name of the class and also the constructor method, $p : P$ is used to pass parameters to the constructor, S is the body of the constructor method, $i_1 : I_1, ..., i_n : I_n$ are the value parameters of the methods and $res : O_1, ..., res : O_n$ are the result parameters. Assignment within $Body_i$ to the variable *res* returns the value of the assignment as the result of the method.

Definition 3.4 (Classes as Tuples) Classes are represented as the tuple

$$(S, Body_1, ..., Body_n)$$

◇

The type of the constructor S is

$$S : \textbf{Tran}\ (F_1 \times ... \times F_m \times P)(F_1 \times ... \times F_m \times P)$$

where **Tran** *Pre Post* is the type of a predicate transformer from a predicate on the state space *Post* to a predicate on a state space *Pre*[4]. The constructor is modified with state enlargement and reduction commands to alter its type to **Tran** $P\ (F_1 \times ... \times F_m)$.

The types of the methods $Body_{i \in 1..n}$ are:

$$Body_i : \textbf{Tran}(F_1 \times ... \times F_m \times I_i \times O_i)(F_1 \times ... \times F_m \times I_i \times O_i)$$

Similar to the constructor, each method is subjected to state transformation statements changing their type to

$$\textbf{Tran}(F_1 \times ... \times F_m \times I_i)(F_1 \times ... \times F_m \times O_i)$$

Since late binding is determined by the product of the object's field types, it is not clear how Mikhajlova and Sekerinski can model the late binding of subclass instances in which no attributes are added, yet methods have been overwritten.

To ensure behavioural conformance of polymorphic objects, a constraint is placed on the declaration of a subclass. Given classes C and D, if class D is declared as a subclass of C then a proof obligation is generated requiring the developer to ensure that D is a class refinement of C. Class refinement is based on data-refinement. Consequently, an abstraction relation is required to provide a link between the abstract and concrete fields.

Definition 3.5 (Class Refinement of Sum Types) Omitting state modification technicalities, class refinement is defined as follows. If $C = (S, Body_1, ...Body_m)$ and $D = (S', Body'_1, ..., Body'_m)$

$$\begin{array}{l} C \sqsubseteq D \ \widehat{=} \\ \quad \exists R : \tau(D) \leftrightarrow \tau(C) \bullet \\ \qquad \{\pi_P\};\ S \sqsubseteq S';\ \{R\} \wedge \\ \qquad \forall j : 1..m \bullet \{R \times \pi_{I_j}\};\ Body_j \sqsubseteq Body'_j;\ \{R \times \iota_{O'_j}\} \end{array}$$

where R is an abstraction relation used to coerce the fields of D to those of C. π_P is the projection that coerces the constructor parameters, π_{I_j} coerces the input parameters and $\iota_{O'_j}$ coerces the output parameters.

[4]Care is required here as the *Ptrans* syntax used later in the thesis swaps the *Pre* and *Post* state spaces.

◇

Mikhajlova and Sekerinski also provide a relationship between classes that potentially have entirely different method parameter types. They term the relationship interface refinement.

3.3 Storing Procedure in Variables

Using a set-theoretic model, Naumann and Calvanti [Nau94a, Nau94b, CN00] define a higher-order language that supports extensible records and procedure-type variables. Procedure-type variables allow procedures to be stored in variables, and be passed as parameters to other procedures. Late binding can be modelled using these language features.

Placing methods inside objects which manipulate themselves complicates the typing of the objects. Research has provided typings for 'objects' which manipulate themselves yet do not contain specifications [Sch88] and for 'objects' that contain specifications yet do not do manipulate themselves [GHP95] but not both. Naumann simplifies the typing of the objects by imposing restrictions. Procedures cannot be recursive and the only external variables that may be modified are those that are global.

The definition of a call on a procedure variable and consequently an object's method is straightforward.

Definition 3.6 (Procedure Variable Call) Given predicate ψ, procedure variable *pv*, and the denotation brackets $|[\ ]|$

$$|[\,\textbf{call}\ \ pv\,]|.\psi = pv.\psi$$

◇

Naumann introduces a subtyping relation[5] that is reflexive, allows records to be extended and field types to subtype *covariantly.* In contrast with the approaches presented earlier, Naumann's work does not provide a generalised subsumption rule. That is, the following rule of subsumption is not a rule in Naumann's logic. The following rules are simplified versions provided for readability. Given object *a* and object types *A* and *B*:

$$\frac{a : A \quad A \preceq B}{a : B}\ \text{NOT A RULE}$$

The absence of this rule is deemed by Naumann to simplify the typing system. The role of subsumption is instead carried by several typing rules. The assignment rule allows objects to be assigned instances of subtypes. Given object types *A* and *B* where $A \preceq B$, variable *b* of type *B* and object $a : A$ then *b* can be assigned the instance *a* of subtype *A*:

$$\frac{b : B \quad a : A \quad A \preceq B}{b := a\ : com}(assign)$$

[5] Naumann originally used the syntax: $\Subset$. This thesis uses the $\preceq$ syntax for consistency.

where *com* is the type of well-formed statements or commands. There are two similar rules which support this typing rule in its role of subsumption. The call rule allows value parameters to vary covariantly and the type guard rule provides a type for a type case statement.

Unfortunately, no 'best' technique appears evident for the specification and refinement of the bag example. A naive solution is to use records as objects, e.g.,

$$
\begin{array}{l}
|[\ \textbf{var}\ bag : BAGTYPE \bullet \\
\quad bag := [bound : \mathbb{N}, bagdata : bag(ELEMENTS), \\
\qquad put : \left(\begin{array}{l} \textbf{pro}\ el : ELEMENTS \bullet \\ bag.bagdata\textbf{:}\left[\dfrac{\#bag.bagdata < bag.bound}{bag.bagdata = bag_0.bagdata \uplus el} \right] \end{array} \right), \\
\qquad get : \left(\begin{array}{l} \textbf{pro var}\ el : ELEMENTS \bullet \\ bag.bagdata, el\textbf{:}\left[\dfrac{bag.bagdata \neq [\![\,]\!]}{bag_0.bagdata = bag.bagdata \uplus el} \right] \end{array} \right), \\
\qquad card : \left(\begin{array}{l} \textbf{pro var}\ num : \mathbb{N} \bullet \\ num\textbf{:}\left[num = \#bag.bagdata \right] \end{array} \right) \\
\quad] \\
]|
\end{array}
$$

where *BAGTYPE* contains the appropriate typings for *bound*, *bagdata*, *put*, *get* and *card* and the (**pro** ....) syntax provides unnamed procedure abstractions.

For technical reasons, the external variables of a procedure (variables not passed as parameters to the procedure) need to be declared in the outermost scope. Using the technique above, all objects would need to be declared in the outermost scope. Consequently object instances could not be dynamically created.

Alternatively the fields could be separated from the methods. This involves the definition of two records. One of these, *databag*, holds the object's fields and the other, *procbag*, holds the object's methods:

$$
\begin{array}{l}
|[\ \textbf{var}\ databag : DATABAGTYPE \bullet \\
\quad databag := databag(bound = Bound, bagdata = [\![\,]\!]); \\
\quad |[\ \textbf{var}\ procbag : PROCBAGTYPE \bullet \\
\qquad procbag := procbag(\\
\qquad\quad put = \left(\begin{array}{l} \textbf{pro}\ el : ELEMENTS,\ \textbf{var}\ databag : DATABAGTYPE \bullet \\ databag.bagdata\textbf{:}\left[\dfrac{\#databag.bagdata < databag.bound}{databag.bagdata = databag_0.bagdata \uplus el} \right] \end{array} \right) \\
\qquad\quad get = \left(\begin{array}{l} \textbf{pro var}\ el : ELEMENTS,\ databag : DATABAGTYPE \bullet \\ databag.bagdata, \\ \qquad el \end{array} \textbf{:}\left[\dfrac{databag.bagdata \neq [\![\,]\!]}{databag_0.bagdata = databag.bagdata \uplus el} \right] \right) \\
\qquad\quad card = \left(\begin{array}{l} \textbf{pro var}\ num : \mathbb{N}, databag : DATABAGTYPE \bullet \\ num\textbf{:}\left[num = \#databag.bagdata \right] \end{array} \right) \\
\qquad) \\
\quad]| \\
]|
\end{array}
$$

where *DATABAGTYPE* provides the appropriate typings for *bound* and *bagdata*, and *PROCDATATYPE* provides the correct types for *put*, *get* and *card*. Here, the data portion is passed to each procedure as a value-result parameter. This approach does not permit subtyping as the $(call)$ rule only allows value parameters to be subtyped, disallowing the use of subtype instances for the value-result parameters. This could be remedied if the $(call)$ rule were weakened to permit value-result parameter types to be subtyped as well. However, Naumann suggests that this would lead to "complicating the semantics in an operationally unnatural way."

3.4 Miscellaneous Approaches

This section presents a summary of the work of other researchers investigating object-oriented verification and refinement logics.

Earlier work by Sekerinski [Sek96] investigated the use of packed record types in the formation of an object-oriented refinement calculus framework. Packed record types consist of two records, one for methods and one for fields. The records are then 'packed'[6] so that the fields are hidden and the methods can use field records that are possibly extended. The approach was abandoned as the invocation of methods involved complex unpacking and repacking of objects.

Lewerentz et al. [LLRS95] use the classical lattice-theoretic refinement calculus as the foundation for a logic in which lattices of classes can be formed. Then, given two classes, they show that a superclass and subclass of these classes can be calculated by using the lattice concepts of meet and join.

Leino [Lei95] used Dijkstra's Guarded Command Language as the foundation for an object-oriented verificational logic. Leino [Lei97] subsequently used a weakest liberal precondition semantics to provide an axiomatic semantics for an object-oriented programming language called Ecstatic. A weakest liberal precondition semantics is similar to a weakest precondition semantics except that termination is not considered.

Abadi and Leino [AL97] use Hoare logic to provide a wide-spectrum, object-based programming language and verificational logic.

a, b	$::=$	x	variables
	$\mid$	$false \mid true$	constants
	$\mid$	$if\ x\ then\ a_0\ else\ a_1$	conditional
	$\mid$	$let\ x = a\ in\ b$	let
	$\mid$	$[f_i = x_i^{i \in 1..n}, m_j = \varsigma(y_j)\ b_j^{j \in 1..m}]$	object construction
	$\mid$	$x.f$	field selection
	$\mid$	$x.m$	method invocation
	$\mid$	$x.f := y$	field update

[6] Using existential quantification of types as discussed by Abadi and Cardelli [AC96].

The language is typed and permits subtyping, subsumption and inheritance. An *operational semantics* is provided and a soundness theorem is given that guarantees an outcome of a program derived using the *axiomatic semantics* is the outcome achieved using the operational semantics.

The language above is extended with specifications that are mutations of Hoare triples. The Hoare triple

$$\{pre\}s\{post\}$$

is represented using the *transition relation*

$$s : S :: pre \Rightarrow post$$

where S is the type of s. Within Hoare triples, s is a statement. In transition relations, s may also be a constant, variable or expression. Using transition relations, a Hoare-like logic is developed for the language. Like Hoare, the resulting axiomatic semantics guarantees only partial correctness. Verification can be performed using the axiomatic semantics. Unfortunately, the axiomatic semantics is not complete; hence many verifications cannot be performed despite the correctness of the program.

While Abadi and Leino mutate Hoare triples into transition relations, de Figueiredo [dF95] extends Hoare triples by including a result condition in the traditional triple. The specification

$$\{p\}c\{q,t\}$$

denotes p as the precondition, c as the programming language command, q as the post-condition and t as the condition whose interpretation denotes the value returned. The inclusion of the t result is motivated by the fact that the commands of object-oriented languages can modify a state (hence p and q) and also return a value (hence t). The Hoare triple extension provides the foundation for an object-oriented verificational calculus.

Ahmed and Morris [AM91] use Martin-Löf's type theory to define a module refinement calculus. They introduce propositional expressions into types to allow a module's type to be its specification. This "allows us to specify the detailed behaviour of objects purely by their type." Unfortunately, the authors admit the refinement definition is not adequate and can not refine away some intermediary constructs introduced during refinement. This means that many intuitive refinements can not be proven.

3.5 Conclusions

The theoretical foundations of existing object-oriented refinement calculi vary significantly. Additionally, the pervasive effects of object representation means that the resulting object-oriented refinement calculi semantics are so divergent that it is difficult to compare

the calculi except on a feature-by-feature comparison. The divergent nature of the foundations also makes it difficult to translate properties and techniques from one approach to another. Even the syntax of related concepts varies significantly. This problem is glossed over in this chapter as the syntax has been made more consistent.

The majority of the approaches summarised here decouple fields from methods to simplify the typings of objects. This thesis (§4.4) takes the alternative approach and incorporates methods into objects. This approach is intended to simplify the semantics for late binding and to capture the inherent encapsulation principles of object-orientation.

The complications that decoupling fields from methods introduces are indicated by the need to 'compile' the late binding mechanism. For Utting's [Utt92] function, the introduction of a new class requires the 'recompilation' of the function. The loss of encapsulation in Mikhajlova and Sekerinski's [MS97] sum types approach is illustrated by the fact that classes must be aware of all their subclasses. The semantics of a class' method is determined by summing all corresponding methods from all classes in the class hierarchy. Consequently adding a new subclass requires 'recompilation' of all superclasses. While there are no known practical side effects, it is disconcerting that a class must be aware of its improved subclasses, that its semantics are altered by the introduction of a new class, and that 'recompilation' of superclasses of the new class is, at least in theory, required.

The slight complication of Utting's late binding mechanism, is outweighed however, when its flexibility is considered. Not only does it handle multiple inheritance but it can also be considered an abstraction of the late binding mechanism that is employed by all approaches that decouple fields from methods. For example, given that the sum type approach is analogous to dynamic type checking [BB95, p5] it is a special case of Utting's function where the expressions are type checks of the host.

Chapter 4

Basis for An Object-Oriented Refinement Calculus

4.1 Introduction

The aim of this chapter is to present a framework for an object-oriented refinement calculus. As a consequence, a wide spectrum, object-oriented language is developed here. The highlights of this chapter are:

- a novel definition of refinement that handles heterogeneously typed statements;
- a unique object model that handles embedded methods;
- object history constraints that allow dual-state behavioural constraints; and
- object specifications which are an extension of specifications that permits fine grained attribute control.

Like the object-oriented refinement calculi of Utting and Robinson [UR92, Utt92], Mikhajlova and Sekerinski [MS97], and Calvanti and Naumann [CN00], the framework is based on a lattice-theoretic foundation. The lattice-theoretic model is developed from the Booleans. The Booleans, $\{\mathbf{false}, \mathbf{true}\}$, form a trivial *complete Boolean lattice* under the ordering $\mathbf{false} \Rightarrow \mathbf{true}$. Many of the lattice properties are extended to predicates and predicate transformers by pointwise extension as described by Back and von Wright [BvW98, p128,p190].

This foundation is complemented with the object types introduced in Section 2.2. To allow for the composition of objects, Section 4.2 introduces the concepts of meet and join for object types. Section 4.3 provides the type models for states, predicates and predicate transformers. These models are based on classical techniques, yet the definitions are modified to allow for both typing and subtyping. While typing and subtyping rules have been provided before, to the author's knowledge, this presentation is the first to explore the definitions of heterogeneously typed connectives. Section 4.3 also introduces a novel definition that permits refinement of statements of heterogeneous types. Section 4.4 presents

a novel model of objects that permits methods to be embedded. Section 4.5 introduces an abstract object syntax, a technique that uses subtyping to enforce the encapsulation of object attributes, and object invariants and object history properties. Section 4.6 provides semantics for the client constructs used to access and/or modify objects, e.g., field selections, field updates, method invocations and object specifications. Object specifications are an innovative language feature that adapt specifications for use in an object-oriented refinement calculus. The frames of classical specifications consist of variables. This level of modification control is too coarse for dealing with objects. Object specifications use fine grained *field paths* to apply appropriate constraints, allowing modification control of individual attributes.

Most existing object-oriented refinement calculi are defined using a semantics for values. This chapter, up to Section 4.7, presents a framework for such a semantics. In contrast, practical object-oriented languages use a semantics for references. To incorporate such practicalities, Section 4.7 presents a model for supporting object references (or identities) and introduces new constructs that are the 'reference' analogies of the 'value' constructs of Section 4.6. The semantics for references is based on the multiple stores model described by Utting [Utt95] and Bancroft [Ban97].

4.2 Least-Upper and Greatest-Lower Bounds

The definitions for the greatest lower bound and least upper bound of collections of types are not part of the object calculus developed by Abadi and Cardelli (introduced in Chapter 2). They are developed here for determining the type of predicate and predicate transformer connectives (as given in Section 4.3.1).

Definition 4.1 (Greatest Lower Bound) For types α and β the greatest lower bound is the 'greatest' type that is a subtype of both.

$$\frac{\vdash \alpha \quad \vdash \beta \quad \vdash (\alpha \sqcap \beta)}{\vdash (\alpha \sqcap \beta) \preceq \alpha \quad \vdash (\alpha \sqcap \beta) \preceq \beta \quad \vdash \forall \gamma \bullet (\gamma \preceq \alpha \land \gamma \preceq \beta) \Rightarrow \gamma \preceq (\alpha \sqcap \beta)}$$

$\diamond$

The antecedents ensure that the types α, β and $(\alpha \sqcap \beta)$ are well-formed. The consequent $\forall \gamma \bullet (\gamma \preceq \alpha \land \gamma \preceq \beta) \Rightarrow \gamma \preceq (\alpha \sqcap \beta)$ denotes that for all lower bounds (γ) of α and β, the greatest lower bound is greater (or equal): $\gamma \preceq (\alpha \sqcap \beta)$.

The syntactic calculation of the greatest lower bound of two object types is obtained by combining the identifiers of both. For disjoint types, the greatest lower bound can be determined by:

$$(\mathit{Objtype} \left\{ \ \alpha \ \right\} \sqcap \mathit{Objtype} \left\{ \ \beta \ \right\}) \equiv \mathit{Objtype} \left\{ \ \alpha \cup \beta \ \right\}$$

$\cup$ is used in this context in a syntactic fashion to denote the explicit construction of an object type.

When α and β are not disjoint the greatest lower bound of those attributes that are shared must be calculated. This is determined by the pointwise calculation of the greatest lower bounds of the corresponding attributes, provided that each of these is well-formed.

$$\textit{for all } (i \in \operatorname{dom}\alpha \cap \operatorname{dom}\beta) \bullet \vdash (\alpha(i) \sqcap \beta(i))$$

where $\operatorname{dom}\alpha$ denotes the set of identifiers in α.

The identifier-type mappings for identifiers only in α are obtained using $(\operatorname{dom}\beta \lhd \alpha)$. Similarly the identifier-type mappings for identifiers only in β are obtained using $(\operatorname{dom}\alpha \lhd \beta)$. Adding these together forms the greatest lower bound:

$$\frac{\textit{for all } (i \in (\operatorname{dom}\alpha \cap \operatorname{dom}\beta)) \bullet \vdash (\alpha(i) \sqcap \beta(i))}{(\alpha \sqcap \beta) \equiv \textit{Objtype} \left\{ \; \{i \in (\operatorname{dom}\alpha \cap \operatorname{dom}\beta) \bullet i \mapsto (\alpha(i) \sqcap \beta(i))\} \cup (\operatorname{dom}\alpha \lhd \beta) \cup (\operatorname{dom}\beta \lhd \alpha) \; \right\}}$$

The premise ensures that the greatest lower bound for each pair of associated types is well-formed.

The greatest lower bound of basic types is undefined. For example $(\mathbb{B} \sqcap \mathbb{N})$ is undefined.

It may appear counterintuitive that the syntax '$\sqcap$' is used for unioning identifiers in object types. However, the following example illustrates that it is semantically intuitive even if syntactically it is not.

Example 4.2 (Meet Syntax) For example, the object type $\textit{Objtype} \left\{ \; a : \mathbb{B} \; \right\}$ contains all objects that have a field *a* of type $\mathbb{B}$. Similarly, the object type $\textit{Objtype} \left\{ \; b : \mathbb{B} \; \right\}$ contains all objects that possess a field *b* of type $\mathbb{B}$. Using the greatest lower bound, a subtype of both can be calculated: $\textit{Objtype} \left\{ \; a : \mathbb{B}, b : \mathbb{B} \; \right\}$. This object type contains all objects that possess a field *a* and a field *b*, both of which are of type $\mathbb{B}$. Therefore, all objects in the type $\textit{Objtype} \left\{ \; a : \mathbb{B}, b : \mathbb{B} \; \right\}$ are in $\textit{Objtype} \left\{ \; a : \mathbb{B} \; \right\}$. The calculation of the greatest lower bound of two object types is actually an intersection of the sets of objects that form the types, even though syntactically the fields are unioned.

$$\begin{array}{l} \textit{Objtype} \left\{ \; a : \mathbb{B}, b : \mathbb{B} \; \right\} \\ \equiv \\ \textit{Objtype} \left\{ \; a : \mathbb{B} \; \right\} \sqcap \textit{Objtype} \left\{ \; b : \mathbb{B} \; \right\} \end{array}$$

$\diamond$

Using $\sqcap$ with a single non-empty set argument denotes generalised greatest lower bound on that set. The generalised greatest lower bound of an empty set is $\textit{Objtype}\,\{\}$. $\textit{Objtype}\,\{\}$ contains all objects.

Definition 4.3 (Least Upper Bound) Given object types α and β the least upper bound of α and β is the 'least' type that is a supertype of both.

$$\frac{\vdash \alpha \quad \vdash \beta}{\vdash \alpha \preceq (\alpha \sqcup \beta) \quad \vdash \beta \preceq (\alpha \sqcup \beta) \quad \vdash \forall \gamma \bullet (\alpha \preceq \gamma \land \beta \preceq \gamma) \Rightarrow (\alpha \sqcup \beta) \preceq \gamma}$$

$\diamond$ It is syntactically calculated by taking those identifiers common to both α and β. For each common identifier, the resulting type is the least upper bound of the associated types of the identifier.

$$(\alpha \sqcup \beta) \equiv \mathit{Objtype}\left\{\ i \in (\operatorname{dom}\alpha \cap \operatorname{dom}\beta) \bullet i \mapsto (\alpha(i) \sqcup \beta(i))\ \right\}$$

The least upper bound of basic types is $\mathit{Objtype}\,\{\}$. For example $(\mathbb{B} \sqcup \mathbb{N})$ is $\mathit{Objtype}\,\{\}$.

Using $\bigsqcup$ with a single set argument denotes generalised least upper bound on that set. The generalised least upper bound of the empty set is undefined.

4.3 Predicate Transformer Foundation

This section provides models for states, predicates, and predicate transformers (Sections 4.3.1, 4.3.2, and 4.3.3 respectively). The connectives for predicates and predicate transformers permit the combination of heterogeneously typed predicates and predicate transformers, respectively. Section 4.3.4 explores the effects of subsumption on the monotonic subclass of predicate transformers: *statements*. Finally, Section 4.3.5 presents a novel definition of refinement that permits the refinement of heterogeneously typed statements.

4.3.1 State Modelling

This section provides a generic model for states and identifies the subtyping behaviour required of states. States are collections of values in which each value is associated with an identifier. A state type is a collection of identifiers, each of which is associated with a type. The syntax for construction of state types is set notation subscripted with $\mathbb{ST}$. Given identifiers $i_1, ..., i_n$ and types $T_1, ..., T_n$ the following asserts a well-formed state type.

$$\frac{\mathit{for\ all}\ j \in 1..n \bullet \vdash T_j \quad \mathit{distinct}\ i_j}{\vdash \{j \in 1..n \bullet i_j \mapsto T_j\}_{\mathbb{ST}}}$$

State instances are explicitly described using set notations subscripted with $\mathbb{S}$. The state $\{j \in 1..n \bullet i_j \mapsto o_j\}_{\mathbb{S}}$ is a well formed state that has type $\{j \in 1..n \bullet i_j \mapsto T_j\}_{\mathbb{ST}}$ provided its attributes are well-formed:

$$\frac{\mathit{for\ all}\ j \in 1..n \bullet o_j \in T_j \quad \vdash \{j \in 1..n \bullet i_j \mapsto T_j\}_{\mathbb{ST}}}{\{j \in 1..n \bullet i_j \mapsto o_j\}_{\mathbb{S}} : \{j \in 1..n \bullet i_j \mapsto T_j\}_{\mathbb{ST}}}$$

The greatest state type, $\top_{\mathbb{ST}}$, is a state type which possesses no identifiers. Every state instance is, via subsumption, a member of the greatest state type.

States subtype so that 'larger' states (states with more identifiers) are subtypes of 'smaller' states, and the types of state elements may vary covariantly (or monotonically) provided the types of 'new' state components are well-formed. These requirements are expressed formally in Desideratum 4.4.

Desideratum 4.4 (Sub State Type) Given an environment including state types α and β:

$$\frac{\vdash \operatorname{dom}\beta \subseteq \operatorname{dom}\alpha \quad \textit{for all } i \in \operatorname{dom}\beta \bullet \vdash \alpha(i) \preceq \beta(i) \quad \textit{for all } i \in \operatorname{dom}(\beta \lhd \alpha) \bullet \vdash \alpha(i)}{\vdash \alpha \preceq \beta}$$

This property is consistent with that used by Sekerinski [Sek96].

Having stated the desired properties for state types, a model that upholds these properties should be chosen. One possible approach is to model states as objects. However, such a model (using the objects introduced in Section 2.2) is not consistent with the stated requirements as the attributes of objects, as they are modelled, are invariantly typed. Desideratum 4.4, however, permits the attributes of states to vary covariantly. Extensible, covariantly annotated object types would be a sufficient model for state types. However, records (which are extensible and whose components subtype covariantly) form a recognisable type and are therefore suggested as an alternative, simpler model. Desideratum 4.4 is shown to hold by Proof B.1 using the definition of states: Definition 4.5.

Definition 4.5 (States as Records)

$$\{j \in 1..n \bullet i_j \mapsto OT_j\}_{\boxed{ST}} \mathrel{\widehat{=}} [j \in 1..n \bullet i_j \mapsto OT_j]$$

The syntax $[j \in 1..n \bullet i_j \mapsto OT_j]$ denotes a record type as defined by Abadi and Cardelli [AC96] and summarised by the laws in Section A.2.
◇

States can be constructed, extended, updated, or components selected using the corresponding record operation upon the state. The notations for the state operations are consistent with those for record operations. State cut is equivalent to record cut—it removes an element from a state. For example:

$$\{a \mapsto v, b \mapsto w\}_{\boxed{S}} \setminus_{\boxed{S}} \{a\} \equiv \{b \mapsto w\}_{\boxed{S}}$$

State union is defined as record override on the proviso that states have an empty syntactic intersection. For example, since $\{a \mapsto v\}_{\boxed{S}}$ and $\{b \mapsto w\}_{\boxed{S}}$ have no identifiers in common, their state union is justified and is calculated using record override:

$$\{a \mapsto v\}_{\boxed{S}} \cup_{\boxed{S}} \{b \mapsto w\}_{\boxed{S}} \equiv \{a \mapsto v\}_{\boxed{S}} \oplus \{b \mapsto w\}_{\boxed{S}} \equiv \{a \mapsto v, b \mapsto w\}_{\boxed{S}}$$

The object calculus provides an equivalence relation $=_\alpha$ for comparing terms under any given type α. The relation has the property that any two terms equivalent under one type are also equivalent under a supertype.

The greater lower bound and least upper bound of states can be syntactically calculated in a similar manner to that used for object types.

Subsumption of States Like objects, a state does not alter when it is subsumed. Instead, the static information known about which identifiers are in a state is reduced. Subsumption of states means that a ('larger') state can be used in a situation where a ('smaller') state with a subset of the former's identifiers is required. For example, a function which takes states with the identifier *a* can also take a state with identifiers *a* and *b*—the function disregards knowledge about *b*. Analogously, if a state function returns a state with identifiers *a* and *b*, that same state can be regarded as a state containing only *a*—knowledge of *b* has been disregarded.

4.3.2 Predicates

This section provides a definition of predicates and also provides novel definitions of connectives for joining heterogeneously typed predicates.

Definition 4.6 (Predicates) Predicates are defined as functions from states to booleans. Given state type α then

$$\mathit{Pred}\ \alpha \mathrel{\widehat=} \alpha \to \mathbb{B}$$

For well-formed state type α, the predicate type $\mathit{Pred}\ \alpha$ is well-formed.

$$\frac{\vdash \alpha}{\vdash \mathit{Pred}\ \alpha}$$

◇

The predicate type on the greatest state type, $\mathrm{pred}\ \top_{\boxed{\mathrm{ST}}}$, contains only two predicates, *True* and *False* (see Definition 4.8). *True* and *False* are also predicates on 'longer' state types. The following example illustrates this idea further. The predicate $a = 1$ on predicate type $\mathrm{pred}\ \{a \mapsto \mathbb{N}\}_{\boxed{\mathrm{ST}}}$ constrains *a* to 1. By subsumption, the predicate $a = 1$ also types as $\mathrm{pred}\ \{a \mapsto \mathbb{N}, b \mapsto \mathbb{N}\}_{\boxed{\mathrm{ST}}}$. In the subsumed context, the predicate $a = 1$ constrains *b* to be a natural number. Again, by subsumption, the original predicate $(a = 1)$ also types as $\mathrm{pred}\ \{a \mapsto \mathbb{N}, b \mapsto \mathbb{Z}\}_{\boxed{\mathrm{ST}}}$ thereby constraining *b* to be an integer. The unsubsumed predicate $a = 1$ on type $\mathrm{pred}\ \{a \mapsto \mathbb{N}\}_{\boxed{\mathrm{ST}}}$ is a predicate that constrains *a* to 1 yet allows *b* (and all other identifiers) to be of any value and type.

Theorem 4.7 (Sub Predicate) **Proof on page 167**

The subtyping rule for predicates is

$$\frac{\vdash \alpha \preceq \beta}{\vdash \mathit{Pred}\ \beta \preceq \mathit{Pred}\ \alpha}$$

That is, predicates subtype when the states on which they are defined vary contravariantly.

Definition 4.8 (Predicate Falsity) The falsity predicate is defined as the lifted version of the Boolean false.

$$\mathit{False} \mathrel{\widehat{=}} \lambda s : \top_{\boxed{\mathrm{ST}}} \bullet \mathbf{false}$$

The domain of *False* is the greatest state type. All state types are subtypes of the greatest state type. The range of *False* is simply the Boolean false value. Consequently, this is a function that maps all states to **false**.
◇

Postulate 4.9 (Val Falsity)
For every well-formed state type α, *False* can be subsumed to a predicate on that state type.

$$\frac{\vdash \alpha}{\vdash \mathit{False} : \mathit{Pred}\ \alpha}$$

The truth predicate *True* can be constructed analogously.

The Boolean logical connectives, $\Rightarrow$, $\wedge$, $\vee$, $\neg$ can be used for pointwise extensions on states to define corresponding predicate connectives. The definition of predicate implication is now provided.

Definition 4.10 (Predicate Implication) Given state types α and β, predicate implication is defined as the lifting of boolean implication on the greatest lower bounds of these types.

$$\frac{\vdash p_1 : \mathit{Pred}\ \alpha \quad \vdash p_2 : \mathit{Pred}\ \beta \quad \vdash (\alpha \sqcap \beta)}{p_1 \Rightarrow p_2 \mathrel{\widehat{=}} \lambda s : (\alpha \sqcap \beta) \bullet p_1(s) \Rightarrow p_2(s)}$$

◇

For example, given $(a = 1) : \mathrm{pred}\ \{a \mapsto \mathbb{N}\}_{\boxed{\mathrm{ST}}}$ and $(b = 1) : \mathrm{pred}\ \{b \mapsto \mathbb{N}\}_{\boxed{\mathrm{ST}}}$, then

$$a = 1 \Rightarrow b = 1$$

is a predicate on the type $\mathrm{pred}\ \{a \mapsto \mathbb{N}, b \mapsto \mathbb{N}\}_{\boxed{\mathrm{ST}}}$. The definitions of predicate conjunction and predicate disjunction are analogous.

Predicate validity is used to determine whether or not the predicate is true for all states on which it is defined.

Definition 4.11 (Validity) Given state type α, and predicate *p*, then validity is defined as:

$$\frac{\vdash \alpha \qquad \vdash p : \mathrm{pred}\ \alpha}{\lceil p \rceil_{\alpha} \mathrel{\widehat{=}} \forall s : \alpha \bullet p(s)}$$

◇

Definition 4.12 (Entailment) Entailment is the validation of predicate implication. For predicates p and q on state types α and β respectively, entailment is defined as:

$$\frac{\vdash p : \mathit{Pred}\ \alpha \quad \vdash q : \mathit{Pred}\ \beta \quad \vdash (\alpha \sqcap \beta)}{p \Rrightarrow q \ \widehat{=}\ \lceil p \Rightarrow q \rceil_{(\alpha \sqcap \beta)}}$$

◇

4.3.3 Predicate Transformers

This section defines predicate transformers and explores their subtyping behaviours. Predicate transformers are defined as functions from postcondition predicates to precondition predicates.

Definition 4.13 (Predicate Transformers) For state types *Post* and *Pre*:

$$\mathit{Ptrans}\ \mathit{Post}\ \mathit{Pre} \ \widehat{=}\ \mathit{Pred}\ \mathit{Post} \rightarrow \mathit{Pred}\ \mathit{Pre}$$

◇

Theorem 4.14 (Sub Predicate Transformers) **Proof on page 168**

Predicate transformers subtype when the postcondition state varies covariantly and the precondition state varies contravariantly.

$$\frac{\vdash \alpha \preceq \beta \quad \vdash \gamma \preceq \delta}{\vdash \mathit{Ptrans}\ \alpha\ \delta \preceq \mathit{Ptrans}\ \beta\ \gamma}$$

as $\mathit{Pred}\ \beta \preceq \mathit{Pred}\ \alpha$ and $\mathit{Pred}\ \delta \preceq \mathit{Pred}\ \gamma$.

For instance, the predicate transformer $a, b := 1, 1$ guarantees postcondition $a = 1 \land b = 1$ with a precondition *True* on a state containing elements a and b (pred $\{a, b : \mathbb{Z}\}_{\mathrm{ST}}$). However, by subsumption the postcondition's type can be given as a predicate on the state including only a, that is, pred $\{a : \mathbb{Z}\}_{\mathrm{ST}}$. Having lost information about the type of b, the knowledge about the postcondition, under subsumption, is weakened to $a = 1$. In a similar fashion, under subsumption the predicate transformer may guarantee $a = 1 \land b = 1$ for a precondition *True* on a state type including a, b and $c : \mathbb{Z}$ (pred $\{a, b, c : \mathbb{Z}\}_{\mathrm{ST}}$). Information about the original predicate transformer has been lost by this subsumption as the original predicate transformer could have been alternatively subsumed to work on a precondition state type where c was a Boolean (pred $\{a, b : \mathbb{Z}, c : \mathbb{B}\}_{\mathrm{ST}}$).

4.3.4 Statements

Statements are monotonic predicate transformers. The abort statement and demonic choice operators are examined here to illustrate the effects of typing the predicate transformers. The remaining statements and operators used in this thesis are listed in Section A.4.

Definition 4.15 (Abort) The abort predicate transformer always returns the predicate *False*. Typing affects the abort statement by forcing it to be parameterised on the post-condition state—forming a family of abort statements. This parameter is omitted when it is obvious from the context.

$$\mathbf{abort}_\alpha \mathrel{\widehat=} \lambda\, p : \mathit{Pred}\ \alpha \bullet \mathit{False}$$

◇

Example 4.16 (Subsuming an Abort) The difference between members of the abort family can be distinguished by an example in which one is subsumed. For instance, $\mathbf{abort}_{\{a \mapsto \mathbb{N}, b \mapsto \mathbb{N}\}_{\mathrm{ST}}}$ is a predicate transformer that maps all predicates of type $\mathrm{pred}\ \{a \mapsto \mathbb{N}, b \mapsto \mathbb{N}\}_{\mathrm{ST}}$ (and through subsumption, predicates of types $\mathrm{pred}\ \{a \mapsto \mathbb{N}\}_{\mathrm{ST}}$, $\mathrm{pred}\ \{b \mapsto \mathbb{N}\}_{\mathrm{ST}}$ and $\mathrm{pred}\ \top_{\mathrm{ST}}$) to the predicate *False*. However, $\mathbf{abort}_{\{a \mapsto \mathbb{N}\}_{\mathrm{ST}}}$ only maps predicates of types $\mathrm{pred}\ \{a \mapsto \mathbb{N}\}_{\mathrm{ST}}$ and $\mathrm{pred}\ \top_{\mathrm{ST}}$ to *False*. Hence, $\mathbf{abort}_{\{a \mapsto \mathbb{N}, b \mapsto \mathbb{N}\}_{\mathrm{ST}}}$ can be used wherever $\mathbf{abort}_{\{a \mapsto \mathbb{N}\}_{\mathrm{ST}}}$ is used and it will produce the same effects but not vice versa. For $PT \mathrel{\widehat=} \mathit{Ptrans}\ \{a \mapsto \mathbb{N}\}_{\mathrm{ST}}\ \top_{\mathrm{ST}}$

◇ $\mathbf{abort}_{\{a \mapsto \mathbb{N}, b \mapsto \mathbb{N}\}_{\mathrm{ST}}} =_{PT} \mathbf{abort}_{\{a \mapsto \mathbb{N}\}_{\mathrm{ST}}}$

Postulate 4.17 (Val Abort)
Given state type α, the abort predicate transformer types as $\mathit{Ptrans}\ \alpha\ \top_{\mathrm{ST}}$.

$$\frac{\vdash \alpha}{\vdash \mathbf{abort}_\alpha : \mathit{Ptrans}\ \alpha\ \top_{\mathrm{ST}}}$$

Given state type α, the magic predicate transformer $\mathbf{magic}_\alpha$ can be constructed in an analogous manner.

Definition 4.18 (Demonic choice) Demonic choice (predicate transformer meet) is the lifting of predicate conjunction. Given predicate transformers pt_1 and pt_2, demonic choice is defined when the greatest lower bound of the precondition state types of pt_1 and pt_2 is well-formed. This restriction ensures that all weakest preconditions of pt_1 can be conjuncted with those of pt_2. Informally, the types of pt_1 and pt_2 are termed *compatible* if their greatest lower bound exists.

$$\frac{\vdash pt_1 : \mathit{Ptrans}\ \alpha\ \delta \quad \vdash pt_2 : \mathit{Ptrans}\ \beta\ \gamma \quad \vdash (\delta \sqcap \gamma)}{pt_1 \sqcap pt_2 \mathrel{\widehat=} \lambda\, p : \mathit{Pred}\ (\alpha \sqcup \beta) \bullet pt_1\ p \wedge pt_2\ p}$$

◇

Postulate 4.19 (Val Demonic Choice)
Demonic choice types as a predicate transformer from the join of the state types of the postconditions to the meet of the state types of the preconditions provided the meet is well-formed.

$$\frac{\vdash pt_1 : \mathit{Ptrans}\ \alpha\ \delta \quad \vdash pt_2 : \mathit{Ptrans}\ \beta\ \gamma \quad \vdash (\delta \sqcap \gamma)}{\vdash (pt_1 \sqcap pt_2) : \mathit{Ptrans}\ (\alpha \sqcup \beta)\ (\delta \sqcap \gamma)}$$

4.3.5 Statement Refinements

Refinement, the predicate transformer ordering relation, is defined via predicate entailment. Classically, the refinement relation is used to 'compare' two statements of the same type. Following is an innovative refinement relation that allows 'comparison' between statements of heterogeneous types.

Definition 4.20 (Refinement) Given predicate transformers pt_1 and pt_2, refinement between them is defined as:

$$\frac{\vdash pt_1 :_\bot \mathit{Ptrans}\ \alpha\ \delta \quad \vdash pt_2 : \mathit{Ptrans}\ \beta\ \gamma \quad \vdash \beta \preceq \alpha \quad \vdash (\delta \sqcap \gamma)}{pt_1 \sqsubseteq pt_2 \ \widehat{=}\ \forall p : \mathit{Pred}\ \alpha \bullet pt_1\ p \Rrightarrow pt_2\ p}$$

The antecedents $\beta \preceq \alpha$ and $(\delta \sqcap \gamma)$ are the minimum typing requirements for the definition to be well-formed. Since the goal of using refinement is to replace the original code with the refined code the definition may be more general than is required: the subtyping of $\tau(pt_2) \preceq \tau(pt_1)$, that is, $\beta \preceq \alpha \wedge \delta \preceq \gamma$, may be sufficient.

The 'least type' antecedent

$$pt_1 :_\bot \mathit{Ptrans}\ \alpha\ \delta$$

ensures that *Ptrans* $\alpha\ \delta$ is the declaration type of pt_1 and not a type obtained by subsumption. That is, there is no subtype of *Ptrans* $\alpha\ \delta$ that pt_1 is an element of. The least type property is static as opposed to a type check which is evaluated dynamically. It is used to constrain the declaration type of an object, not to regain attributes lost due to subsumption. The semantics of least types are provided in Section 2.2.

The reason the least type constraint is applied is to stop the subsumption of pt_1 to, for example, the type *Ptrans* $\top_{\mathbb{SET}}\ \delta$. If this were allowed, refinement could be reduced to the following:

$$pt_1 \sqsubseteq pt_2 \ \widehat{=}\ pt_1(\mathit{true}) \Rrightarrow pt_2(\mathit{true}) \wedge pt_1(\mathit{false}) \Rrightarrow pt_2(\mathit{false})$$

as the predicate on $\top_{\mathbb{SET}}$ only contains *true* and *false*. This definition would allow the refinement of, for example, $a := 2$ to $a := 3$.

◇

Example 4.21 (Heterogeneously Typed Statement Refinement) The definition of refinement allows the use of statements that modify variables outside the state to act in lieu of statements that only modify state variables. For example, the statement $a, b, c := a + 1, b + 1, 1$ is a refinement of the statement $a, b := a + 1, b + 1$. Similarly the statement $a, b, c := a + 1, b + 1,$ *false* is also a refinement. Intuitively, these statements are refinements. Since c is not in the environment the correctness of the program does not depend upon the value of c. Consequently, c can be arbitrarily modified without violating

the program's properties. This is an *open world* view[1] [Mah99, p90]. In an open world, variables outside the state may be modified. In a *closed world* variables outside the state may not be modified.
◇

Theorem 4.22 (Open World Specification) **Proof on page 171**
Using the open world view of the refinement relation, the frame of a specification can be expanded with variables outside the specification's environment. The specification

$$\vec{z}: \left[pre \ , \ post \right]$$

in an environment with state variables $\vec{z}$, disjoint from $\vec{w}$:

$$\vec{z}: \left[pre \ , \ post \right] \sqsubseteq \vec{z}, \vec{w}: \left[pre \ , \ post \right]$$

Since this frame extension has the effect of enlarging the specification's environment, to use the specification in lieu of the original requires reducing its environment back to that of the original specification. This can be achieved by encapsulating it with a variable block scope. For variables $\vec{w}$ of type $\vec{W}$:

$$\left| \left[\mathbf{var}\ w : W \bullet \right. \right. \\ \left. \left. \quad \vec{z}, \vec{w}: \left[pre \ , \ post \right] \right] \right|$$

In fact, this is the Introduce Local Variable Block (A.55) rule.

[1] The phrase open world view was coined by Utting[Utt92].

4.4 An Object Representation

This section presents an original model for an object representation that has embedded methods and is based on an object calculus. It partially supports binary methods and the ability to add methods to subclasses, while maintaining a general rule of subsumption. Late binding can be modelled with this object representation by invocation of the embedded method. Section 4.4.1 motivates the object representation by identifying the necessary typing behaviour of objects with embedded methods and the lack of success using λ-calculus-based semantics. Section 4.4.2 introduces the object calculus concepts used to construct the model. Based on these concepts, Section 4.4.3 presents several object representation models. Some of these models are not sufficient as they do not uphold the typing and subtyping properties discussed in Section 4.4.1. The reasons for these violations are discussed. The final model, however, is proven to be sufficient.

4.4.1 Motivation

Choosing an object representation is fundamental to the semantics of an object-oriented refinement calculus. Chapter 3 presents several examples of object representations in existing object-oriented refinement calculi. To simplify the typing system the majority of those approaches have object representations that decouple fields from methods. This approach, however, forces a late binding mechanism to be added as an extra feature to the calculus. Examples of this late binding mechanism are the use of sum types [MS97] or Utting's function [Utt92]. This late binding mechanism adds an additional layer of complexity to the semantics of the resulting object-oriented refinement calculus. In contrast, given an appropriate object representation in which methods are embedded, it is possible to model late binding as the execution of the contained method. Given that the sole reason provided for the decoupling of fields from methods is that it simplifies the typing system, it would seem more productive to search for a simple method-embedded object type rather than to complicate the semantics.

Many researchers have investigated the types required for modelling objects. Palsberg and Schwartzbach [PS94] present an introduction to the problems of object typing and the solutions that current object-oriented languages employ. They suggest that the lack of maturity of types for objects stems from the origins of type theoretic research as a discipline of logic—remote from the types required for practical object-oriented programming languages. It is evident that the λ-calculus alone is not sufficient for practical object-oriented language semantics [AC96, Section 6.7, Chapter 18] [PS94, Section 2.6] as it does not have the "desired modelling power" [AC96, p51]. Informally, if object-oriented languages could be reduced to procedural languages then λ-calculi would be sufficient. Since programming in an object-oriented fashion using a procedural language is difficult, it can be argued that λ-calculi are not sufficient.

Example 4.23 (Self-Application Semantics) One example of a λ-calculus-based object-oriented semantics is the self-application semantics. This semantics maps objects to records of functions. Consider a method f that has type $C \to R$ where C is the method's host type and R is the result type of the method. Given subtype D of object C, that is $D \preceq C$, then f is also of type $D \to R$ as f accepts arguments of type D as they have type C by subsumption[2]. By this argument it can be deduced that function types are contravariant in their left argument: $D \preceq C \Rrightarrow C \to R \preceq D \to R$. Unfortunately, the opposite is required to model object types. Given an object of type D, by subsumption, it should also be of type C. So its methods, of type $D \to R$, should, by subsumption, be of type $C \to R$—thus requiring function types to be covariant in their left argument, contradicting the original analysis. Similar problems occur for other encodings of objects using λ-calculi.
$\diamond$

To solve the typing difficulties of object-orientation, Abadi and Cardelli [AC96] investigated object calculi. Amongst other successes, this research provides types for objects with methods that return other objects of the same type (the *self* type). To solve technical problems they eventually [AC96, Chapter 16] abandon standard denotational models and provide an object type that is shown consistent by means of an operational semantics[3].

Despite this extensive work, types for objects that possess binary methods and support subsumption still do not exist. This causes difficulties for the integration of the refinement calculus with the object-oriented programming paradigm. An intuitive, abstract model of (refinement calculus) objects is one in which objects contain predicate transformers as methods that manipulate the host. Since the objects manipulate themselves, a variable of the host type O must occur in both the methods' precondition and postcondition state types. That is, the methods must of the type $Pred\ \alpha\{O\} \to Pred\ \beta\{O\}$ where $\alpha\{O\}$ denotes the occurrence of O in state type α. The use of O in both the precondition and postcondition means that this approach requires its semantics to support binary methods. A model is provided here that solves the binary method problem for the specific case of embedded predicate transformers. The rule for subtyping these objects is provided along with its proof. Certain restrictions are placed on this model to ensure that it is consistent with a standard denotational semantics.

4.4.2 Additional Object Calculus Concepts

This section introduces several object calculus concepts that are used later for the presentation of models for (object-oriented refinement calculus) object representations. These concepts are variance annotations, recursive object types, dynamic type checking and covariant self types.

[2] See the work of Abadi and Cardelli [AC96, p21,p76] for further discussion.
[3] A similar approach is followed by Palsberg and Schwartzbach [PS94, p26].

Variance Annotations

Given object types C and D as introduced in Section 2.2, for D to be a subtype of C ($D \preceq C$) the attributes types must remain invariant (yet D may have more attributes than C). To provide object types with variant attribute types Abadi and Cardelli introduce variance annotations. They introduce three annotations: invariance (^{o}), covariance ($^+$) and contravariance ($^-$). Omitted annotations are, by convention, invariant. An invariance annotation constrains the annotated field type to be invariant (no variance permitted). Annotating an attribute with the covariant syntax $^+$ allows the attribute to subtype covariantly (or monotonically). For example, for types C and D where $D \preceq C$ then

$$\mathit{Objtype}\left\{\ l \mapsto D^+\ \right\} \preceq \mathit{Objtype}\left\{\ l \mapsto C^+\ \right\}$$

To prevent unsoundness[4] in the typing system covariant attributes cannot be updated other than by the host. Annotating an attribute with the contravariant syntax $^-$ allows contravariant (anti-monotonic) subtyping.

$$\mathit{Objtype}\left\{\ l \mapsto C^-\ \right\} \preceq \mathit{Objtype}\left\{\ l \mapsto D^-\ \right\}$$

Contravariant attributes can only be selected from within the host.

Recursive Object Types

With the object types of Section 2.2, it is not possible to type an object with a method that returns an object of its host's type. Recursive types [AC96, Chapter 9], however, provide this facility. The recursive object type $\mu(X)\ [\ C\{X\}\ ]$ is the unique solution to $X = C\{X\}$ where C is the 'desired' object type with recursive references to self. The syntax $C\{X\}$ means that the self type X can occur freely, or unbound, within C. In a similar fashion to unfolding a recursive procedure, the type $\mu(X)\ [\ C\{X\}\ ]$ can be unfolded by replacing the bound X with the body $C\{X\}$ to achieve the type $C[X \setminus \mu(X)\ [\ C\{X\}\ ]]$ where $C[X \setminus Y]$ denotes the syntactic substitution of X with Y within C. The type $\mu(X)\ [\ C\{X\}\ ]$ and its unfolding $C[X \setminus \mu(X)\ [\ C\{X\}\ ]]$ are isomorphic yet not equivalent. Thus instances of the former are not instances of the latter, and vice versa. To relate instances of these types the *fold* and *unfold* constructs are used. For example, given the recursive type $A \mathrel{\widehat{=}} \mu(X)\ [\ C\{X\}\ ]$ and an instance a of that type, then unfolding a produces an isomorphic object of the unfolded type:

$$\mathit{unfold}(a) : C[X \setminus \mu(X)\ [\ C\{X\}\ ]]$$

Conversely, folding an instance of the unfolded type leads to an isomorphic instance of the original type:

$$\mathit{fold}(A, \mathit{unfold}(a)) : A$$

[4] As illustrated by Abadi and Cardelli [AC96, p109].

Dynamic Type Checking

When an object is subsumed the object does not change, however, statically less information is known about it attributes. Even though static typing information may be lost during subsumption, the 'lost' attributes may still be utilised through late binding. In a pure object-oriented language, dynamic dispatch is the only mechanism for accessing attributes 'lost' by subsumption. Most languages, however, provide a facility for regaining direct access to lost attributes by examining the run-time type of objects. This feature is known as the type-case construct. It is also known as dynamic type check and type casting. Abadi and Cardelli introduce the following syntax for dynamic type casing:

$$\textbf{typecase}\ a \mid (x : A)d_1 \mid d_2$$

When a is of dynamic type A, it is bound to x and d_1 is returned, otherwise d_2 is returned.

Dynamic type checks can be methodologically unsound. They can violate object encapsulation, allowing access to private attributes. Dynamic type checks can also reduce the extensibility of the code. When a new subclass is introduced, previous uses of type-cases may need to be extended. This violates the purist principle of object-orientation that the addition of a new class does not require the recoding of existing classes.

These methodological issues require that dynamic type checks be used judiciously. Much of the work on object-oriented types has focused on reducing the need for dynamic type checks. Given the current incapacity for type theoretic research to provide static types to binary methods, dynamic type checking may be unavoidable.

Self Types

Self types [AC96, Chapter 15] have been investigated as part of the effort to reduce the reliance upon dynamic type checks. Consider the following class[5] where an instance of the class type is returned from one of the class' methods.

class C **is**
    var $x : \mathbb{Z} := 0$;
    method $m() : C$ **is** *body* **end**;
end

Given the following subclass D and an instance d of D, then in general it is unsound to type the result of method m of d as D as m may return an arbitrary instance of type C that is not of type D—one, for instance, that does not contain the attribute y.

subclass D **of** C **is**
    var $y : \mathbb{Z} := 0$;
end

However, if m returns the host (or self), perhaps after modifying x, then it would be sound to type the inherited method as D. The type **Self** is introduced to allow this more precise

[5] This example is derived from the work of Abadi and Cardelli [AC96, p23]

typing, and hence reduce the need for subsequent dynamic type checks. The original class would be written with the result types replaced by the type **Self** :

class C **is**
    var $x : \mathbb{Z} := 0$;
    method $m()$: **Self** **is** *body* **end**;
end

No longer can *body* return an arbitrary instance: it must return an object of exactly the same type as the host. Hence, for the (new) subclass D, the type of **Self** for the method m is implicitly specialised.

This example uses self types in a covariant position. Contravariant use of the self type, as exemplified in class E, is unsound for subsumption as discussed by Cook [Coo89].

class E **is**
    var $x : \mathbb{Z} := 0$;
    method $m(other$: **Self** $)$ **is** *body* **end**;
end

4.4.3 Object Typed Models

This section presents an innovative model of an (object-oriented refinement calculus) object type that contains embedded predicate transformers as methods. Using this object type, late binding can be modelled simply by the invocation of the contained predicate transformer. Additionally, since the objects have embedded methods the model is useful for both object-based and class-based refinement calculi.

Embedding methods in objects forces the objects to be recursive as the methods are a component of the entities that they manipulate. If the object type is represented as X then an object representation must be chosen such that methods of type $Ptrans\ \{self : X\}_{ST}\ \{self : X\}_{ST}$[6] can be embedded and can be subsumed appropriately: methods of subtypes subsume to the methods of supertypes. For instance, given an object type with a method m

$$X \mathrel{\widehat{=}} Objtype \left\{ \ m : Ptrans\ \{self : X\}_{ST}\ \{self : X\}_{ST}\ \right\}$$

and the intended 'subtype'

$$Y \mathrel{\widehat{=}} Objtype \left\{ \begin{array}{l} m : Ptrans\ \{self : Y\}_{ST}\ \{self : Y\}_{ST}, \\ n : Ptrans\ \{self : Y\}_{ST}\ \{self : Y\}_{ST} \end{array} \right\}$$

then the type of method m in Y should be a subtype of the type of method m in X, informally:

$$Ptrans\ \{self : Y\}_{ST}\ \{self : Y\}_{ST} \preceq Ptrans\ \{self : X\}_{ST}\ \{self : X\}_{ST}$$

[6] Here the choice of associating the host object with the variable *self* is made.

A model for (object-oriented refinement calculus) object types with embedded methods should display this subtyping behaviour. Choosing a model that conforms to this subtyping behaviour is, however, analogous to solving the binary method problem (for this context) as the occurrence of the 'self' type (X) in the precondition is contravariant. Current research has not found a suitable model for contravariant occurrences of 'self.'

Recursive object types are not a sufficient model for object-oriented refinement calculus object types. Even the covariant occurrence of X in the 'postcondition' portion of the type causes difficulties. Consider the following recursive objects and their types[7]. The object c is a recursive object that has a field y, a method *return_a_value* which returns the y field, and a method *return_host* which returns the host.

$$c \mathrel{\widehat{=}} \mathit{object} \left\{ \; y = 0, \mathit{return\_a\_value} = \varsigma(s : C)\; s.y, \mathit{return\_host} = \varsigma(s : C)\; s \; \right\}$$

where C, the type of c, is defined as:

$$C \mathrel{\widehat{=}} \mu(X) \left[\; y : \mathbb{Z}, \mathit{return\_a\_value} : \mathbb{Z}, \mathit{return\_host} : X \; \right]$$

The type D has an additional z field.

$$D \mathrel{\widehat{=}} \mu(X) \left[\; y, z : \mathbb{Z}, \mathit{return\_a\_value} : \mathbb{Z}, \mathit{return\_host} : X \; \right]$$

Instance d of type D overrides the first method, *return_a_value*, to return the z field of the object resulting from the invocation of the *return_host* method.

$$d \mathrel{\widehat{=}} \mathit{object} \left\{ \begin{array}{l} y = 0, z = 0, \mathit{return\_a\_value} = \varsigma(s : D)\; s.\mathit{return\_host}.z, \\ \mathit{return\_host} = \varsigma(s : D)\; s \end{array} \right\}$$

Although D merely extends C by an additional field z, it is not a subtype: $\neg\ (D \preceq C)$. Assuming it is a subtype ($D \preceq C$) leads to a contradiction. If it were possible to subsume d to the type C then the method *return_host* could be updated to return the object c:

$$d_{\odot}\mathit{return\_host} \leftleftarrows c \equiv \mathit{object} \left\{ \begin{array}{l} y = 0, z = 0, \\ \mathit{return\_a\_value} = \varsigma(s : D)\; s.\mathit{return\_host}.z, \\ \mathit{return\_host} = \varsigma(s : D)\; c \end{array} \right\}$$

The result of invoking the method *return_a_value* on the updated d would be:

$$\begin{array}{l} (d_{\odot}\mathit{return\_host} \leftleftarrows c).\mathit{return\_a\_value} \\ \equiv \\ (d_{\odot}\mathit{return\_host} \leftleftarrows c).\mathit{return\_host}.z \\ \equiv \\ c.\mathit{return\_host}.z \\ \equiv \\ c.z \end{array}$$

A semantic error would occur during execution as c does not have a z field. This problem arises as the updated d returns an object of type C. One solution to the problem is to

[7] Example adapted from [AC96, p123].

restrict d to only return modified versions of itself—not arbitrary objects. This can be achieved by covariantly annotating the methods, thus blocking the update in the previous example of the *return_host* method. The subtyping rule for recursive, variance annotated objects (ignoring well-formedness constraints) is as follows. Given object types $C \mathrel{\widehat{=}} \mu(X) \left[\; i \in 1..n \bullet l_i : B_i\{X\}^{v_i} \; \right]$ and $D \mathrel{\widehat{=}} \mu(X') \left[\; i \in 1..n+m \bullet l_i : B'_i\{X'\}^{v'_i} \; \right]$ where v_i and v'_i are variance annotations:

$$\frac{\textit{for all } i \in 1..n \bullet \; (E, X' \preceq C) \vdash B'_i\{X'\}^{v'_i} \preceq (B_i^{v_i}[X \backslash C])}{E \vdash D \preceq C}$$

That is, D is a subtype of C if under the assumption $X' \preceq C$, the types of the attributes of C can be shown to be supertypes of the respective attribute types of D.

Covariant occurrences of 'self' are supported by variance annotated, recursive object types, yet contravariant occurrences are not.

Theorem 4.24 (Variance Annotated Recursive Types)
Consequently, given the object type

$$C \mathrel{\widehat{=}} \mu(X) \left[\; m : \mathit{Ptrans}\, \{\mathit{self} : X\}_{\boxed{\text{SD}}}\, \{\mathit{self} : X\}_{\boxed{\text{SD}}}{}^{+} \; \right]$$

the closest type to the desired subtype which extends C with a method n is:

$$D \mathrel{\widehat{=}} \mu(X') \left[\begin{array}{l} m : \mathit{Ptrans}\, \{\mathit{self} : X'\}_{\boxed{\text{SD}}}\, \{\mathit{self} : C\}_{\boxed{\text{SD}}}{}^{+}, \\ n : \mathit{Ptrans}\, \{\mathit{self} : X'\}_{\boxed{\text{SD}}}\, \{\mathit{self} : C\}_{\boxed{\text{SD}}}{}^{+} \end{array} \right]$$

Proof
Proof requires showing that, assuming $X' \preceq C$ then:

$$\mathit{Ptrans}\, \{\mathit{self} : X'\}_{\boxed{\text{SD}}}\, \{\mathit{self} : C\}_{\boxed{\text{SD}}}{}^{+} \preceq \mathit{Ptrans}\, \{\mathit{self} : C\}_{\boxed{\text{SD}}}\, \{\mathit{self} : C\}_{\boxed{\text{SD}}}{}^{+}$$

which holds according to the subtyping rules of (variant) functions (essentially Theorem 4.14).
QED

Using self types instead of variance annotated, recursive types does not result in similar properties as the subtyping rule only allows occurrences of self in covariant positions.

Given that the static typing systems considered are not sufficient to deal with contravariant occurrences of self, one solution is to use dynamic typing—or dynamic type checks. The dynamic type checks that are used here avoid the methodological issues identified earlier. Consider the following object d of type D[8] with methods m and n (with method bodies m_{pt} and n_{pt} respectively).

$$d \mathrel{\widehat{=}} \mathit{object} \left\{ \; m = m_{pt}, n = n_{pt} \; \right\}$$

[8]The requirement to recursively fold the object has been ignored here.

By dissecting m (and n) and inserting an appropriate dynamic type check, the lost attributes of the precondition are reacquired. Given that

$$\begin{array}{l} m_{pt} \equiv \lambda\, post : \mathrm{pred}\, \{self : X'\}_{\boxed{SD}} \bullet \\ \qquad \lambda\, s : \{self : C\}_{\boxed{SD}} \bullet \\ \qquad\quad m_{pt}\ post\ s \end{array}$$

a type check on *self* in the precondition state s can be made, thereby 'lifting' the precondition state type from $\{self : C\}_{\boxed{SD}}$ to $\{self : D\}_{\boxed{SD}}$. The object *new_d* uses such a type check to provide an improved version of d.

$$new\_d \mathrel{\widehat{=}} object \left\{ \begin{array}{l} m = \lambda\, post : \mathrm{pred}\, \{self : X'\}_{\boxed{SD}} \bullet \lambda\, s : \{self : C\}_{\boxed{SD}} \bullet \\ \qquad \textbf{typecase}\ \ s\ \textbf{when}\ \ (t : \{self : D\}_{\boxed{SD}}) \\ \qquad\ \ \textbf{then}\ (new\_m_{pt}\ post)(t)\ \ \textbf{else false}, \\ n = \ldots \end{array} \right\}$$

The state s is known statically to be of type $\{self : C\}_{\boxed{SD}}$. Consequently, it is not possible to access the attribute n of $s(self)$ as n is not an attribute of C. Thus m_{pt} can only be written as a statement of type $Ptrans\ \{self : D\}_{\boxed{SD}}\ \{self : C\}_{\boxed{SD}}$ (X' equates to D in this context). If s is dynamically of type $\{self : D\}_{\boxed{SD}}$ then the type-case binds s to t which is statically of type $\{self : D\}_{\boxed{SD}}$. Since t has regained the 'lost' attributes of s, the predicate transformer new_m_{pt} can be written so that it has type $Ptrans\ \{self : D\}_{\boxed{SD}}\ \{self : D\}_{\boxed{SD}}$. Consequently, dynamically *new_d* acts as though it is of type:

$$\mu(X') \left[\begin{array}{l} m : Ptrans\ \{self : X'\}_{\boxed{SD}}\ \{self : X'\}^{+}_{\boxed{SD}}, \\ n : Ptrans\ \{self : X'\}_{\boxed{SD}}\ \{self : X'\}^{+}_{\boxed{SD}} \end{array} \right]$$

though statically (for subtyping purposes) it is of type D:

$$\mu(X') \left[\begin{array}{l} m : Ptrans\ \{self : X'\}_{\boxed{SD}}\ \{self : C\}^{+}_{\boxed{SD}}, \\ n : Ptrans\ \{self : X'\}_{\boxed{SD}}\ \{self : C\}^{+}_{\boxed{SD}} \end{array} \right]$$

Since the method m of *new_d* is embedded inside an object of type D the method will only be invoked when *self* is of type D. Consequently the **else** type case branch will never actually be executed. The actual value of the **else** type case branch is therefore of no significance. It has been modelled here as the abort statement:

$(\textbf{abort}\ post)(t)$
$\equiv$ Definition Abort (4.15)
$False(t)$
$\equiv$ Definition Predicate Falsity (4.8)
false

The semantics of the object calculus is based on a variant of the self-application denotational semantics. This semantics uses a continuous function space. Predicate transformers, however, have the continuity constraint removed as continuity forces the nondeterminism to be bounded or finite. Although of little practical importance, it is theoretically more elegant for an object-oriented refinement calculus to support unbounded

non-determinism. Unbounded non-determinism allows such specifications as

$$a: \left[a \in \mathbb{Z} \right]$$

whereas a bounded non-deterministic specification can only use a finite set of integers:

$$a: \left[a \in -Bound..Bound \right]$$

where *Bound* is an arbitrary integer.

The combination of recursive types and the loss of continuity causes difficulties within denotational semantics research. Standard denotational techniques exist for recursive domain equations [Sch88] and for equations without continuity [GHP95] but not both. In fact, Utting [Utt92, p73] proves that, informally, without continuity no domain exists for a denotational semantics in which every program can be found as a method of an object. Naumann [CN00, p15] provides a solution by restricting object methods to disallow mutual recursion. Since not every program can be an object's method, the isomorphism between programs and the value space model that Utting's theorem relies on is abandoned. To ensure the existence of a denotational semantics, any object-oriented refinement calculus built using the object model described previously must also ensure that no method is mutually recursive.

4.5 Predicate Transformer Objects

This section discusses the integration of several object-oriented features with (object-oriented refinement calculus) objects, termed *predicate transformer objects*. This includes the definition of an object syntax, designed to abstract the details of (object-oriented refinement calculus) objects. Section 4.5.1 illustrates the use of subsumption for encapsulating an object's attributes. Section 4.5.2 integrates invariants and history properties with predicate transformer objects.

Having developed an appropriate object model in the previous section, an object syntax is provided to abstract from the model's details. This abstraction also allows the substitution of the object representation with other models with similar properties.

A *predicate transformer object type* (referred to hereafter as an object type), with fields $f_1..f_p$ and methods $m_1..m_q$ has the syntax:

Object
  field $f_{i \in 1..p} : F_i$
  method $m_{i \in 1..q}$
end

The method types can be deduced and for conciseness are omitted.

A predicate transformer object (instance) has the syntax:

object
  field $f_{i\in 1..p} : F_i := fv_i$
  method $m_{i\in 1..q} = mv_i$
end

The field types may be omitted when the object's type is provided in the context.

To allow easy object-oriented style variable introductions, the type of the object can be omitted when it is introduced as the type can be deduced from the object.

Definition 4.25 (Scoped Object Definition (Semantics for Values))

$$
\begin{array}{l}
\lbag\, [\, \textbf{var}\ o := \ \textbf{object} \\
\qquad\qquad \textbf{field}\ f_{i\in 1..p} : F_i := fv_i \\
\qquad\qquad \textbf{method}\ m_{i\in 1..q} = mv_i \\
\qquad\quad \textbf{end} \bullet \\
\qquad Prog \\
\,]\, \rbag \\
\widehat{=} \\
\lbag\, [\, \textbf{var}\ o : \ \textbf{Object} \\
\qquad\qquad \textbf{field}\ f_{i\in 1..p} : F_i \\
\qquad\qquad \textbf{method}\ m_{i\in 1..q} \\
\qquad\quad \textbf{end} := \\
\qquad\quad \textbf{object} \\
\qquad\qquad \textbf{field}\ f_{i\in 1..p} : F_i := fv_i \\
\qquad\qquad \textbf{method}\ m_{i\in 1..q} = mv_i \\
\qquad\quad \textbf{end} \bullet \\
\qquad Prog \\
\,]\, \rbag
\end{array}
$$

$\diamond$

4.5.1 Private Attributes

Private attributes are attributes of an object that are inaccessible to clients of an object. They are used for security (holding sensitive information) and for data-abstraction (withholding the data's representation from the client). This allows the alteration of an object without affecting the object's clients.

Private attributes can be modelled using subsumption. Given the object definition

$$
\begin{array}{l}
\lbag\, [\, \textbf{var}\ o := \ \textbf{object} \\
\qquad\qquad \textbf{field}\ f : F := fv \\
\qquad\qquad \textbf{field}\ pf : PF := pfv \\
\qquad\qquad \textbf{method}\ m = mv \\
\qquad\qquad \textbf{method}\ pm = pmv \\
\qquad\quad \textbf{end} \bullet \\
\qquad Prog \\
\,]\, \rbag
\end{array}
$$

a client can create instances of the object:

$$
\begin{array}{lll}
\mathit{Prog} \;\widehat{=}\; & |[\, \textbf{var}\ p : & \textbf{Object} \\
& & \quad \textbf{field}\ f : F \\
& & \quad \textbf{field}\ pf : PF \\
& & \quad \textbf{method}\ m \\
& & \quad \textbf{method}\ pm \\
& & \textbf{end} := o \bullet \\
& \quad \mathit{SubProg} & \\
&]| &
\end{array}
$$

SubProg has access to all attributes of *p*. If the syntax $\tau(o)$ is used to denote the type of object *o* then the following program fragment is equivalent.

$$
\begin{array}{lll}
\mathit{Prog} \;\widehat{=}\; & |[\, \textbf{var}\ p : \tau(o) := o \bullet & \\
& \quad \mathit{SubProg} & \\
&]| &
\end{array}
$$

Alternatively, to model private attributes, the **client** may decide to hide certain attributes of *p* from itself by introducing *p* as a supertype of $\tau(o)$:

$$
\begin{array}{lll}
\mathit{Prog} \;\widehat{=}\; & |[\, \textbf{var}\ p : & \textbf{Object} \\
& & \quad \textbf{field}\ f : F \\
& & \quad \textbf{method}\ m \\
& & \textbf{end} := o \bullet \\
& \quad \mathit{SubProg} & \\
&]| &
\end{array}
$$

Now *SubProg* only has access to attributes f and *m* of object *p*. The advantage of hiding attributes of *p* is that the hidden attributes can, in future, be data-refined without affecting *SubProg*. 'Good practice' involves hiding at least all fields.

To force all clients to introduce objects as supertypes of their actual types, the **private**

syntax is used[9].

$$
\begin{array}{l}
\lbag \textbf{var } o := \ \textbf{object} \\
\qquad\qquad \textbf{field } f : F := fv \\
\qquad\qquad \textbf{private field } pf : PF := pfv \\
\qquad\qquad \textbf{method } m = mv \\
\qquad\qquad \textbf{private method } pm = pmv \\
\qquad\quad \textbf{end} \bullet \\
\qquad Prog \\
\rbag \\
\widehat{=} \\
\lbag \textbf{var } o : \ \textbf{Object} \\
\qquad\qquad \textbf{field } f : F \\
\qquad\qquad \textbf{method } m \\
\qquad\quad \textbf{end} := \\
\qquad\quad \textbf{object} \\
\qquad\qquad \textbf{field } f : F := fv \\
\qquad\qquad \textbf{field } pf : PF := pfv \\
\qquad\qquad \textbf{method } m = mv \\
\qquad\qquad \textbf{method } pm = pmv \\
\qquad\quad \textbf{end} \bullet \\
\qquad Prog \\
\rbag
\end{array}
$$

Clients of o can now only access the public attributes.

4.5.2 Object Invariants and Dynamic Constraints

This section defines and discusses the use of invariants and dynamic constraints within objects. An *invariant* is a single-state constraint on an object. It holds for each externally observable (i.e., client observable) state. That is, it must be upheld when the object is initialised and at the conclusion of the execution of any method calls on the object. For instance, a typing constraint can be considered as an invariant property. A *dynamic constraint* is a dual-state history property that dictates behaviour that externally observable consecutive states (of all methods) must adhere to. For example, the inclusion of the dynamic constraint $a \geq a_0$ in an object with a field a ensures that the methods of the object can never decrease a. Dynamic constraints are also known as *explicit history properties* [LW94]. They are re-termed here to distinguish between those constraints that an object implements (properties) and those that are specification constraints that are yet to be implemented.

Consider the following model of a bank's credit account[10]. One of the system's functions is to record the transactions on credit accounts. To achieve this, the following class

[9] For a different approach the reader is referred to the work of Sekerinski [Sek96] who uses existential types.

[10] This example is derived from one presented by Mikhajlova and Sekerinski [MS97].

Transaction is introduced using the given type *Date*.

class *Transaction* **is**
  field $amount : \mathbb{Z}$
  field $date : Date$
  constructor $Transaction(amnt : \mathbb{Z}, when : Date) =$
    $amount := amnt;\ date := when$
end

The *amount* field is an integer as the *Transaction* class must be able to store deposits and withdrawals. The class defines a special constructor method which is used to instantiate instances of the class.

The system also has a *CreditAccount* class. The class has an integer field *balance* representing the amount of credit that customer has used. The invariant specifies that this field is limited above by a given constant *LIMIT*. The invariant has not yet been implemented within the method bodies though clients can rely on this knowledge. The class also has a field *t* which maintains a sequence of transactions performed on the account. The dynamic constraint ensures that the sequence of transactions can only be appended to. The zero-subscripted *t*, that is t_0, refers to the initial, or precondition, value of *t* in any method. The non-subscripted *t* refers to the final, or postcondition, value of *t*. This ensures that methods cannot delete the transaction history.

The class also contains an **initially** clause [Mor94] (on modules) which constrains the initial state of instances of the class. The clause requires that *balance* be set to zero and the transaction history is empty ($\langle\,\rangle$ represents the empty sequence). There are also two methods, *deposit* and *withdraw*, which update the fields *balance* and *t* appropriately.

class *CreditAccount* **is**
  field $balance : \mathbb{Z}$
  field t : seq *Transaction*
  invariant $\{\ balance < LIMIT\ \}$
  dynamic $\{\ (t_0$ **prefix of** $t)\ \}$
  initially $balance = 0 \wedge t = \langle\,\rangle$
  method $deposit(amnt : \mathbb{N}, when : Date) =$
    $t := t \frown \langle$**new** $Transaction(-amnt, when)\rangle;$
    $balance := balance - amnt$
  method $withdraw(amnt : \mathbb{N}, when : Date) =$
    $\{balance + amnt < LIMIT\};$
    $t := t \frown \langle$**new** $Transaction(amnt, when)\rangle;$
    $balance := balance + amnt$
end

The invariant and dynamic constraint are crucial to the correct behaviour of instances of the class. Clients of the class may rely on the knowledge that the balance does not exceed the limit. For instance, the knowledge may be used to allow the developer to implement the balance using a certain number of bits. Similarly, the knowledge that the transaction

history is only extended may be used to implement it using a tape backup mechanism since earlier data will not require alteration. The declaration of invariants and dynamic constraints must be inherited into sub-classes to ensure correct functionality. If they were not inherited, the sub-class may introduce a new method that removed the transaction history and/or exceeded the limit.

Formalisation of Invariants Liskov and Wing use an intentionally descriptive and informal style to present invariants and, as they term them, history properties. In contrast, this treatment addresses these issues formally in the refinement calculus. This leads to a better understanding of the concepts and related issues. formal definitions extend the concepts of invariants and dynamic constraints by separating those properties that are desired yet the objects have not yet implemented from those that merely document the implemented behaviour. The nomenclatures *coerced invariant* and *coerced dynamic constraint* are used, respectively, for invariants and dynamic constraints that are yet to be implemented. Within objects, the corresponding syntax $\{Inv\}$ and $\{Dynamic\}$ are used. The nomenclatures *extant dynamic constraint* and *extant invariant* are used, respectively, for invariants and dynamic constraints that are already implemented. Within objects, the corresponding syntax $[Inv]$ and $[Dynamic]$ are used. This division provides clearer and more flexible rules.

The declaration of an *invariant* constrains the object (class) and any sub-object (sub-class). *Extant invariants* document the properties that the methods of the current object uphold and that the new methods of sub-objects must also uphold. For an object with an extant invariant (*Inv*) to be well-formed (or valid), each method must satisfy the invariant. It may assume the invariant beforehand and if it terminates then it must reestablish the invariant at the conclusion of the method. Additionally, the initialisation clause must be at least at strong as the invariant—otherwise the invariant is not established to begin with.

Definition 4.26 (Valid Extant Invariant) More formally, for an object with methods $body_{i \in 1..n}$, initialisation clause *Init* and extant invariant *Inv* to be well-formed the following properties must be maintained.

$$(\forall\, i \in 1..n \bullet Inv \Rrightarrow wlp.body_i.Inv) \wedge$$
$$Init \Rrightarrow Inv$$

◇

The definition uses the weakest liberal precondition rather than the weakest precondition as the method does not need to reestablish the invariant if it does not terminate. Using a weakest precondition definition (instead) is too strong as it also requires that the invariant ensure that the method terminates. It is the client code that must ensure that the precondition holds (and hence the method terminates), not the invariant.

Invariants can only involve an object's private attributes as public attributes can be freely modified by clients. Properties involving only the private attributes of an object

are termed here *protected* properties. At the end of a method call the guard $\lceil Inv \rceil$ is established. From this guard, the assertion $\{Inv\}$ can be immediately obtained:

$$\lceil Inv \rceil \sqsubseteq \lceil Inv \rceil ; \{Inv\}$$

Using the knowledge that the invariant cannot be violated outside method calls, the invariant can be propagated forward to the start of the next method call. This is referred to as a *protected system* [Utt92, p86].

Law 4.27 (Introduce Protected System Assertion) In a protected system, a protected property (refers only to private attributes), P which is coerced by the methods and the initialisation of the object can be re-asserted before the methods.

object
  field $f_{i\in 1..p} : F_{i\in 1..p}$
  initially $Init \wedge P$
  method $m_{i\in 1..q} = body_i;\ \lceil P \rceil$
end
$\equiv$
object
  field $f_{i\in 1..p} : F_{i\in 1..p}$
  initially $Init \wedge P$
  method $m_{i\in 1..q} = \{P\};\ body_i;\ \lceil P \rceil$
end

Recursive calls. During the execution of the method the invariant may temporarily be broken. Consequently for recursive (self calls) or mutually recursive calls, the onus is on the developer to ensure that the object's invariant is upheld just prior to the recursive call.

Coerced invariants can be used to further constrain the implementation of methods. They dictate the desired behaviour of the methods, not the actual properties. However, clients of the object may assume coerced invariants as eventually (through the refinement process) the methods will implement them. Declaring a coerced invariant actually denotes the encapsulation of each method body by an assumption and guard involving the invariant. After encapsulating the method bodies, the coerced invariant can then be replaced by an equivalent *extant invariant*.

Definition 4.28 (Coerced Invariants)

object
  field $f_{i\in 1..p} : F_{i\in 1..p}$
  invariant $\{\ Inv\ \}$
  initially *Init*
  method $m_{i\in 1..q} = body_i$
end
$\widehat{=}$
object
  field $f_{i\in 1..p} : F_{i\in 1..p}$
  invariant $[\ Inv\ ]$
  initially $Init \wedge Inv$
  method $m_{i\in 1..q} = \{Inv\}\ body_i\ [Inv]$
end

$\diamond$

Each method can use the invariant for its refinement by assuming that the invariant holds initially. Each method must reestablish the invariant upon conclusion.

Care must be taken when declaring a coerced invariant so as not to constrain the object such that the invariant cannot be met. In the extreme case, declaring the coerced invariant *False* introduces a *miracle* as it can be neither established by the initialisation or reestablished by the methods. As an alternative example, a method may establish the postcondition $a = 1$. Specifying the coerced invariant $a = 2$ would mean that the object is miraculous. Miraculous objects cannot be implemented as code.

In contrast the inclusion of an extant invariant merely documents properties of the object. The object's methods already reestablish the invariant.

It is intended that objects be specified using coerced invariants. Subsequently, during the implementation of the object the coerced invariant should be translated into extant invariants. Further development is used to remove the assertion and coercion (guard) of the invariant as they are not code.

Example 4.29 (Invariants) The following object with field f is a valid object as the extant invariant is upheld.

object
  field $f : \mathbb{N}$
  invariant $[\ f \in \mathbf{Even}\ ]$
  initially $f = 0$
  method $increase = f := f + 1;\ f := f + 1$
end

However, the following object is not valid as the extant invariant is not reestablished by

the method *increase*.

object
  field $f : \mathbb{N}$
  invariant $\lceil\ f \in \textbf{Even}\ \rceil$
  initially $f = 0$
  method *increase* $= f := f + 1$
end

Changing the initialisation to $f = 1$ does not produce a valid object as the initialisation must entail the invariant.

In contrast, the following object that uses a coerced invariant instead is valid as the concluding guard reestablishes the invariant. For this example a *miracle* is required to reestablish the invariant. Consequently, although this object is valid it cannot be implemented.

object
  invariant $\{\ f \in \textbf{Even}\ \}$
  initially $f = 0$
  field $f : \mathbb{N}$
  method *increase* $= f := f + 1$
end
$\widehat{=}$ Coerced Invariants (4.28)
object
  invariant $\lceil\ f \in \textbf{Even}\ \rceil$
  initially $f = 0$
  field $f : \mathbb{N}$
  method *increase* $=$
    $\{f \in \textbf{Even}\};\ \ f := f + 1;\ \ \lfloor f \in \textbf{Even} \rfloor$
end

$\diamond$

Formalisation of Dynamic Constraints The modelling of coerced dynamic constraints introduced here allows the specification of dual-state properties similar to those of Liskov and Wing [LW94]. Analogous to *extant invariants*, the inclusion of an *extant dynamic constraint* actually denotes a well-formedness (validity) constraint on the object. The methods of an object with an extant dynamic constraint must establish the dual-state property.

Definition 4.30 (Valid Extant Dynamic) For an object with methods $body_{i \in 1..n}$, extant dynamic constraint D on (final state) variables $\vec{u}$ and (initial) variables $\vec{u_0}$, and invariant *Inv* to be well-formed the following property must be maintained.

$$\forall\, i \in 1..n \bullet Inv \wedge \vec{U} = \vec{u} \Rrightarrow wlp.body.D[\vec{u_0} \backslash \vec{U}]$$

for fresh $\vec{U}$.

◇

The declaration of a *coerced dynamic constraint* further constrains each method possibly beyond the properties it currently upholds.

Definition 4.31 (Coerced Dynamic Constraint)

$$
\begin{array}{l}
\textbf{object} \\
\quad \textbf{field}\ f_{i\in 1..p} : F_{i\in 1..p} \\
\quad \textbf{dynamic}\ \{\!|\ D\ |\!\} \\
\quad \textbf{initially}\ \mathit{Init} \\
\quad \textbf{method}\ m_{i\in 1..q} = \mathit{body}_i \\
\textbf{end} \\
\widehat{=} \\
\textbf{object} \\
\quad \textbf{field}\ f_{i\in 1..p} : F_{i\in 1..p} \\
\quad \textbf{dynamic}\ [\!|\ D\ |\!] \\
\quad \textbf{initially}\ \mathit{Init} \\
\quad \textbf{method}\ m_{i\in 1..q} = \\
\qquad |[\ \textbf{con}\ \vec{U} \bullet \\
\qquad\quad \{\vec{u} = \vec{U}\};\ \ \mathit{body}_i;\ \ [\!|\ D[\vec{u_0} \backslash \vec{U}]\ |\!] \\
\qquad]| \\
\textbf{end}
\end{array}
$$

The syntax $|[\ \textbf{con}\ \vec{U} \bullet \ldots\ ]|$ denotes the introduction of logical constants in an analogous manner to a variable block. In this context, it *saves* the initial values of the variables $\vec{u_0}$ for use in the guard following the method body. The definition is similar to the model of initial variables provided by Morgan [Mor94].

◇

Included in first class provided at the start of this section was a *constructor* method *Transaction*. The inclusion of invariants and dynamic constraints highlights the need to treat constructor methods in a differing manner to normal methods. That is, constructor methods do not need to initially obey the the invariant and they also do not need to adhere to the dynamic constraint. Consequently, ignoring the preconditions on parameters to constructor methods, they should be refinements of the specification:

$$\vec{w} : [\ \mathit{True}\ ,\ \mathit{Inv} \wedge \mathit{Init}\ ]$$

where *Inv* is the invariant, *Init* is the initialisation clause and $\vec{w}$ are the object's fields. Constructor methods must be called at most once, and they are the first method called.

Related Work Liskov and Wing [LW94] discuss both implicit and explicit approaches to modelling dynamic constraints. They term the explicit approach a 'constraint' approach and the implicit approach an 'explanation' approach. The term 'explanation' arises due to the approach taken to support subclasses. To ensure the implicit dynamic constraints are

not violated by new methods, the new methods must be explained in terms of the existing methods. This ensures that no new reachable states are introduced by the new methods. In their conclusions they recommend using the explicit approach. Only explicit declarations of invariants and dynamic constraints are considered in this paper. Liskov and Wing's explicit invariants and history properties are most closely related to our extant invariants and extant dynamic constraints.

Mikhajlova [Mik99a] has modelled invariants for components. Her interpretation and hence model of invariants is quite different to ours (in some sense, reversed). She also does not distinguish between coerced invariants and extant invariants. Many of the concepts are similar, however. For instance, she identified the need to reestablish invariants prior to self calls. She suggests further that this is not sufficient when super-calls are introduced as an encompassed self-call (occurring within the super-call) will only reestablish the super-class' invariant, not necessarily the sub-class' invariant. This property is possibly not strong enough. Since the self-call (within the super-class call) will invoke a method on the sub-class it is the sub-class' invariant that should hold. Provided that super-calls are constrained to ensure that a strong enough invariant is established prior to encompassed self-calls, this does not appear to be problematic.

4.6 Client Constructs

This section provides predicate transformer semantics for the language constructs used for accessing and manipulating objects within a semantics for values. These language constructs, termed *client constructs*, include field selection, internal field update, object specification and method call. Distinctions are made between internal and external field updates and method calls. *Internal* constructs are those which update their *host* while *external* are those that update objects other than their host.

4.6.1 Field Selection

Field selection is used to extract the value of an object's field. It is defined for both object variables and explicit object constructions. Field selection is performed as part of an expression, typically in an expression that may occur on the right hand side of an assignment, or guard of an iteration or alternation statement.

Definition 4.32 (Object Field Selection) Selecting a field is defined as object calculus field selection.

$$o.\mathbf{select}\, f \;\widehat{=}\; o_{\odot} f$$

◇

Example 4.33 (Object Field Selection Example) For example, given an integer p and an object o with an integer field f then the predicate transformer $p := o.\mathbf{select}\, f$ is the predicate transformer $p := o_{\odot} f$.

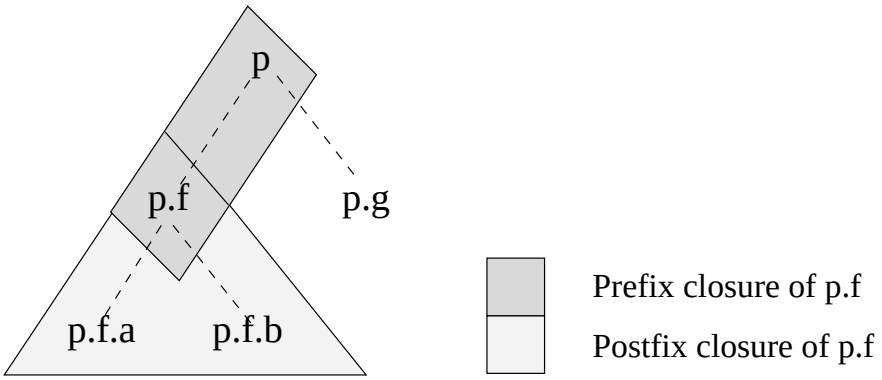

Figure 4.1: Attribute Path Closures

◇

Internal field selection is accomplished using the variable *self* in lieu of *o*, e.g., *self* .**select** f selects the host's f field.

4.6.2 Object Specifications (Semantics for Values)

Specification statements, as presented in Section 2.3 and Definition A.48, deal with the alteration of program variables (the frame). Consequently, given an object variable o with a Boolean field l, the specification o**:** $\left[\, o.\textbf{select}\ l = \mathit{True} \,\right]$ specifies the desire for the l field of o to be equated to *True*. However, it does not restrict any other attributes of o. Typically it is desired that some attributes be constrained to their original values. An object specification is a novel language construct that provides a shorthand for automatically constraining the attributes not listed in the frame. For example, the object specification:

$$p.f \mathbf{::} \left[\, \mathit{post} \,\right]$$

allows the attribute f of object p to be modified. The term *attribute path* is used to denote a path to an attribute (or object), e.g., $p.f$ and o are attribute paths. The modification of p's attribute f is equivalent to the modification of p (needed to allow $p.f$ to alter) and all attributes of $p.f$ etc., but not all attributes of p. All shaded areas in Figure 4.1 are eligible for modification.

Object specifications are defined using a conservative extension of classical specifications by adding constraints about which attributes can be modified—these are determined by the attribute paths in the frame of the object specification. An attribute path is treated as a sequence of identifiers. For example, the attribute path $p.f$ has the sequence $\langle p, f \rangle$ associated with it:

$$p.f = \langle p, f \rangle$$

The only program variable that needs to be modified is p. If a classical specification were used instead of an object specification, it would be of the form:

$$p\mathbf{:} \left[\, \mathit{post} \wedge \mathit{constraint} \,\right]$$

where *constraint* ensures that those attributes not in the attribute paths remain stable. The frame consists of the heads of the attribute path sequences.

From Figure 4.1 it can be seen that the set of attributes that may be modified for a particular attribute path is the union of all attributes leading to the attribute path and all attributes 'under' the attribute path. When attribute paths are treated as sequences, these are the prefix and postfix closures, respectively. Thus the calculation of the modifiable attributes is split into the calculation of the prefix and postfix closures.

The postfix closure of an attribute path s is denoted by $poc(s)$. All attribute paths in the tree beneath s are included. Hence $poc(s)$ returns a (finite) set of sequences of labels. Using the example from Figure 4.1, $poc(p.f) = \{p.f, p.f.a, p.f.b\}$. The postfix closure of s is the set of attributes of s and the postfix closure of those attributes.

$$poc(s) \mathrel{\widehat=} \{s\} \cup \bigcup\{i \in attributes(s) \bullet poc(s \frown \langle i \rangle)\}$$

The postfix closure is formed by including all attributes of s, and then recursively including the attributes of those.

The prefix closure of the attribute path s, denoted by $prec(s)$, are those attribute paths that lead to s. It is a set of sequences, for example, $prec(p.f) = \{p, p.f\}$. The prefix closure of s is the combination of all prefixes of s.

$$prec(s) \mathrel{\widehat=} \{j \in 1..\#\, s \bullet s \upharpoonright j\}$$

Definition 4.34 (Object Specifications (Semantics for Values)) The frame of the equivalent classical specification is the heads of each attribute path in the frame, i.e., $head(\!|\ L\ |\!)$. These are termed the *bases*. The attributes in the prefix and postfix closures of the frame may be altered. *frame* returns the prefix and postfix closures of the frame.

$$frame \mathrel{\widehat=} \bigcup\{s \in L \bullet poc(s) \cup prec(s)\}$$

Those attributes not in the prefix and postfix closures must remain invariant. *nonframe* returns those attributes not in the closure.

$$nonframe \mathrel{\widehat=} \bigcup\{s \in head(\!|\ L\ |\!) \bullet poc(s)\} \backslash frame$$

The constraint added to the classical specification's postcondition is that all attributes, except those in the prefix and postfix closures of the frame, remain invariant.

$$L{::}\left[post\right] \mathrel{\widehat=} head(\!|\ L\ |\!){:}\left[\begin{array}{c} post \wedge \\ \forall\, a \in nonframe \bullet a = a_0 \end{array}\right]$$

◇

Here the annotation of the attribute path a actually denotes the annotation of its base, e.g., $(p.f)_0 = p_0.\mathbf{select}\, f$. For the example from Figure 4.1, the set of bases is $\{p\}$. The postfix closure of these bases is:

$$\{p,\ p.f,\ p.f.a,\ p.f.b,\ p.g\}$$

The postfix closure of the attribute paths is the whole tree, excluding p and $p.g$.

$$\{p.f,\ p.f.a,\ p.f.b\}$$

The prefix closure of the attribute paths is

$$\{p,\ p.f\}$$

Using Object Specifications (Semantics for Values) (4.34) definition, the constraint to be added reduces to (informally):

$$\forall\, a \in \mathit{all\_attributes} \setminus (\mathit{prefix\_closure} \cup \mathit{postfix\_closure}) \bullet a = a_0$$

In this example, this is equivalent to:

$$\begin{array}{l} \forall\, a \in \{p,\ p.f,\ p.f.a,\ p.f.b,\ p.g\} \setminus \\ \qquad (\{p.f,\ p.f.a,\ p.f.b\} \cup \{p,\ p.f\}) \bullet \\ \qquad\qquad a = a_0 \\ \equiv \\ \forall\, a \in \{p.g\} \bullet a = a_0 \end{array}$$

This means $p.g$ must remain stable. The equivalent classical specification is therefore:

$$p.f{::}\left[\mathit{post}\right] \equiv p{:}\left[\mathit{post} \wedge p.g = p_0.g\right]$$

The overlap of closures in the frames of object specifications provides opportunities for modifying the object specification's frame. An example of such frame modification is provided by Theorem 4.35.

Theorem 4.35 (Expanding the Frame with a Postfix Closure) **Proof on page 174**

The object specification frame may be extended with an attribute path (p) that is in the postfix closure of an attribute path (o) that is already in the frame.

$$o{::}\left[\mathit{post}\right] \equiv o, p{::}\left[\mathit{post}\right]$$

provided $p \in \mathit{poc}(o)$.

For the previous example,

$$p.f{::}\left[\mathit{post}\right] \equiv p.f, p.f.a{::}\left[\mathit{post}\right]$$

as $p.f.a$ is in the postfix closure of $p.f$.

4.6.3 Field Update

Object field updates are used to change the value of object fields. An update of an object o is defined as an assignment to o of an (object calculus) updated object.

Definition 4.36 (Object Field Update (Semantics for Values)) Updating field f of object o with value e is defined as:

$$o.\textbf{update}\ f\ \textbf{with}\ e\ \mathrel{\widehat{=}}\ o := (o_{\odot} f \leftleftarrows e)$$

The updated object is formed using object calculus method update to replace the body of field f with e.
◇

Object field updates can be introduced from object specifications as shown in Theorem 4.37.

Theorem 4.37 (Introduce Field Update (Semantics for Values)) **Proof on page 173**
Given object value variable o

$$o.f :: \left[\, o.\textbf{select}\ f = e \,\right] \equiv o.\textbf{update}\ f\ \ \textbf{with}\ e$$

4.6.4 Method Invocation

The methods of objects in our semantics are actually fields (with a predicate transformer type) and consequently the self quantifier of the object calculus is not used in determining the effects of method calls (see Section 2.2 on page 12). When a method is called on an object, the appropriate field corresponding to the method name is selected from the object. The result of the field selection is then 'executed.' This is similar in nature to the copy rule procedure semantics of the refinement calculus.

Definition 4.38 (Internal Object Method Invocation (Semantics for Values)) Invoking another method of the host reduces to object calculus field selection. Informally:

$$\textbf{call}\ m\ \mathrel{\widehat{=}}\ self_{\odot} m$$

The value of the variable *self* is evaluated in the precondition state s. For postcondition p, internal method call is defined as:

$$(\textbf{call}\ m)\ p\ s\ \mathrel{\widehat{=}}\ s(self)_{\odot} m\ p\ s$$

◇

External Object Method Invocation

External object method invocation involves calling a non-hosted method. This requires modifying the *self* variable to refer to the new object, an internal invocation, and finally, resetting *self* back to its original value. While *self* is set to the new 'called' object, the old value of *self* must be stored. The overall effect is to provide a stack of *self*s. Modelling states as records (Definition 4.5) means that the state type is not sophisticated enough to handle stacking. A similar effect can be achieved by introducing a fresh variable (v) to temporarily hold *self*. The variable *self* is updated with the new host object (o) on which the method is called and an internal method call is made. After the method call, the current *self* is copied back to o. The value of *self* is then reset to that temporarily held in v. The type of *self* is switched between the original and new types by exiting one state space and entering the other. The statement **exit** *self*, as given in Definition A.39, is used to remove *self* from the state space. It is analogous to the closing brackets of a local variable block. The statement **enter** $self : Self := v$, as given in Definition A.37, is used to add *self* to the state space. It is analogous to the opening brackets of a local variable block.

Definition 4.39 (External Object Method Invocation (Semantics for Values))

$$
\begin{array}{l}
o.\mathbf{call}\ m\ \widehat{=} \\
\quad |[\ \mathbf{var}\ v : Self := self; \\
\qquad \mathbf{exit}\ self; \\
\qquad\ |[\ \mathbf{var}\ self : O \bullet \\
\qquad\qquad self := o; \\
\qquad\qquad \mathbf{call}\ m \\
\qquad\qquad o := self; \\
\qquad\ ]| \\
\qquad \mathbf{enter}\ self : Self := v; \\
\quad]|
\end{array}
$$

◇

This definition does not handle recursion. For instance, if the call to m updates the self object through o, rather than *self*, then the update to o after the call ($o := self$) would overwrite the alterations, reinstating o's original value. One solution to this dilemma involves the use of reference variables. By stacking references to *self*, rather than stacking the *self* values, all recursive calls on the stack dereference the same value. This is the approach taken for the reference semantics introduced later. Since this problem is due to the handling of the variable *self*, another solution would be to stop associating the variable *self* with the host. Each method call would then require explicit identification of the host.

4.7 Semantics for References

Object identity is a defining aspect of object-oriented languages. Therefore object-oriented languages usually incorporate a semantics based on object references. To provide such a capability the current semantics for values is extended with a store. New constructs are introduced that augment yet do not replace the existing constructs. A set of object references *Ref* is introduced. An object reference is a location in the store: given an object reference, an object's value is determined by indexing the store at the given location. The store is provided as a function from object references to objects:

$$Ref \pfun \gamma$$

where γ is the type of the objects in the store. To allow the use of aliasing control techniques, such as those presented by Utting [Utt95] and Bancroft [Ban97], multiple stores are supported. Stores are held within the state. A store function called *store*, containing objects of type α, would use the following state type:

$$\{store : (Ref \pfun \alpha)\}_{\boxed{SD}}$$

4.7.1 Objects (Semantics for References)

This section introduces new constructs for use with the extended semantics that incorporates references. Specifically, object construction and assignment are provided with additional, reference semantics definitions.

Object Construction Object constructions in a semantics for references introduce a variable of type *Ref*, allocate space in the store $store_j$ and copy the object into the store at the allocated location. Thus the program

$$\begin{array}{l} |[\ \textbf{var}\ o : Ref\ (store_j) := \ \textbf{object} \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{field}\ f : F := fv \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{method}\ m = mv \\ \qquad\qquad\qquad\qquad\quad \textbf{end} \bullet \\ \quad Prog \\ ]| \end{array}$$

where F is not a reference has the following semantics:

$$\begin{array}{l} |[\ \textbf{var}\ o : Ref\ \bullet \\ \quad \sqcap x \mid x \in Ref \setminus (\bigcup i \bullet \mathrm{dom}\, store_i) \bullet o := x; \\ \quad store_j(o) := \ \textbf{object} \\ \qquad\qquad\qquad\qquad \textbf{field}\ f : F := fv \\ \qquad\qquad\qquad\qquad \textbf{method}\ m = mv \\ \qquad\qquad\qquad \textbf{end}; \\ \quad Prog \\ ]| \end{array}$$

where the demonic choice non-deterministically chooses a reference that has not already been allocated. The quantifier i is used to range over all existing stores.

Assignment to a particular domain element in a function is defined as replacing the entire function with one updated at that point with the new value.

Definition 4.40 (Accessed Function Assignment)

$$x(n) := e \;\widehat{=}\; x := x \oplus \{n \mapsto e\}$$

◇

Consequently:

$$\begin{array}{l} store(o) := \ \textbf{object} \\ \qquad\qquad \textbf{field}\ f : F := fv \\ \qquad\qquad \textbf{method}\ m = mv \\ \qquad \textbf{end} \\ \equiv \\ store := store \oplus \{o \mapsto \ \textbf{object} \\ \qquad\qquad\qquad\qquad \textbf{field}\ f : F := fv \\ \qquad\qquad\qquad\qquad \textbf{method}\ m = mv \\ \qquad\qquad\qquad \textbf{end}\} \end{array}$$

Within the value semantics, an object can have a field that is itself an object. For a reference semantics, to represent an object with a field that is itself an object, the convention is taken that the field type is a reference. Consequently the construction of nested objects may take multiple steps and the 'value' of an object can only be determined in the context of the store (wherein all references can be dereferenced). A nested object is constructed by defining the nested objects, entering them into the store and using these references to form the overall object. For example, the code:

$$\begin{array}{l} |[\ \textbf{var}\ o : Ref\ (store_j) := \ \textbf{object} \\ \qquad\qquad\qquad\qquad \textbf{field}\ f : F := fv \\ \qquad\qquad\qquad\qquad \textbf{field}\ ref : Ref\ (store_k) := \ \textbf{object} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{field}\ y : Y := yv \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{end}; \\ \qquad\qquad\qquad\qquad \textbf{method}\ m = mv \\ \qquad\qquad\qquad \textbf{end} \bullet \\ \qquad Prog \\ \ ]| \end{array}$$

can be given a semantics such as:

$$
\begin{array}{l}
|[\ \textbf{var}\ \mathit{freevar} : \mathit{Ref}\ \bullet \\
\quad \sqcap x \mid x \in \mathit{Ref} \setminus (\bigcup i \bullet \operatorname{dom} \mathit{store}_i) \bullet \mathit{freevar} := x; \\
\quad \mathit{store}_j(\mathit{freevar}) := \ \textbf{object} \\
\qquad\qquad\qquad\qquad \textbf{field}\ y : Y := yv \\
\qquad\qquad\qquad \textbf{end}; \\
\quad |[\ \textbf{var}\ o : \mathit{Ref}\ \bullet \\
\qquad \sqcap x \mid x \in \mathit{Ref} \setminus (\bigcup i \bullet \operatorname{dom} \mathit{store}_i) \bullet o := x; \\
\qquad \mathit{store}_k(o) := \ \textbf{object} \\
\qquad\qquad\qquad\qquad\quad \textbf{field}\ f : F := fv \\
\qquad\qquad\qquad\qquad\quad \textbf{field}\ \mathit{ref} : \mathit{Ref}\ := \mathit{freevar} \\
\qquad\qquad\qquad\qquad\quad \textbf{field}\ m : M := mv \\
\qquad\qquad\qquad\quad \textbf{end}; \\
\qquad \mathit{Prog} \\
\quad]| \\
]|
\end{array}
$$

4.7.2 Client Constructs (Semantics for References)

The client constructs introduced in Section 4.6 were defined for a value semantics. These constructs are revisited in this section in the context of a semantics for references. The main difference for each construct of the semantics for values and references is, of course, that the semantics for references construct must use the store in the evaluation of an object's value.

Object Dereference

To encourage good practice for programming with references, the direct access or manipulation of stores is prevented. To obtain the value of an object stored in a reference variable, object dereference is used. Object dereference returns the value of the object in the store.

Definition 4.41 (Object Dereference) Given a store A the dereference of object o is defined as follows.

$$o{\uparrow_A} \mathrel{\widehat{=}} A(o)$$

The store subscript (A) can be omitted when obvious from the context. The dereference syntax is motivated by that used within Pascal [DW91].
◇

Field Selection

Field selection requires no new definition although the reference must be dereferenced before selection can occur.

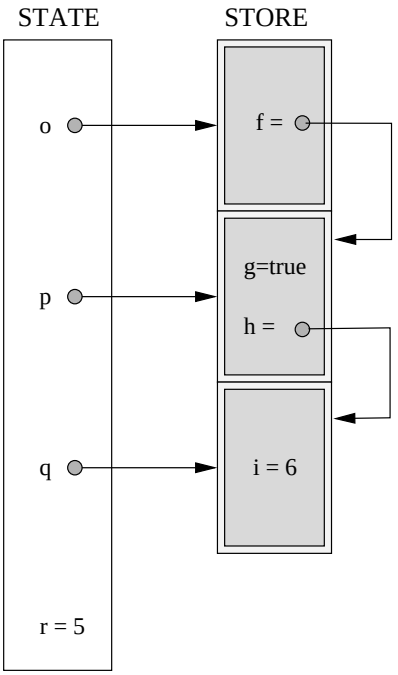

Figure 4.2: Object Specification Example

Example 4.42 (Field Selection using References) Given a reference object o with a field f, to access f the following is used.

$$o\uparrow.\textbf{select}\ f$$

That is, the value of o is obtained by dereference, whereupon Definition 4.32 is used to obtain the value of f.

$$o\uparrow.\textbf{select}\ f \equiv store(o)_{\odot} f$$

$\diamond$

Object Specifications (Semantics for References)

Similar constraints to those required for object specifications in a semantics for values are also needed for object specifications in a semantics for references.

Example 4.43 (Object Specifications Constraints) The program state in Figure 4.2 has three reference variables, o, p, and q and one integer variable r. Object $o\uparrow$ has a field f which is aliased to reference p, that is $o\uparrow.\textbf{select}\ f = p$. Object $p\uparrow$ has a field h which is aliased to q. Object $p\uparrow$ also has a boolean field g. Object $q\uparrow$ has an integer field i.

Reference Attribute Paths: To alter the reference f in the object $o\uparrow$, the following *reference attribute path* syntax should be used:

$$o\uparrow.f$$

Dereference Attribute Paths: The *dereference attribute path*

$$o\uparrow.f\uparrow$$

is used to denote that any attributes of the object $o\uparrow$.**select** $f\uparrow$ may be modified. That is, the attributes g and h may be modified.

Reference Closure: The following attribute path form is *reference closure*:

$$o\uparrow.f!$$

Reference closure is used when all attributes of all objects reachable by references through $o\uparrow$.**select** f may be altered. That is, the attributes of $p\uparrow$ and $q\uparrow$ may be modified.

◇

To simplify the definition of object specifications, the frame is split into four differing types, B for base attribute paths (e.g., o or p), D for dereference attribute paths (e.g., $o\uparrow.f\uparrow$), R for reference attribute paths (e.g., $o\uparrow.f$) and C for reference closure attribute paths (e.g., $o\uparrow.f!$). Also, a function $\zeta(D,R,C) : Ref \nrightarrow \mathbb{P}\,Attributes$ which maps references to the attributes that may be modified at that reference is used. The formation of ζ is decomposed into three parts—one for each of D, R and C.

D: For the dereference attribute path $o\uparrow.f\uparrow$:

$$\zeta = \{o\uparrow.f \mapsto \{g,h\}\}$$

indicating that any attribute of the object in store location $o\uparrow.f$ (or equivalently, via aliasing, p) may be modified. In general, for dereference attribute paths, ζ is defined as:

$$\{i \in D \bullet i \mapsto attributes(i)\}$$

R: For the reference attribute path $o\uparrow.f$;

$$\zeta = \{o \mapsto \{f\}\}$$

indicating that the store can only be altered at the index o, and even then, only attribute f of o may be altered. Since attribute paths are treated as sequences the general definition of ζ for reference attribute paths is:

$$\{i \in R \bullet front(i) \mapsto last(i)\}$$

where $front(i)$ returns all but the last element of the sequence i and $last(i)$ returns the element at the end of the sequence. Consequently, for the reference attribute path $o\uparrow.f\uparrow.h$ the set ζ is:

$$\zeta = \{o\uparrow.f \mapsto \{h\}\}$$

C: For the reference closure attribute path $o\uparrow.f!$:

$$\zeta = \{o\uparrow.f \mapsto \{g,h\}, o\uparrow.f\uparrow.h \mapsto \{i\}\}$$

Aliasing can be used to produce the following, more readable, equivalent set.

$$\zeta = \{p \mapsto \{g,h\}, q \mapsto \{i\}\}$$

This value is calculated recursively. For a particular reference closure attribute path $c \in C$, ζ includes the mapping from c to its attributes and then recursively includes the reference closure of each attribute that is a reference. The function $refcl(c) : Ref \rightarrow \mathbb{P}\, Ref$ is used to calculate reference closures:

$$refcl(c) \mathrel{\widehat{=}} \left(\begin{array}{l} \mu X \bullet \{c \mapsto attributes(c)\} \cup \\ \quad \bigcup \left\{ \begin{array}{r} \{ref \mapsto attrs\} : X \mid \tau(attrs) = Ref \bullet \\ ref\uparrow.attrs \mapsto attributes(ref\uparrow.attrs) \end{array} \right\} \end{array} \right)$$

The condition $\tau(c) = Ref$ is used to select only reference attributes. A least fixed-point definition is provided to account for reference loops, e.g., an object with a reference to itself. Using the Knaster-Taski theorem [BvW98, Theorem 19.1], instantiated with the subset-equality ordering and knowledge of the monotonicity of set union, the least fixed-point can be shown to exist.

To calculate ζ over C the image of *refcl* is taken.

$$refcl(\!|\ C\ |\!)$$

The definition of ζ is achieved by unioning the individual attribute path cases:

$$\zeta(D,R,C) \mathrel{\widehat{=}} \{i \in D \bullet i \mapsto attributes(i)\} \Cup \{i \in R \bullet front(i) \mapsto last(i)\} \Cup refcl(\!|\ C\ |\!)$$

where for each domain element $\Cup$ unions their ranges. For example,

$$\{m \mapsto \{p\}, n \mapsto \{q\}\} \Cup \{m \mapsto \{r\}\} \equiv \{m \mapsto \{p,r\}, n \mapsto \{q\}\}$$

Thus $\Cup$ is defined as:

$$\begin{array}{rcl} j \Cup k & \mathrel{\widehat{=}} & \{i \in \operatorname{dom} j \setminus \operatorname{dom} k \bullet i \mapsto j(i)\} \cup \\ & & \{i \in \operatorname{dom} k \setminus \operatorname{dom} j \bullet i \mapsto k(i)\} \cup \\ & & \{i \in \operatorname{dom} j \cap \operatorname{dom} k \bullet i \mapsto j(i) \cup k(i)\} \end{array}$$

The first set disjunct is for elements only in the domain of j, the second for elements only in the domain of k, the third for elements in both.

Now the constraints to be added to the specification can be determined. Given ζ, the domain of ζ is the set of references that may be altered. One of the constraints to be added

is that store should remain stable at all indices except those that may be altered. This set of indices is calculated by taking the domain of ζ from the domain of *store*:

$$\forall n \in \mathrm{dom}(store) \setminus \mathrm{dom}\,\zeta(D,R,C) \bullet store(n) = store_0(n)$$

For $\zeta(D,R,C) = \{p \mapsto \{g\}\}$ the attribute h of object $p\uparrow$ should remain invariant. The second constraint ensures this. For each modifiable store position, the attributes not in the associated set of modifiable attributes (determined using ζ), should remain constant.

$$\forall refs \in \mathrm{dom}(\zeta(D,R,C)) \bullet \forall a \in attributes(refs\uparrow) \setminus (\zeta(D,R,C))(refs) \bullet \\ refs\uparrow.\mathbf{select}\ a = refs\uparrow_0.\mathbf{select}\ a$$

Collecting these constraints provides the following definition.

Definition 4.44 (Object Specifications (Semantics for References)) Object specifications for a semantics of references, using the store function *store*, are defined as:

$$B, D, R, C :: [post] \mathrel{\widehat=} \\ B, store: \left[\begin{array}{c} post \wedge \\ \forall n \in \mathrm{dom}(store) \setminus \mathrm{dom}\,\zeta(D,R,C) \bullet store(n) = store_0(n) \\ \forall refs \in \mathrm{dom}(\zeta(D,R,C)) \bullet \\ \quad \forall a \in attributes(refs\uparrow) \setminus (\zeta(D,R,C))(refs) \bullet \\ \quad\quad refs\uparrow.\mathbf{select}\ a = refs\uparrow_0.\mathbf{select}\ a \end{array}\right]$$

where B is the base attribute paths, (e.g., o or p), D is the dereference attribute paths (e.g., $o\uparrow.f\uparrow$), R is the reference attribute paths (e.g., $o\uparrow.f$) and C is the reference closure attribute paths (e.g., $o\uparrow.f!$), and $\zeta(D,R,C)$ is a function mapping store indices to the set of attributes which may be altered at that location.
◇

Postulate 4.45 (Extending the Frame with a Reference Closure)
For object specifications using reference attribute paths, the frame can be extended with a reference in the reference closure (p) of a reference closure attribute path ($o!$).

$$o! :: [post] \equiv p, o! :: [post]$$

provided $p \in \mathrm{dom}\, refcl(o)$.

Field Update

Definition 4.46 (Field Update (Semantics for References)) Like value semantics field update, the reference semantics version reduces field update to object calculus method update.

$$o\uparrow.\mathbf{update}\ f\ \ \mathbf{with}\ e \mathrel{\widehat=} store(o) := store(o)_{\odot} f \leftleftarrows e$$

◇

The following theorem can be used to replace an appropriate object specification with a field update.

Theorem 4.47 (Introduce Field Update (Semantics For References)) **Proof on page 173**

An object specification where the attribute f of object reference o may be altered such that its final value is p is equivalent to updating field f of o with p.

$$o\uparrow.f :: \lceil o\uparrow.\textbf{select}\ f = p \rceil \equiv o\uparrow.\textbf{update}\ f\ \ \textbf{with}\ p$$

Internal Method Invocation (Semantics for References)

The definition of internal method invocation is similar to the approach taken within the value semantics version—with the exception that the store is used to calculate *self*. Informally:

$$self\uparrow_{store}.\textbf{call}\ m\ \widehat{=}\ store(self)_{\odot}m$$

Definition 4.48 (Internal Method Invocation (Semantics for References)) Formally, for postcondition p, the method m is invoked on the object $store(self)$ as determined by the precondition state s.

$$self\uparrow_{store}.\textbf{call}\ m\ p\ s\ \widehat{=}\ s(store)(s(self))_{\odot}m\ p\ s$$

◇

External Method Invocation (Semantics for References)

The value semantics version of external method invocation used a technique of stacking object values. That technique did not support recursion. This problem is avoided by stacking object references instead of object values as any 'aliased' references created by recursive calls point to the same object value.

Definition 4.49 (External Method Invocation (Semantics for References))

$$\begin{array}{lll} o\uparrow.\textbf{call}\ m\ \widehat{=} & |[\ \textbf{var}\ v : Ref\ \bullet & \\ & \quad v, self := self, o; & \\ & \quad self\uparrow.\textbf{call}\ m; & \\ & \quad self := v & \\ &]| & \end{array}$$

◇

Chapter 5

Object- and Class-Refinement

Building on the *object-based* specification language of the previous chapter, an object-based refinement calculus is developed by exploring two novel object-refinement relations and providing related object-refinement rules. The calculus is then extended to a *class-based* refinement calculus.

Both object-refinement relations introduced in this chapter are monotonic with respect to object-refinement. Consequently it is a refinement to replace an object with an object-refinement. For example, given classes C and C' such that C' is an object-refinement of C, that is, $C \sqsubseteq_{\varsigma} C'$, and client code *Prog* then

$$\begin{array}{l} \quad \left|\left[\textbf{var}\ \ x : \mathit{Ref}\ \tau(C) \bullet x := \textbf{new}\ \ C;\ \ \mathit{Prog}\ \right]\right| \\ \sqsubseteq \\ \quad \left|\left[\textbf{var}\ \ x : \mathit{Ref}\ \tau(C) \bullet x := \textbf{new}\ \ C';\ \ \mathit{Prog}\ \right]\right| \end{array}$$

where $\tau(C)$ is used to refer to the type of class C. This property, termed *object-refinement simulation*, is similar to Theorem 1 from [BMvW97] and is generalised by Theorem 5.23 on page 96. The generalisation is termed *construction monotonicity* and considers the specific refinement of the **new** construct.

The object-refinement relations of other object-oriented refinement calculi are typically based on data-refinement. That approach is motivated by the inclusion of additional state in an object's subtype. This is aesthetically displeasing as data-refinement need only be used when the state space is altered. Since, by subsumption, the objects with additional attributes are also objects of the original type, the state space does not need altering. Hence data-refinement is not required. For this reason, the classical data-refinement approach is supplemented here with a novel object-refinement relation based on an algorithmic-refinement approach. This relation, defined in Section 5.1, handles the object-refinement of an object to another of either the same type or a subtype. Section 5.2 illustrates the use of the algorithmic-refinement-based relation for the incremental refinement of objects and their clients. Section 5.3 formalises *object-refinement simulation* as a corollary of *construction monotonicity*. The more flexible data-refinement-based relation, introduced in Section 5.4, allows the refinement of an object to one that is not a subtype.

The tradeoff, however, is that the entire object must be data-refined as the data-refinement of a statement is not in general monotonic. These techniques, relations and properties are extended to a class-based approach in Section 5.5.

5.1 Algorithmic Object-Refinement

This section introduces an object-refinement relation based on an algorithmic-refinement approach and presents object-based refinement rules. For instance, rules are presented for object-refining attributes, for adding new attributes and for strengthening invariants and history properties.

Algorithmic object-refinement is defined so that it is possible to substitute (specification) objects with object-refinements—implementation objects. Since object-refinements must guarantee behavioural consistency, the methods of the implementation object must be refinements of those of the specification object. For basic field types, e.g. integers, the fields must be equal, and for object field types, the implementation object's fields must be object-refinements.

The definition of *object-refinement* is broken into three separate rules. These rules show the sufficiency requirements to determine the existence of an object-refinement relation between

- Two objects (Object-Refinement (5.1)),
- Two basic types (Object-Refinement Basic Types (5.2)), and
- Two methods (Object-Refinement Predicate Transformers (5.3)).

Definition 5.1 (Object-Refinement) For objects *impl* and *spec*, *impl* is an object-refinement of *spec* if all attributes of *spec* are correspondingly object-refined in *impl* and if the type of *impl* is a subtype of the type of *spec*.

$$\frac{impl : Impl \qquad spec : Spec \qquad Impl \preceq Spec \qquad \forall j \in \operatorname{dom} attributes(spec) \bullet spec_{\odot} j \sqsubseteq_{\varsigma} impl_{\odot} j}{spec \sqsubseteq_{\varsigma} impl}$$

◇

Definition 5.2 (Object-Refinement Basic Types) Basic types have a discrete subtype ordering. This means that for basic type Σ, if $\Sigma \preceq \Theta$ or $\Theta \preceq \Sigma$ then $\Sigma \equiv \Theta$. For example the Booleans are a basic type. For basic types, two entities are object-refinements if they are equal. For basic type Σ

$$\frac{\sigma : \Sigma \qquad \tau : \Sigma \qquad \sigma = \tau}{\sigma \sqsubseteq_{\varsigma} \tau}$$

◇

Definition 5.3 (Object-Refinement Predicate Transformers) Refinement of predicate transformers implies object-refinement. For predicate transformers pt_{spec} and pt_{impl}

$$\frac{pt_{spec} \sqsubseteq pt_{impl}}{pt_{spec} \sqsubseteq_{\varsigma} pt_{impl}}$$

◇

Like classical refinement, for object-refinement to be useful in practice it must be a preorder (reflexive: Theorem A.58, and transitive: Theorem A.57). This allows, for instance, separate object-refinements to be combined into a single object-refinement.

Object-Refinement (Semantics for References) In a semantics for values, the notion of object-refinement is applicable to *nested* objects: objects that have fields containing other objects. The object-refinement of a field results in the object-refinement of its host. Within a semantics for references, objects are *single-level* entities as their fields are of basic types (including references). Consequently, the nested notion of object-refinement has little practical significance as object-refinement on basic types reduces to equality. For an object-refinement to hold between objects with reference fields the references must be equal. To effect the refinement of an object's field, the object at the referenced store location must be separately object-refined. In summary, whereas for a semantics of values, object-refinement has a recursive nature, for a semantics for references, object-refinement has a single, top level nature—and refinement of an object's fields must be performed as a separate refinement of the store.

The following refinement rules can be used to refine an object's attributes, add new attributes to an object and for strengthening an object's invariant and history property.

Theorem 5.4 (Update Object Field) **Proof on page 175**

Replacing an object's field with an object-refinement is an object-refinement. If object o has field l and $o_{\odot}l \sqsubseteq_{\varsigma} f$, then o's l field can be replaced by f.

$$\frac{o_{\odot}l \sqsubseteq_{\varsigma} f}{o \sqsubseteq_{\varsigma} (o_{\odot}l \leftleftarrows f)}$$

The construct $o_{\odot}l \leftleftarrows f$[1] is object calculus syntax representing the replacement of o's field l with f.

Example 5.5 (Update Object Field) For example, for a semantics for values, this rule

[1] Introduced in Section 2.2.

can be used to show that

object
  field *incrementor* ::= **object**
    field $val : \mathbb{Z} := 8$
    method $inc = (\sqcap y \in \mathbb{N} \bullet val := val + y)$
  end
end
$\sqsubseteq_\varsigma$
object
  field *incrementor* ::= **object**
    field $val : \mathbb{Z} := 8$
    method $inc = val := val + 1$
  end
end

provided, as shown in the next example, that

object
  field $val : \mathbb{Z} := 8$
  method $inc = (\sqcap y \in \mathbb{N} \bullet val := val + y)$
end
$\sqsubseteq_\varsigma$
object
  field $val : \mathbb{Z} := 8$
  method $inc = val := val + 1$
end

$\diamond$

Theorem 5.6 (Update Object Method) **Proof on page 175**

This refinement rule permits the refinement of an object's method, resulting in an object-refinement. Given an object o with a method l, and a method m that is a refinement of $o_\odot l$, then the replacement of $o_\odot l$ with m is a valid object-refinement.

$$\frac{o_\odot l \sqsubseteq m}{o \sqsubseteq_\varsigma (o_\odot l \leftleftarrows m)}$$

Example 5.7 (Update Object Method) This example complements Example 5.5 by showing its proviso. Given that $(\sqcap y \in \mathbb{N} \bullet val := val + y)$ refines to $val := val + 1$, the following object-refinement is valid.

object
  field $val : \mathbb{Z} := 8$
  method $inc = (\sqcap y \in \mathbb{N} \bullet val := val + y)$
end
$\sqsubseteq_\varsigma$ Update Object Method (5.6)
object
  field $val : \mathbb{Z} := 8$
  method $inc = val := val + 1$
end

$\diamond$

The following theorem permits the addition of new attributes to an object.

Theorem 5.8 (Introduce Object Attributes) **Proof on page 176**
Given an object with fields $f_{1..i}$ and methods $m_{1..k}$, adding new fields $f_{i+1..i+j}$ (for $j \geq 0$) or methods $m_{k+1..k+l}$ (for $l \geq 0$) to an object produces an object-refinement. For field values $fv_{1..i+j}$ of types $F_{1..i+j}$, and methods $mv_{1..k+l}$:

object
  field $f_{h\in 1..i} : F_h := fv_h$
  method $m_{h\in 1..k} = mv_h$
end
$\sqsubseteq_\varsigma$
object
  field $f_{h\in 1..i+j} : F_h := fv_h$
  method $m_{h\in 1..k+l} = mv_h$
end

The remainder of this section provides refinement rules for adding new methods to objects, and for strengthening coerced invariants, extant invariants, coerced dynamic constraints and extant dynamic constraints.

When a new method is added to an object with an *extant invariant* or an *extant dynamic constraint* the method must be constrained to avoid violation of these properties. This constraint is achieved by encapsulating the methods with an assertion and guard in a similar manner to the coerced invariant and coerced dynamic constraint definitions (Coerced Invariants (4.28) and Coerced Dynamic Constraint (4.31)). The new method *body* should be encapsulated as follows using the extant invariant *Inv* and extant dynamic constraint *D*:

$|[$ **con** $\vec{U}$ $\bullet$
  $\{Inv \wedge \vec{u} = \vec{U}\}$;
  body;
  $[\, Inv \wedge D[\vec{u_0}\backslash\vec{U}] \,]$
$]|$

Lemma 5.9 (Specification Body) When the method bodies are specifications, the following refinement can be used to simplify the resulting bodies of objects.

$|[$ **con** $\vec{U}$ $\bullet$
  $\{Inv \wedge \vec{u} = \vec{U}\}$;
  $\vec{w}$**:** $[\, p \;,\; q \,]$;
  $[\, Inv \wedge D[\vec{u_0}\backslash\vec{U}] \,]$
$]|$
$\sqsubseteq$
$\vec{w}$**:** $[\, p \wedge Inv \;,\; q \wedge Inv \wedge D \,]$

Postulate 5.10 (Conjoin Extant Invariant)
To strengthen the extant invariant with a conjunct (*EInv*), each method must be further strengthened to ensure it reestablishes the invariant. The initialisation clause must also be strengthened to ensure the invariant is true in the initial state.

object
field $f_{i\in 1..p} : F_{i\in 1..p}$
invariant $\left[\ Inv\ \right]$
initially *Init*
method $m_{i\in 1..q} = body_i$
end
$\sqsubseteq_\varsigma$
object
field $f_{i\in 1..p} : F_{i\in 1..p}$
invariant $\left[\ Inv \wedge EInv\ \right]$
initially $Init \wedge EInv$
method $m_{i\in 1..q} = \{EInv\}\ body_i\ \left[EInv\right]$
end

Proof
The proof of this rule relies on the fact that a *protected system* is used. The rule Introduce Assertion (A.28) and those for propagating assertions [BvW98, Theorem 28.3, p468] [Gro98, Gro00] are also used.
QED

Postulate 5.11 (Conjoin Coerced Invariant)
In contrast, a coerced invariant can be strengthened without altering the method bodies.

object
field $f_{i\in 1..p} : F_{i\in 1..p}$
invariant $\left\{\ Inv\ \right\}$
initially *Init*
method $m_{i\in 1..q} = body_i$
end
$\sqsubseteq_\varsigma$
object
field $f_{i\in 1..p} : F_{i\in 1..p}$
invariant $\left\{\ Inv \wedge EInv\ \right\}$
initially *Init*
method $m_{i\in 1..q} = body_i$
end

Proof

The proof relies on Definition Coerced Invariants (4.28) and rule Conjoin Extant Invariant (5.10).

QED

Postulate 5.12 (Conjoin Extant Dynamic Constraint)
In a similar manner, extant dynamic constraints can be strengthened. When strengthening extant dynamic constraints, each method should be encapsulated in the following manner.

body

becomes

$$
\begin{array}{l}
|\,[\,\textbf{con}\ \vec{U} \bullet \\
\quad \{\vec{u} = \vec{U}\} \\
\quad \mathit{body} \\
\quad [\,D[\vec{u_0} \backslash \vec{U}]\,] \\
]\,|
\end{array}
$$

When *body* is a specification, e.g., $\vec{w}$**:** $[\,p\ ,\ q\,]$, this simplifies to:

$\vec{w}$**:** $[\,p\ ,\ q \wedge D\,]$

Postulate 5.13 (Conjoin Coerced Dynamic Constraint)
Coerced dynamic constraints can be strengthened without modification to the remainder of the object.

Proof

The proof relies on Definition Coerced Dynamic Constraint (4.31) and rule Conjoin Extant Dynamic Constraint (5.12).

QED

Refining in the presence of Invariants and Dynamic Constraints When extant invariants or extant dynamic constraints are declared in an object, refining the methods of that object may cause the extant invariant or extant dynamic constraint to be violated. For example, consider the following specification which assigns a method parameter *b* to *c* in the context of an extant invariant which constrains *c* to be larger than zero ($c > 0$).

c**:** $[\,b > 0\ ,\ c = b\,]$

The classical refinement calculus allows the precondition to be weakened:

c**:** $[\,\mathit{True}\ ,\ c = b\,]$

However, an object with this specification as a method is no longer well-formed as the invariant may be broken if $b \leq 0$ initially.

The violation of extant invariants and extant dynamic constraints occurs because the object's methods are refined to increase the reachable states or scope of the object. The new reachable states are outside those for which the invariant is safely maintained. The properties are violated, however, only when the client takes advantage of the increased scope.

The coerced invariant and coerced dynamic constraint are not violated by refining the method bodies of an object when only a coerced invariant or coerced dynamic constraint is declared (no extant invariant or extant dynamic constraint) as the methods are coerced to satisfy them.

To solve this problem the validity of the extant invariant and extant dynamic constraint as presented in Definitions Valid Extant Invariant (4.26) and Valid Extant Dynamic (4.30) must be rechecked after each refinement. The following rule is a contextual refinement rule that already includes the validity check.

Theorem 5.14 (Revised Weaken Pre-condition)
When the method body is a specification, $\vec{w}: \left[Pre \ , \ Post \right]$, the precondition of the specification can be weakened provided that the invariant, weakened precondition and the postcondition are strong enough to reestablish the invariant and the dynamic constraint. Given the invariant *Inv*, the weakened precondition *Pre′*, and the dynamic constraint *D*:

$$Inv \Rrightarrow (Pre' \Rightarrow (\forall \vec{w} \bullet Post \Rightarrow (Inv \wedge D))[\vec{v_0} \backslash \vec{v}])$$

where $\vec{v}$ is a vector containing all environment variables.

Proof
The theorem is merely an expansion of definitions. The validity check to be shown is:

$Inv \wedge \vec{V} = \vec{v} \Rrightarrow wlp.(\vec{w}: \left[Pre' \ , \ Post \right]).(Inv \wedge D[\vec{v_0} \backslash \vec{V}])$
$\equiv$ Wlp Specification (2.12)
$Inv \wedge \vec{V} = \vec{v} \Rrightarrow Pre' \Rightarrow (\forall \vec{w} \bullet Post \Rightarrow (Inv \wedge D[\vec{v_0} \backslash \vec{V}]))[\vec{v_0} \backslash \vec{v}]$
$\equiv$ Substitutions
$Inv \Rrightarrow (Pre' \Rightarrow (\forall \vec{w} \bullet Post \Rightarrow (Inv \wedge D))[\vec{v_0} \backslash \vec{v}])$

QED

Not all classical refinement rules require new versions as the original refinement rule does not increase the scope of the object. For example, Morgan provides a rule for strengthening the postcondition that maintains extant invariants and extant dynamic constraints as it merely decreases the non-determinism of the object. This rule can, however, be revised to also use knowledge of the invariant.

Theorem 5.15 (Revised Strengthen Postcondition) **Proof on page 90**
The postcondition in the method $\vec{w}: \left[Pre \ , \ Post \right]$ can be strengthened to *Post′* when

$$(Inv \wedge Pre)[\vec{w} \backslash \vec{w_0}] \wedge Post' \Rrightarrow Post$$

Proof of 5.15 from p89 (Revised Strengthen Postcondition)
The validity check of the method with the stronger postcondition is:

$$Inv \wedge \vec{V} = \vec{v} \Rrightarrow wlp.(\vec{w}\!:\!\left[\, Pre \;,\; Post' \,\right]).(Inv \wedge D[\vec{v_0}\backslash\vec{V}])$$

Similar to the proof of Revised Weaken Pre-condition (5.14) this reduces to:

$$Inv \wedge \vec{V} = \vec{v} \Rrightarrow (Pre \Rightarrow (\forall\, \vec{w} \bullet Post' \Rightarrow (Inv \wedge D[\vec{v_0}\backslash\vec{V}]))[\vec{v_0}\backslash\vec{v}])$$

However, it is known that:

$$Inv \wedge \vec{V} = \vec{v} \Rrightarrow wlp.(\vec{w}\!:\!\left[\, Pre \;,\; Post \,\right]).(Inv \wedge D[\vec{v_0}\backslash\vec{V}])$$

which, similarly, is:

$$Inv \wedge \vec{V} = \vec{v} \Rrightarrow (Pre \Rightarrow (\forall\, \vec{w} \bullet Post \Rightarrow (Inv \wedge D[\vec{v_0}\backslash\vec{V}]))[\vec{v_0}\backslash\vec{v}])$$

The proof can be completed given $(Inv \wedge Pre)[\vec{w}\backslash\vec{w_0}] \wedge Post' \Rrightarrow Post$.

QED

A complementing solution to re-checking the validity of the object is to introduce *private* refinement. When a client uses a privately refined object it must must adhere to the specification of the *public* object of which the object is a private refinement of. The object can consequently increase its scope, for instance by arbitrarily weakening preconditions. Since the client cannot use the object outside the original scope, the problems of violating extant invariants and extant dynamic constraints cannot occur. Private refinement should be used to implement a class. The benefit of private refinement is that the classical refinement rules can be used. Public refinement should be used to declare a new sub-class.

For the example specification presented above, that is, the method $c\!:\!\left[\, b > 0 \;,\; c = b \,\right]$ in an object with the extant invariant $c > 0$, the method can be privately refined by weakening the precondition to *True*. Since the client uses the public specification it will establish the precondition $b > 0$ before calling the method. Consequently, although the invariant and the method are not strong enough by themselves to reestablish the invariant, the method will only be called in a situation where it will reestablish the invariant.

Summary One difficulty that invariants introduce is the loss of extant invariants when method bodies are refined. This problem is rectified here by introducing revised refinement rules that maintain extant invariants when methods are refined. An alternative, private refinement technique which also solves this problem is introduced. The advantage of private refinement is that the classical refinement calculus laws can be used.

5.2 Refinement of Client Constructs

This section presents several refinement rules involving *client constructs*: the language constructs that use objects. Without restrictions, object-refinement is not monotonic. Section 5.2.1 shows a solution, inspired by Naumann [Nau94b], that constrains the semantics, allowing object-refinement to be monotonic. Section 5.2.2 presents the definition of the **new** language construct found in practical object-oriented languages. It is defined as a language construct for a semantics for references and provides the capability to non-deterministically choose a store location and clone (copy) an object into that location. Section 5.2.3 redefines the field update language constructs, to permit monotonic object-refinements of fields. Section 5.2.4 provides rules that allow method calls to be introduced.

5.2.1 Object-Assignment

When predicates are permitted to be arbitrary sets of states, object-refinement is not monotonic. For instance, given objects e and f where f is an object-refinement of e, $e \sqsubseteq_\varsigma f$, and an assignment statement that assigns e to o, then it is not a valid refinement to replace e with f:

$$o := e \not\sqsubseteq o := f$$

The assignment $o := e$ establishes the postcondition $o = e$. In contrast, the assignment $o := f$ establishes $o = f$. Since $o = e$ does not entail $o = f$, that is $o = e \not\Rightarrow o = f$, the refinement does not hold. To obtain object-refinement monotonicity, an approach similar to that suggested by Naumann [Nau94b] is taken: predicates are restricted to those monotonic under object-refinement. This is achieved by only permitting predicates that are *upwards closed* under object-refinement. This subset of predicates is also known as the *Alexandrov topology*. In general, for a poset X (with the ordering $\leq$), a subset ϕ of X is upwards closed (up-closed) when

$$\forall a, b : X \bullet a \leq b \Rightarrow (a \in \phi \Rightarrow b \in \phi)$$

By applying the following constraint to predicates, the monotonicity of predicates under object-refinement can be guaranteed. For objects e and f, and predicates $p(e)$ and $p(f)$,

$$\forall e, f \bullet e \sqsubseteq_\varsigma f \Rightarrow (p(e) \Rightarrow p(f))$$

In practice this restriction means that the predicates must be monotonic in the program's variables and logical constants, but not necessarily in explicit object constructions. For variable o and explicit object e, predicates such as $o \sqsupseteq_\varsigma e$ are monotonic in o and are consequently allowed. Predicates such as $o = e$ and $o \sqsubseteq_\varsigma e$ are not permitted as they are not monotonic in o: they are satisfied by the object o but not by all proper object-refinements of o.

For basic types the restriction to monotonic predicates has no significance. This is because object-refinement reduces to equality for these types and all predicates are monotonic under the equality relation: $l = m \Rightarrow p(l) = p(m)$. Consequently, the results of the classical refinement calculus are upheld.

To ensure that the calculus does not introduce non-monotonic predicates the language constructs must be shown to maintain upwards closure. Proofs that the standard guarded command language constructs maintain the monotonicity of predicates under refinement are presented by King [Kin99, p76]. The proofs for object-refinement (instead of refinement) are essentially the same. King omits the proofs for logical constants and local variables, however, these are shown trivially given that the bodies of the logical constant and local variable blocks preserve object-refinement monotonicity.

To preclude specification statements from introducing non-monotonic predicates, King constrains the precondition to be monotonic and the postcondition to be anti-monotonic in variables. Theorem 5.16 generalises those results by relaxing the anti-monotonicity constraints in the postcondition on the variables in the frame. Consequently, only the variables in the postcondition that are not in the frame (and any initial variables) are required to be anti-monotonic.

Theorem 5.16 (Specifications Object-Refinement Monotonicity)
Informally, a specification statement with appropriately constrained precondition and postcondition preserves object-refinement monotonicity. More formally, given

- program variables $\vec{x} \cup \vec{y}$ where $\vec{x}$ and $\vec{y}$ are disjoint;
- frame variables $\vec{y}$;
- the monotonicity of $pre(\vec{x}, \vec{y})$ for $\vec{x}$ and $\vec{y}$;
- the anti-monotonicity of $post(\vec{x}, \vec{y}, \vec{y_0})$ for $\vec{x}$ and $\vec{y_0}$;
- a predicate $\phi(\vec{x}, \vec{y})$ object-refinement monotonic on the variables $\vec{x}$ and $\vec{y}$; and
- variable vectors $\vec{x'}$ and $\vec{y'}$ such that $\vec{x} \sqsubseteq_\varsigma \vec{x'}$ and $\vec{y} \sqsubseteq_\varsigma \vec{y'}$

then the predicate

$$y{:}\left[\, pre(\vec{x}, \vec{y}) \;,\; post(\vec{x}, \vec{y}, \vec{y_0}) \,\right] \; \phi(\vec{x}, \vec{y})$$

is object-refinement monotonic in $\vec{x}$ and $\vec{y}$.

Proof

Hence, given the assumptions, show that

$$\begin{array}{l} y{:}\left[\, pre(\vec{x}, \vec{y}) \;,\; post(\vec{x}, \vec{y}, \vec{y_0}) \,\right] \; \phi(\vec{x}, \vec{y}) \\ \Rrightarrow \\ (y{:}\left[\, pre(\vec{x}, \vec{y}) \;,\; post(\vec{x}, \vec{y}, \vec{y_0}) \,\right] \; \phi(\vec{x}, \vec{y}))[\vec{x}, \vec{y} \backslash \vec{x'}, \vec{y'}] \end{array}$$

$\equiv$ Specification Statement (A.48)

$pre(\vec{x},\vec{y}) \wedge (\forall \vec{y} \bullet post(\vec{x},\vec{y},\vec{y_0}) \Rightarrow \phi(\vec{x},\vec{y}))[\vec{y_0}\backslash\vec{y}]$
$\Rrightarrow$
$(pre(\vec{x},\vec{y}) \wedge (\forall \vec{y} \bullet post(\vec{x},\vec{y},\vec{y_0}) \Rightarrow \phi(\vec{x},\vec{y}))[\vec{y_0}\backslash\vec{y}])[\vec{x},\vec{y}\backslash\vec{x'},\vec{y'}]$

$\equiv$ Substitution

$pre(\vec{x},\vec{y}) \wedge (\forall \vec{y} \bullet post(\vec{x},\vec{y},\vec{y_0}) \Rightarrow \phi(\vec{x},\vec{y}))[\vec{y_0}\backslash\vec{y}]$
$\Rrightarrow$
$pre(\vec{x'},\vec{y'}) \wedge (\forall \vec{y} \bullet post(\vec{x},\vec{y},\vec{y_0}) \Rightarrow \phi(\vec{x},\vec{y}))[\vec{x},\vec{y_0}\backslash\vec{x'},\vec{y'}]$

$\equiv$ Substitutions and renaming of bound quantifier

$pre(\vec{x},\vec{y}) \wedge (\forall \vec{z} \bullet post(\vec{x},\vec{z},\vec{y}) \Rightarrow \phi(\vec{x},\vec{z}))$
$\Rrightarrow$
$pre(\vec{x'},\vec{y'}) \wedge (\forall \vec{z} \bullet post(\vec{x'},\vec{z},\vec{y'}) \Rightarrow \phi(\vec{x'},\vec{z}))$

$\Lleftarrow$ Monotonicity of $pre(\vec{x},\vec{y})$ with respect to $\vec{x}$ and $\vec{y}$, $\vec{x} \sqsubseteq_\varsigma \vec{x'}$ and $\vec{y} \sqsubseteq_\varsigma \vec{y'}$.

$(\forall \vec{z} \bullet post(\vec{x},\vec{z},\vec{y}) \Rightarrow \phi(\vec{x},\vec{z}))$
$\Rrightarrow$
$(\forall \vec{z} \bullet post(\vec{x'},\vec{z},\vec{y'}) \Rightarrow \phi(\vec{x'},\vec{z}))$

$\Lleftarrow$ Universal Quantification Weak Distribution (A.19)

$(\forall \vec{z} \bullet post(\vec{x},\vec{z},\vec{y}) \Rightarrow \phi(\vec{x},\vec{z}) \Rightarrow post(\vec{x'},\vec{z},\vec{y'}) \Rightarrow \phi(\vec{x'},\vec{z}))$

Given that $post(\vec{x},\vec{y},\vec{y_0})$ is anti-monotonic for $\vec{x}$ and $\vec{y_0}$, then $post(\vec{x'},\vec{z},\vec{y'})$ is anti-monotonic for $\vec{x'}$ and $\vec{y'}$.

$\Lleftarrow (\forall \vec{z} \bullet \phi(\vec{x},\vec{z}) \Rightarrow \phi(\vec{x'},\vec{z}))$

This is shown using the assumption that $\phi(\vec{x},\vec{y})$ is monotonic for $\vec{x}$ and $\vec{y}$.

QED

By relaxing the anti-monotonicity of frame variables in the postcondition, specifications such as (for variables o and p)

$$o\mathbf{:}\left[p \sqsubseteq_\varsigma o\right]$$

can be used. This specification statement appears to be able to establish $p \sqsubseteq_\varsigma o$. Unfortunately, the predicate $p \sqsubseteq_\varsigma o$ is not monotonic for p. However, for an explicit object l, the weakest precondition of

$$o\mathbf{:}\left[p \sqsubseteq_\varsigma o\right]$$

with respect to $l \sqsubseteq_\varsigma o$ is $l \sqsubseteq_\varsigma p$. Since this is true for all ls:

$$\forall l \bullet l \sqsubseteq_\varsigma p \equiv (o\mathbf{:}\left[p \sqsubseteq_\varsigma o\right]\ l \sqsubseteq_\varsigma o)$$

this predicate can be used in lieu of the non-monotonic predicate $p \sqsubseteq_\varsigma o$.

The following refinement rules extend (or explicitly prove) the works of Naumann and King with the properties required in this thesis. The first two refinement rules show that object-refining objects in assignments and specification statements produces a refinement.

Theorem 5.17 (Object-Refine in Assignment (Semantics for Values)) **Proof on page 176**

For object variable o and (object) expressions e and f:

$$\frac{e \sqsubseteq_\varsigma f}{o := e \sqsubseteq o := f}$$

That is, substituting e for f in the assignment $o := e$ produces the refined statement $o := f$.

Theorem 5.18 (Object-Refine in Specification (Semantics for Values)) **Proof on page 177**

For object variable o and (object) expressions e and f:

$$\frac{e \sqsubseteq_\varsigma f}{o\mathbf{:}\left[e \sqsubseteq_\varsigma o\right] \sqsubseteq o\mathbf{:}\left[f \sqsubseteq_\varsigma o\right]}$$

Substituting f for e in the specification $o\mathbf{:}\left[e \sqsubseteq_\varsigma o\right]$ produces the refined statement $o\mathbf{:}\left[f \sqsubseteq_\varsigma o\right]$.

The following theorems are the analogies of the simple specification [Mor94] refinement rule for the Alexandrov topology.

Theorem 5.19 (Object-Refinement Specification) **Proof on page 178**

For constant expression e:

$$o\mathbf{:}\left[e \sqsubseteq_\varsigma o\right] \sqsubseteq o := e$$

For discretely-ordered types (i.e., basic types) the refinement relation can be strengthened to equality.

$$o\mathbf{:}\left[e \sqsubseteq_\varsigma o\right] \equiv o := e$$

Additionally, for discretely-ordered types, the object-refinement relation reduces to equality (Object-Refinement Basic Types (5.2)).

$$o\mathbf{:}\left[o = e\right] \equiv o := e$$

Together, these properties can be used to show the Simple Specification (A.52) rule as a corollary of the above theorem.

For a semantics of references, the theorems provided above are upheld, but since *Ref* is a discretely-ordered type the theorems are of little practical significance. Object-assignment for a semantics for references is now discussed and analogies of the theorems presented above are provided. This work integrates the Alexandrov topology with the definitions found in Section 4.7.

An important construct for a semantics for references is the assignment of references (termed *reference assignment*). Reference assignment modifies an object reference to point to a different location in the store. Indirectly this alters the value of the dereferenced object, yet this does not alter the store. It may break an alias that previously existed on the original store location and may also form a new alias on the new store location. The following syntax is used to denote the reference assignment of reference p to reference q.

$$p := q$$

Given an object-refinement $s\uparrow$ of $q\uparrow$, it is not, in general, a refinement to substitute s for q as they may point to different store locations. For example, given s and q such that their dereferences are equal, yet they reference different store locations ($s\uparrow = q\uparrow \wedge\ s \neq q$) then the code

$p := q$;
q.**update** f **with** e

has the effect of pointing p to q and then updating q's f field, and consequently also p's f field. Replacing q with s in the assignment would mean that the update of p's field no longer occurs. This behaviour is not a refinement of the original.

Another form of assignment within a semantics for references is *dereferenced assignment*. Dereference assignment alters the value stored in a store location. This may cause, through aliasing, the alteration of other state variables. The following syntax is used to denote the dereference assignment of expression p to the reference variable o.

$$o\uparrow := p$$

Since dereference assignments manipulate objects, opportunities arise for the object-refinement of the objects used. The following refinement rules are analogous to those provided earlier for a semantic for values.

Postulate 5.20 (Object-Refine in Assignment (Semantics for References))
Object-refining an object in a dereference assignment produces a refinement. For object variable o and (object) expressions e and f:

$$\frac{e \sqsubseteq_{\varsigma} f}{o\uparrow := e \sqsubseteq o\uparrow := f}$$

Postulate 5.21 (Object-Refinement Specification (Semantics for References))
This refinement is analogous to Simple Specification (A.52) except this rule is applicable to a semantics for references. For object variable o and (object) expression e, the alteration of the store at o to establish $o\uparrow$ as an object-refinement of e can be implemented by assigning e to $o\uparrow$.

$$o\uparrow :: \left[e \sqsubseteq_\varsigma o\uparrow \right] \sqsubseteq o\uparrow := e$$

5.2.2 Reference Cloning

Reference cloning is a construct designed specifically for a semantics for references to encapsulate the non-determinism of choosing a reference location; the refinement of which is a task for the compiler. Traditionally, the allocation of a reference location is accompanied by the initialisation of the data at that location. Reference cloning achieves these functions by copying an object value into an unused reference location. Given an object reference o, the following assigns o to an unused store location and copies the object denoted by e into that store location.

$$o := \textbf{new}\ e$$

Definition 5.22 (New Operator) The **new** language construct is defined as follows.

$$\begin{array}{l} o := \textbf{new}\ e \\ \widehat{=} \\ \sqcap x \mid x \in \mathit{Ref} \setminus (\bigcup i \bullet \mathrm{dom}\, \mathit{store}_i) \bullet o := x; \\ o\uparrow := e \end{array}$$

Since multiple stores are used, x is chosen such that it is not in the domain of any store.
$\diamond$

Since **new** is defined using an object-assignment of e to $o\uparrow$, the object-refinement of e is permitted. This refinement rule is termed *construction monotonicity*.

Theorem 5.23 (Construction Monotonicity)
Given objects e and f,

$$\frac{e \sqsubseteq_\varsigma f}{o := \textbf{new}\ e \sqsubseteq o := \textbf{new}\ f}$$

Proof

The proof is a straightforward application of Theorem Object-Refine in Assignment (Semantics for References) (5.20).
QED

5.2.3 Field Updates

This section provides refinement rules for the object-refinement of the value being assigned in a field update construct.

Postulate 5.24 (Object-Refine Field Update (Semantics for Values))
This refinement rule allows the expression being assigned in a field update to be object-refined.

$$\frac{e \sqsubseteq_\varsigma f}{(o.\textbf{update}\ \mathit{fld}\ \textbf{with}\ e) \sqsubseteq (o.\textbf{update}\ \mathit{fld}\ \textbf{with}\ f)}$$

Postulate 5.25 (Object-Refine Field Update (Semantics for References))
This refinement rule allows the expression being assigned in a field update to be object-refined for a semantics for references.

$$\frac{e \sqsubseteq_\varsigma f}{(o\uparrow.\textbf{update}\ \mathit{fld}\ \textbf{with}\ e) \sqsubseteq (o\uparrow.\textbf{update}\ \mathit{fld}\ \textbf{with}\ f)}$$

5.2.4 Introduce Method Calls

This section illustrates the refinement rules that can be used to introduce method calls. The proofs of such rules rely on the definitions for method invocation (Definitions 4.49 and 4.39).

Postulate 5.26 (Result Specification)
This rule permits the use of a result parameter in the method being introduced. This rule is drawn from Morgan's [Mor94] Law 11.4.

Given the formal result parameter f, the actual result parameter a (both disjoint from w), and a method that refines

$$\textbf{method}\ \mathit{Meth}(\textbf{result}\ f : T) \ \widehat{=}\ w, f\textbf{:}\ [\mathit{pre}\ ,\ \mathit{post}[a\backslash f]\ ]$$

with *pre* containing no f and neither f nor f_0 occurring in *post*, then

$$w, a\textbf{:}\ [\mathit{pre}\ ,\ \mathit{post}\ ] \sqsubseteq \textbf{call}\ \mathit{Meth}(a)$$

Analogous rules exist for value parameters and value-result parameters.

5.3 Object-Refinement Simulation

Using an implementation object to 'simulate' a specification object is often desirable as it may, for example, introduce efficiencies. Given the specification object *Spec* and an object-refinement *Impl*, that is, $Spec \sqsubseteq_\varsigma Impl$, the program

$$
\begin{array}{l}
|[\ \textbf{var}\ obj : \tau(Spec) \bullet \\
\quad obj := \textbf{new}\ Spec;\ Prog \\
]|
\end{array}
$$

may be refined to use *Impl* instead:

$$
\begin{array}{l}
|[\ \textbf{var}\ obj : \tau(Spec) \bullet \\
\quad obj := \textbf{new}\ Impl;\ Prog \\
]|
\end{array}
$$

Notice that the type of *obj* is still *Spec*. This is type correct as all instances of *Impl* are by subsumption instances of *Spec*. As a consequence, however, *Prog* is not permitted to use any of the attributes in *Impl* that are not in *Spec*[2]. The following theorem is a corollary of Theorem 5.23.

Corollary 5.27 (Object-Refinement Simulation)

$$
\frac{Spec \sqsubseteq_\varsigma Impl}{
\begin{array}{l}
|[\ \textbf{var}\ obj : Ref\ \tau(Spec) \bullet obj := \textbf{new}\ Spec;\ Prog\ ]| \\
\qquad\qquad \sqsubseteq \\
|[\ \textbf{var}\ obj : Ref\ \tau(Spec) \bullet obj := \textbf{new}\ Impl;\ Prog\ ]|
\end{array}
}
$$

This property is achieved by ensuring that the client constructs used in *Prog* can only assume that they are using some object-refinement of *Spec*.

5.4 Data-refinement for Objects

This section uses a specialisation of data-refinement to provide an *object-data-refinement* relation that permits the data-refinement of an object's private data-structures while requiring no change to the client's code. After the object-data-refinement, the client can be modified to use the new attributes of the implementation object. The refinement of the client to use the new attributes of the implementation object is termed *client enhancement*.

Example 5.28 (Object-Data-Refinement Client Enhancement) The following is an example of the manner in which a client can be 'enhanced' subsequent to an object-data-refinement.

[2]The technique of construction monotonicity is revisited in Section 5.4. There, a more general technique, *simulation*, is introduced. Simulation provides possibilities for enhancement of the client code by allowing the original method calls to be replaced with (perhaps more efficient) calls to the new methods.

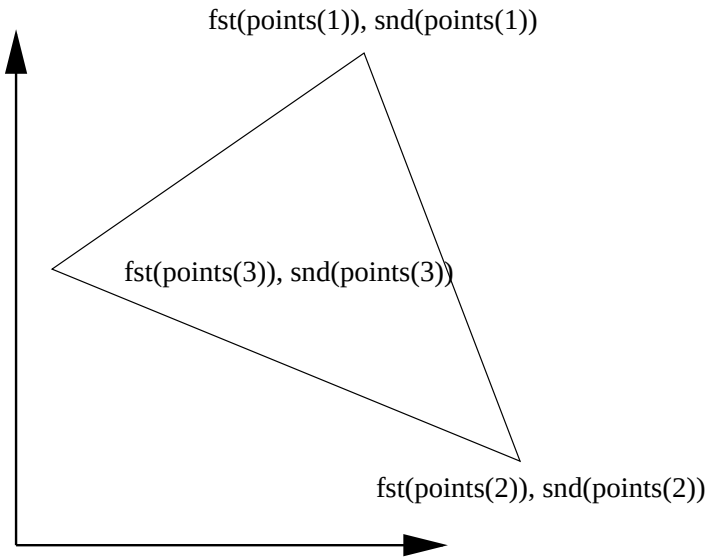

Figure 5.1: Abstract Triangle

The example uses a specification object *Triangle*, represented using (a sequence of) three points. Each point is represented as a pair of natural numbers, where the first element represents the x coordinate, and the second element of the pair represents the y coordinate. For example, $fst(points(3))$ represents the x coordinate of the third point, while $snd(points(3))$ represents the third point's y coordinate. The specification data type is illustrated in Figure 5.1. *Triangle* also has methods for setting the points and obtaining the distances between any two points.

Triangle :=
object
private field *points* : $\text{seq}_3(\mathbb{N} \times \mathbb{N}) := \langle (0,0),(0,0),(0,0) \rangle$
method *setPoints*(**value** *pnts* : $\text{seq}_3(\mathbb{N} \times \mathbb{N})$) =
points := *pnts*
method *getDistance*(**value** $p1 : 1..2$, **value** $p2 : 2..3$, **result** *distance* : $\mathbb{R}$) =
$\{p1 \neq p2\}$

$$distance := \sqrt{\begin{array}{l}(fst(points(p1)) - fst(points(p2)))^2 + \\ (snd(points(p1)) - snd(points(p2)))^2\end{array}}$$

end

Using an unspecified object-data-refinement, an implementation object *ConcreteTriangle* is calculated. *ConcreteTriangle* has an invariant that restricts its first point to the origin, its second point to the y axis and its third to the x axis, thereby forcing a right-angled triangle. This strengthened invariant must, during the proof of the object-data-refinement, be shown to hold using properties of the client[3]. As shown in Figure 5.2, the object *ConcreteTriangle* represents the resulting triangle using two lengths: the first being the

[3]Such an object-data-refinement can be shown if the client consistently sets the first point to the origin, its second to the y axis and third to the x axis.

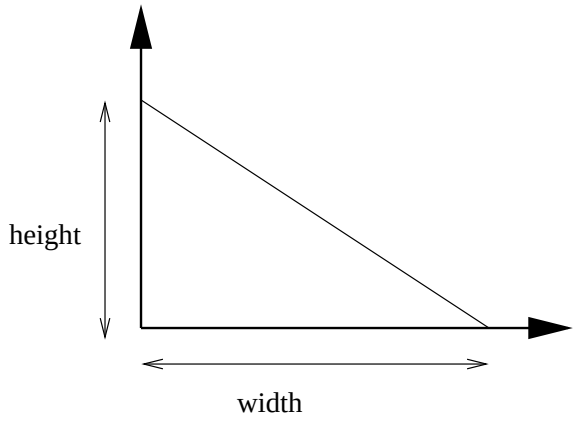

Figure 5.2: Concrete Triangle

height of the triangle and the second being the length of its base. It also has, for efficiency, a *perimeter* field and a new *getPerimeter* method.

object
  private field *height* : $\mathbb{N} := 0$
  private field *width* : $\mathbb{N} := 0$
  private field *perimeter* : $\mathbb{R} := 0$
  method *setPoints*(**value** *pnts* : $\text{seq}_3(\mathbb{N} \times \mathbb{N})$) =
    $\{fst(pnts(1)) = 0 = snd(pnts(1)) \wedge$
    $fst(pnts(2)) = 0 \wedge snd(pnts(3)) = 0\}$;
    height := $snd(pnts(2))$;
    width := $fst(pnts(3))$;
    perimeter := $\sqrt{height^2 + width^2} + width + height$;
  method *getDistance*(**value** $p1 : 1..2$, **value** $p2 : 2..3$, **result** *distance* : $\mathbb{R}$) =
    $\{p1 \neq p2\}$
    if $(p1 = 1)$ **then**
      if $(p2 = 2)$ **then** *distance* := *height*
      else *distance* := *width*;
      end
    else
      distance := $\sqrt{height^2 + width^2}$
    end
  method *getArea*(**result** *area* : $\mathbb{R}$) =
    area := $width * height/2.0$
  method *getPerimeter*(**result** *peri* : $\mathbb{R}$) =
    peri := *perimeter*;
end

Consequently, ‘old’ client code that calculates the perimeter can be refined to a call to

getPerimeter.

$$
\begin{array}{l}
\left|\left[\ \textbf{var}\ x, y, z : \mathbb{N} : o.\textbf{call}\ \mathit{getDistance}(1, 2, x);\quad \sqsubseteq\right.\right. \\
o.\textbf{call}\ \mathit{getDistance}(2, 3, y); \\
o.\textbf{call}\ \mathit{getDistance}(1, 3, z); \\
p := x + y + z \bullet \\
\\
\left.\left]\right|\right. \\
o.\textbf{call}\ \mathit{getPerimeter}(p)
\end{array}
$$

$\diamond$

When data-refining a local variable block in the classical refinement calculus, only the code inside the local variable block has access to the specification or implementation variable. The specification/implementation variables are consequently inaccessible to, or hidden from, the code outside the variable block. In a similar fashion, to effect a data-refinement of objects there must be some state that is inaccessible to the client code. It is the modification of the content and behaviour of the inaccessible fields that forms the data-refinement of objects. To effect a hiding of 'state', as seen in Section 4.5.1, the type of an object is masked by reporting to a client a supertype rather than the actual type.

An object-data-refinement is achieved by replacing the appropriate private fields, and data-refining the methods of an object. A consistency check between the initial values of the implementation and specification variables is also required. The object-data-refinement relation is formalised in Definition 5.29.

Definition 5.29 (Object-Data-Refinement) Object-data-refinement is a specialisation of statement data-refinement in which the implementation public type is a subtype of the specification public type (Line 5.1 below). This constraint allows specialised rules to be developed that allow the implementation client code to remain syntactically the same as the specification client code. After the object-data-refinement is completed, it is possible to further enhance the implementation client code using the new public attributes of the implementation object.

To define object-data-refinement, the private attributes being removed (*PrivateSpec*) and those being added (*PrivateImp*) must be identified. For the specification object *spec*, the private attributes are those remaining after the public attributes ($\tau_{Public}(spec)$) have been 'subtracted' from the actual type. Alternatively, the private and public types should have no attributes in common, while together they should form the actual type:

$$(\tau_{Public}(spec) \sqcup PrivateSpec) = Top \wedge (\tau_{Public}(spec) \sqcap PrivateSpec) = \tau(spec)$$

Similarly, using the implementation object *impl*, the implementation private attributes *PrivateImp* can be identified:

$$(\tau_{Public}(impl) \sqcup PrivateImp) = Top \wedge (\tau_{Public}(impl) \sqcap PrivateImp) = \tau(impl)$$

All public methods of the specification type must data-refine through the chosen *rep* (Line 5.2). All public fields of the specification type must object-refine to the corresponding implementation fields (Line 5.3). Finally, the private implementation fields must be set to a value allowed by the simulation of the private specification fields under *rep* (Line 5.4)[4].

Given object variables *spec* and *impl* in a state *s*, and types containing only the private specification and implementation attributes (*PrivateSpec* and *PrivateImp* respectively) object-data-refinement is defined as follows:

$$
\begin{array}{ll}
spec \preceq_{ODR} impl \;\widehat{=}\; & \\
\quad \tau_{Public}(impl) \preceq \tau_{Public}(spec) \wedge & (5.1)\\
\quad \forall j \in \mathrm{dom}\, methods(\tau_{Public}(spec)) \bullet spec_{\odot} j \preceq_{DR} impl_{\odot} j & (5.2)\\
\quad \forall j \in \mathrm{dom}\, fields(\tau_{Public}(spec)) \bullet spec_{\odot} j \sqsubseteq_{\varsigma} impl_{\odot} j & (5.3)\\
\quad (\forall j \in \mathrm{dom}\, fields(PrivateImp) \bullet impl_{\odot} j = s(impl)_{\odot} j) \Rrightarrow & \\
\qquad rep(\forall j \in \mathrm{dom}\, fields(PrivateSpec) \bullet spec_{\odot} j = s(spec)_{\odot} j)) & (5.4)
\end{array}
$$

◇

5.4.1 Object Invariants and History Properties

This section presents rules for the private data-refinement of an object[5] involving an *invariant* or *dynamic constraint.* This thesis only considers private data-refinement for objects involving invariants and dynamic constraints. The lack of public data-refinement is not a concern as data-refinement is typically an implementation development path. Public refinement, in contrast, is concerned more with the introduction of a sub-class. Sub-classing is used in practice to **specify** differing behaviour—not to implement a class.

The private data-refinement of an object involving an invariant (or dynamic constraint) requires the transformation of that invariant (dynamic constraint).

Theorem 5.30 (Data-Refinement of an Extant Invariant) **Proof on page 103**

For the data-refinement transformer $rep\ p \;\widehat{=}\; \exists \vec{s} \bullet CInv \wedge p$ where *CInv* is the coupling invariant, $\vec{s}$ the specification variables and $\vec{i}$ the implementation variables, then the specification invariant *SInv* is transformed to the implementation invariant *IInv* defined as follows.

$$IInv \;\widehat{=}\; (\forall \vec{s} \bullet CInv \Rightarrow SInv)$$

The proof relies on showing that

$$\frac{prog_s \preceq_{DR} prog_i}{\{SInv\};\ prog_s;\ \lceil SInv \rceil \preceq_{DR} \{IInv\};\ prog_i;\ \lceil IInv \rceil}$$

[4]This property is analogous to the constructor refinement criteria of Back, Mikhajlova and von Wright [BMvW97, p18].

[5]The distinction between public and private data-refinement was made on page 90.

Proof of 5.30 from p102 (Data-Refinement of an Extant Invariant)

$\{SInv\};\ \ prog_s;\ \ \lceil SInv \rceil$
$\sqsubseteq$ Remove Assertion (A.30)
$prog_s;\ \ \lceil SInv \rceil$
$\preceq_{DR}$ Data-Refine Guard (A.56)
$prog_i;\ \ \lceil \forall s \bullet CInv \Rightarrow SInv \rceil$

Since the code occurs in a protected system the invariant (*IInv*) can be asserted at the start of the code.

$\equiv$ Introduce Protected System Assertion (4.27)
$\{IInv\};\ \ prog_i;\ \ \lceil IInv \rceil$

QED

Postulate 5.31 (Data-Refinement of an Extant Dynamic Constraint)
Dynamic constraints must be transformed in a slightly different fashion. Dynamic constraints also reference initial variables which must be transformed. The zero-subscripted coupling invariant $CInv_0$ is used for this purpose. The quantification of the specification variables extends also to the initial variables. For the specification dynamic constraint *SD*, the implementation dynamic constraint *ID* is defined as:

$$ID \mathrel{\widehat{=}} (\forall \vec{s}, \vec{s_0} \bullet CInv \wedge CInv_0 \Rightarrow SD)$$

The proof relies on showing that

$$\frac{prog_s \preceq_{DR} prog_i}{\begin{array}{l} \lvert\lvert \lceil \mathbf{con}\ \vec{S} \bullet \\ \qquad \{\vec{s} = \vec{S}\};\ \ prog_s;\ \ \lceil SD[\vec{s_0} \backslash \vec{S}] \rceil \\ \rceil\rvert\rvert \\ \quad \preceq_{DR} \\ \lvert\lvert \lceil \mathbf{con}\ \vec{I} \bullet \\ \qquad \{\vec{i} = \vec{I}\};\ \ prog_i;\ \ \lceil ID[\vec{i_0} \backslash \vec{I}] \rceil \\ \rceil\rvert\rvert \end{array}}$$

By combining the rules for data-refining initialisations presented by Morgan [Mor94], the following rule is produced.

Law 5.32 (Data-Refinement of Initialisation) Given the initialisation *Init*:

$$Init \preceq_{DR} (\exists \vec{s} \bullet CInv \wedge Init)$$

Example 5.33 (Data-Refinement of Object Properties) For example, consider an object with an integer field s, an invariant $s \in \textbf{Even}$ denoting that s is an even number and dynamic constraint $s > s_0$ denoting that s can only be increased.

object
  field $s : \mathbb{Z}$
  invariant $[\ s \in \textbf{Even}\ ]$
  initially $s = 0$
  dynamic $[\ s > s_0\ ]$
  method *increase* $= s := s + 2$
end

This object can be data-refined to an object with an integer field i where i is half the value of s. Using the coupling invariant $s = 2 * i$, the new invariant becomes

$(\forall \vec{s} \bullet CInv \Rightarrow SInv)$
$\equiv$
$(\forall \vec{s} \bullet s = 2 * i \Rightarrow s \in \textbf{Even})$
$\equiv$ One Point Rule
True

The dynamic constraint becomes

$(\forall \vec{s}, \vec{s_0} \bullet CInv \wedge CInv_0 \Rightarrow SD)$
$\equiv$
$(\forall \vec{s}, \vec{s_0} \bullet s = 2 * i \wedge s_0 = 2 * i_0 \Rightarrow s > s_0)$
$\equiv$ One Point Rule
$2 * i > 2 * i_0$
$\equiv$
$i > i_0$

Data-refining the initialisation and the method, as usual, produces the following object.

object
  field $i : \mathbb{Z}$
  dynamic $[\ i > i_0\ ]$
  initially $i = 0$
  method *increase* $= i := i + 1$
end

$\diamond$

5.4.2 Simulation of Object-Data-Refinements

This section presents a rule and corresponding proof that allows the simulation of an object by an object-data-refinement.

Postulate 5.34 (Object Simulation/Polymorphism)
Given specification object *spec*, and implementation object *impl* such that $spec \preceq_{ODR} impl$ under rep_{si} then:

$$
\begin{array}{l}
|[\ \mathbf{var}\ o : Ref\ \tau_{Public}(spec) \bullet \\
\quad o := \mathbf{new}\ spec;\ Prog \\
]| \\
\sqsubseteq \\
|[\ \mathbf{var}\ o : Ref\ \tau_{Public}(impl) \bullet \\
\quad o := \mathbf{new}\ impl;\ Prog \\
]|
\end{array}
$$

Using this rule (instead of Corollary 5.3) allows the client of the object to use the newly introduced attributes as illustrated by Example 5.28.

Proof

The proof is achieved by data-refining the store so that the object at location o is object-data-refined: $o\uparrow_{store_s} \preceq_{ODR} o\uparrow_{store_i}$ (under rep_{si}). The store should remain the same at all other locations: $\{o\} \lhd store_s = \{o\} \lhd store_i$. However, due to the restriction to monotonic predicates, the objects must be allowed to be object-refined. Consequently, *rep* is defined as follows.

$$
\begin{array}{l}
rep\ \widehat{=} \\
\lambda\, p \bullet (\exists\, store_s \bullet o\uparrow_{store_s} \preceq_{ODR} o\uparrow_{store_i} \wedge (\{o\} \lhd store_s) \sqsubseteq_{\varsigma} (\{o\} \lhd store_i) \wedge p)
\end{array}
$$

The statements leading up to the **new** operator do not involve $store(o)$ and consequently data-refine to themselves. The **new** construct data-refines to the implementation version by Data-Refine New (5.35).

$$o := \mathbf{new}\ spec \preceq_{DR} o := \mathbf{new}\ impl$$

The remainder of the client, *Prog*, data-refines to itself (under the substitution $[store_s \backslash store_i]$).

$$Prog \preceq_{DR} Prog[store_s \backslash store_i]$$

Consequently, the proof reduces to showing that all language constructs and connectives data-refine to themselves. The language constructs that do not involve dereferences of o data-refine to themselves, leaving only the statements that involve $o\uparrow$. The following data-refinement rules are illustrative of the fact that the statements involving $o\uparrow$ data-refine to themselves. These rules use *rep* for the data-refinement of the store, and rep_{si} for the object-data-refinement from *spec* to *impl*.

QED

Theorem 5.35 (Data-Refine New) **Proof on page 179**

$o := \textbf{new}\ spec \preceq_{DR} o := \textbf{new}\ impl$

Postulate 5.36 (Data-Refine Store in Method Call)
For public method m, under rep:

$o{\uparrow}_{store_s}.\textbf{call}\ m$
$\preceq_{DR}$
$o{\uparrow}_{store_i}.\textbf{call}\ m$

5.5 Class-Based Refinement

This section details the extension of the object-based refinement calculus to a class-based refinement calculus. Several principles are used to guide the class-based semantics.

- Classes are defined as objects whose main purpose is the creation of class instances.
- Subclassing is a mechanism for incremental extension of classes. Subclassing is regarded as the class equivalent of subtyping, i.e., syntactic conformance to a signature (but not necessarily behavioural consistency—the field and method types conform but the methods need not be refinements).
- Finally, class refinement is subclassing with an additional behavioural consistency constraint. This split approach to behavioural consistency permits additional freedom during specification, yet allows the refiner to maintain the integrity of critical class sub-hierarchies.

The class-based refinement calculus defined here is a definitional extension of the object-based refinement calculus.

Definition 5.37 (Classes) A class is defined as an object.

$$
\begin{array}{l}
|[\ \textbf{class}\ classname\ \textbf{is} \\
\quad \textbf{field}\ f_{i\in 1..fn} : F_i := fv_i \\
\quad \textbf{method}\ m_{i\in 1..mn} = mv_i \\
\textbf{end} \bullet \\
\quad Prog \\
\ ]| \\
\widehat{=} \\
|[\ \textbf{var}\ classname : \tau(classname) \bullet \\
\quad classname := \ \textbf{object} \\
\qquad\qquad\qquad \textbf{field}\ f_{i\in 1..fn} : F_i := fv_i \\
\qquad\qquad\qquad \textbf{method}\ m_{i\in 1..mn} = mv_i \\
\qquad\qquad \textbf{end} \\
\quad Prog \\
\ ]|
\end{array}
$$

◇

Classes defined in this fashion can be instantiated using the same mechanism that allows cloning in the object-based refinement calculus. That is, class instantiation for a semantics for values is achieved by object assignment or, for a semantics for references, reference cloning.

Manipulation of class attributes is permitted as it is envisaged that minor use of such facilities will prove beneficial. For instance, class manipulation could be used to ensure that only one class instance is created.

A class may be refined either through object-refinement or object-data-refinement. The introduction of a subclass that is also a class refinement maintains the behavioural consistency of the original class hierarchy.

The refinement rules of the object-based refinement calculus are upheld for classes. Additional rules, such as Class Introduction (5.38) can also be used to support the more structured development methods that class-based languages support.

Theorem 5.38 (Class Introduction) **Proof on page 180**

This refinement rule can be used to introduce a new class into an existing class hierarchy. Since a class definition is merely a scoped variable introduction with a specific object initialisation similar rules apply to class introduction as to variable introduction. Namely, a class with class name *Classname* can be introduced into a program *P* provided *Classname* is not free in *P* and *P* is conjunctive.

$$\frac{\mathit{Classname} \mathrel{\mathit{nfi}} P}{P \sqsubseteq \; |[\ \textbf{class}\ \mathit{Classname}\ \textbf{is}\ \ \mathit{Attribs}\ \ \textbf{end} \bullet\ \ P\ ]|}$$

where *Attribs* is a list of attributes.

Chapter 6

Towards A Reference Semantics

Object identity is an important aspect of object-oriented programming. Object-identities are used to ease the dynamic construction of object hierarchies that mirror complex, real-world entity interactions.

A natural way to represent object identities is to associate each object with a key. For example, in a classical Z specification of a library, each book on the library's shelf would be assigned a unique key. Consequently, copies of the same book would be associated with different keys. To access or modify the object, the key must be used. Section 4.7 introduced a *store* facility that provided the mapping between objects and their keys. There the store was defined as a mapping from keys, or *references*, to objects. This chapter introduces several novel techniques that ease the burden of reasoning about refinement calculus programs that use a store.

A problem that occurs with the use of a store is that two variables may contain the same reference. This is termed *aliasing.* The problem is that the modification via one variable may unintentionally alter the value of an *alias*. Given an environment containing variables *a* and *b*, modifying *a* may alter *b*, depending upon whether *a* and *b* are aliased. The problems with aliasing can be categorised as follows.

- A seemingly innocuous predicate may become unsatisfiable in the presence of aliasing. For example, establishing $a = b + 1$ is trivial when *a* and *b* are not aliased, but when they are aliased, it is equivalent to establishing $a = a + 1$ or *False*. Only **magic** can establish *False*.

- A property involving one variable can be violated by the alteration of an aliased variable. For example, if *a* and *b* are aliased and $b = 0$, then modifying *a* to achieve $a = 1$ will falsify the original predicate $b = 0$.

- Reasoning about aliasing is also incomplete in that a property may already hold even though it may not appear to. For example, given $b = 0$, then one may try to modify *a* so that $a = 0$ when it already does—as *a* and *b* are aliased.

Aliasing manifests itself in a variety of means. For example, aliases may be introduced

by assigning one reference to another. Alternatively, passing the same variable twice to a procedure as a reference parameter means that there are two different variables, which are aliased. Additionally, passing a global variable to a procedure as a reference parameter also constructs an alias. Morgan [Mor88] discusses the various means by which such aliasing can be avoided.

Utting [Utt92, Utt95] has investigated the incorporation into the refinement calculus of the store approach to object identities. He showed that a linear increase in the number of reference values (variables) results in an exponential increase in the number of aliasing checks required to reason about references. For example, in an environment with three references, *a,b,c* : *Ref*, to discover the value of an expression *E* after variable modification, five different alias combinations must be considered: $\{\{a, b, c\}, \{a, bc\}, \{ab, c\}, \{ac, b\}, \{abc\}\}$. In an environment with four variables, *a,b,c,* and *d,* fifteen different alias combinations must be considered: $\{\ \{a, b, c, d\},\ \{a, bc, d\},\ \{a, bd, c\},\ \{a, b, cd\},\ \{a, bcd\},\ \{ab, c, d\}, \{ab, cd\}, \{ac, b, d\}, \{ac, bd\}, \{ad, b, c\}, \{ad, bc\}, \{abc, d\}, \{abd, c\}, \{acd, b\},\ \{abcd\}\ \}$.

Many (non-object-oriented) languages use the types of the references to curtail the exponential explosion. For example, if the types of two references are different then the references cannot be aliased. Unfortunately this does not hold for languages which allow subtyping, or the assignment of heterogeneous types. Despite this, a subtyping hierarchy can be used to remove the need for many alias checks. That is, if a reference is of a type that is not a sub- or super-type of the type of another reference, then provided subsumption is the only form of heterogeneous type assignment in the language these two references cannot be aliases.

Utting also split the store into smaller, local stores. Since references in different local stores cannot alias each other, alias checks between such references are not required. He introduced local stores that could contain heterogeneous types and a transfer operation which allowed references in one local store to be transferred to another. Finally, he showed that a program implemented using a number of local stores can be data-refined to a program which uses a single, global store. This data-refinement transforms the transfer operation to **skip**.

Other (refinement calculus) related work includes that of Bancroft [Ban97], who presents an abstract syntax, static semantics and dynamic environment, for a language containing both references and type extension. Butler [But97] has produced a technique for reasoning about trees that are implemented using pointers. Despite this work, the construction of a program that uses references still requires tedious 'array-like' reasoning. Even examples such as that by Bancroft (a linked list implementation of a sequence) which do not involve any aliasing (there is only one path to any element in the linked list) are quite complicated[1].

[1]Utting's[Utt95, p9] linked list implementation of a sequence has an alias to the tail of the linked list, paraphrased as such: $head.next^{num} = tail$ where *num* is the number of elements in the linked list and $next^{num}$

One problem with the development of programs that use references is the verbosity of the reasoning. To improve the clarity and conciseness of the reasoning several syntactic shorthands are introduced in this chapter. Targeting the syntactic representation of aliasing information, Section 6.1 introduces a variable aliasing annotation. This syntax is also designed to aid the collection and distribution of information about aliases and non-aliases. This approach differs to that of Utting who allows for the development of specifications in a context where no aliasing information is known.

Besides aliasing constraints, specifications involving references are also cluttered with predicates involving constraints on the store. To ease the burden of writing and understanding specifications involving references, Section 6.2 presents the syntax and semantics of *reference specifications*. Reference specifications incorporate annotations on the frame variables to denote certain constraints on the store.

A general proof technique is discussed in Section 6.3 which involves performing a data-refinement to permit the temporary development of a program in a simpler environment that is universal join homomorphic[2]. After development in this temporary environment, an inverse of the original homomorphism is used to transform the program back to its original environment. This technique is used in Section 6.4 to remove superfluous aliasing. This involves an innovative technique in which aliased variables are *coalesced*, allowing all but one of the aliased variables to be removed. Data-refinement via inverse commands is also used in Section 6.5 in which an original technique is introduced to transform a reference semantics program into a corresponding value semantics program and vice versa. A corollary of this technique is that programs involving references that do not require aliasing can be developed within a value semantics and automatically converted into an analogous reference semantics program. A linked list implementation, similar to that of Bancroft and Utting is then re-developed, in Section 6.6, using the techniques discussed.

6.1 Aliasing Annotations

A syntactic aliasing annotation that provides a shorthand notation for alias and non-alias information is introduced here. One way of representing the aliasing information of a program is to construct a set of possible partitions (θ) of variable names. For example, given references a through c and no known aliasing information, the set of possible alias combinations is:

$$\theta \equiv \{\ \{\{a\},\{b\},\{c\}\},\ \{\{a\},\{b,c\}\},\ \{\{a,b\},\{c\}\},\ \{\{a,c\},\{b\}\},\ \{\{a,b,c\}\}\ \}$$

If it is known that a and b are aliases, this set can be reduced to:

$$\theta \equiv \{\ \{\{a,b\},\{c\}\},\ \{\{a,b,c\}\}\ \}$$

is the dereferencing of *head* through field *next num* times.

[2] As defined in Desideratum 2.18.

This representation of aliasing information is centralised and quite easy to understand. However, in this form, the information is difficult to use. A decentralised, variable focused representation which is designed for use in program developments is introduced here. The representations can be viewed as constraining the possible sets in θ.

Definition 6.1 (Aliased) Given variable sets $\vec{x}$ and $\vec{y}$, the nomenclature $(a^{=\vec{x}}_{\neq\vec{y}})$ denotes that the variable a is aliased with all the variables in the set $\vec{x}$ and is not aliased to any variable in $\vec{y}$.

$$(a^{=\vec{x}}_{\neq\vec{y}}) \;\widehat{=}\; (\forall\, i \in \vec{x} \bullet a = i) \wedge (\forall\, j \in \vec{y} \bullet a \neq j)$$

◇

If the variables a, $\vec{x}$ and $\vec{y}$ do not partition the variable name space then the remaining variables are potential aliases.

Definition 6.2 (Vector Aliased) The $(\_^{=}_{\neq}\_)$ construct is generalised to permit vectors of variables. Given variable vector $\vec{a}$ and vectors of variable sets $\vec{X}$ and $\vec{Y}$,

$$(\vec{a}^{=\vec{X}}_{\neq\vec{Y}}) \;\widehat{=}\; \forall\, l \in 1..\#\vec{a} \bullet (\vec{a}_l{}^{=\vec{X}_l}_{\neq\vec{Y}_l})$$

◇

Example 6.3 (Aliasing Annotation) Given references a, b, and c the following specification in which a is aliased to b and not aliased to c:

$$a\mathbf{:}\left[a = b \wedge b \neq c \wedge c\uparrow \geq\ b\uparrow,\ a\uparrow \geq\ b\uparrow\right]$$

can be specified, using the aliasing annotations, as follows.

$$a\mathbf{:}\left[(a^{=b}_{\neq c}) \wedge c\uparrow \geq\ b\uparrow,\ a\uparrow \geq\ b\uparrow\right]$$

◇

Once all reference variables of the environment have been distributed across a variable's alias annotations the bottom annotation can be omitted. For example, in an environment with reference variables a, b, and c of the same store,

$$(a^{=\{b\}}_{\neq\{c\}})$$

can be abbreviated as

$$a^{=\{b\}}$$

signifying that a has an alias b and no other environment variable is aliased to a. When a variable has no aliases it is termed a *unique*[3] reference and is denoted by the following.

$$a^{=\varnothing}$$

When no aliasing information is known the annotation is usually omitted. If it is desired that this lack of information be made explicit, the following nomenclature can be used.

$$(a^{=\varnothing}_{\neq\varnothing})$$

[3]Using the nomenclature of Jones [Jon92].

6.2 Reference Specifications

This section introduces a novel statement for reasoning about references, termed *reference specifications*. Reference specifications permit several different frame annotations that denote various constraints on the store. By using reference specifications the store constraints can be dealt with in a methodical manner and the specifications made more readable. The reference specification:

$$a\!:\left[\,pre\ ,\ post\,\right]_{\star}$$

allows the reference a to be altered. It does not, however, permit the store to be altered at location a. This permits the assignment to a of another reference, e.g., a reference variable, object field, or the result of a **new** command.

The reference specification statement:

$$a\uparrow\!:\left[\,pre\ ,\ post\,\right]_{\star}$$

allows the store to be altered at the index a. It does not permit the reference value a to be altered.

The final reference specification frame annotation allows alteration of the store at the initial index a_0, and also allows alteration of the reference value a.

$$a^{*}\!:\left[\,pre\ ,\ post\,\right]_{\star}$$

It also permits the alteration of the store at index a if that location is not aliased (*uniquely* referenced). This situation occurs, for example, if a is assigned to the result of a **new** command. It does not permit the alteration of the store at index a in general as this potentially permits the alteration of the store at all indices. For example, consider the following (classical) refinement where a is assigned to some intermediate value before being assigned to its desired, final value.

$$\begin{array}{l} a\!:\left[\,a = 10\,\right] \\ \sqsubseteq \\ a := 5; \\ a := 10 \end{array}$$

In a similar manner, if it were permitted (in general) to alter the store at the final index, then a specification such as the one following could potentially alter the entire store[4] by assigning a to intermediate values and updating the store at those locations. This specification has a precondition in which a is a unique reference and a postcondition in which a is aliased at least to c. The reference a or the store at location a_0 may be altered

[4]Depending upon the rules for composition.

to establish $a\uparrow = 6$.

$$
\begin{array}{l}
a^* : \left[a^{=\varnothing} \ , \ (a^{=\{c\}}_{\neq\varnothing}) \right]_\star \\
\sqsubseteq \\
a := d; \\
a\uparrow := 7; \\
a := c
\end{array}
$$

The alteration of the store at index d was not intended by the original specification and may violate established properties. In contrast, the ability to modify the store at new, un-aliased locations cannot violate an established property.

To alter the store at the final index the frame should include the intended alias (c). Using this approach the following code can be developed:

$$
\begin{array}{l}
a^*, c\uparrow : \left[a^{=\varnothing} \ , \ (a^{=\{c\}}_{\neq\varnothing}) \wedge c\uparrow = 6 \right]_\star \\
\sqsubseteq \\
a := c; \\
c\uparrow := 6
\end{array}
$$

In the context of this example, it is considered poor practice to alter the store through the dereference of the final value of a as it is the inclusion of $c\uparrow$ in the frame that permits the alteration of the store at this location. Consequently, although technically the following code is a correct refinement, by convention it is not permitted.

$$
\begin{array}{l}
\sqsubseteq \\
a := c; \\
a\uparrow := 6
\end{array}
$$

Given the three different forms of frame annotations it is possible to formally define reference specifications.

Definition 6.4 (Reference Specification) Given disjoint variable vectors (sequences of variables) $\vec{a}$, $\vec{b}$, and $\vec{c}$ then

$$
\begin{array}{l}
\vec{a}, \vec{b}\uparrow, \vec{c}^* : \left[pre \ , \ post \right]_\star \\
\widehat{=} \\
store, \vec{a}, \vec{c} : \left[pre \ , \ post \wedge (\vec{b_0} \cup \vec{c_0}) \vdash\!\!\lhd store \right]
\end{array}
$$

where $(\vec{b_0} \cup \vec{c_0}) \vdash\!\!\lhd store$ denotes that the store remains constant except at the indices $(\vec{b_0} \cup \vec{c_0})$. The definition constrains the initial indices $\vec{b_0}$ in case the vector $\vec{b}$ overlaps with either $\vec{a}$ or $\vec{c}$.

$\diamond$

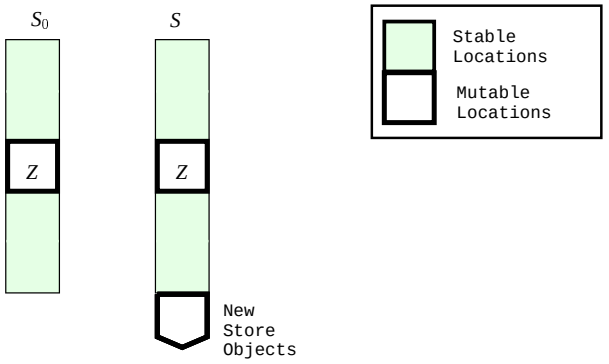

Figure 6.1: Constrained Visualisation

Definition 6.5 (Constrained) Given store S and a set of reference indices Z, then $Z|\lhd S$ is a relation denoting that the store S is the same as its initial value S_0 except possibly at the indices in Z.

$$
\begin{array}{l}
Z|\lhd S \\
\quad \widehat{=} \text{ First attempt} \\
Z \lhd S = Z \lhd S_0
\end{array}
$$

This is generalised to allow additions to store S to be unconstrained.

$$Z|\lhd S \ \widehat{=}\ (Z \lhd (\mathrm{dom}\, S_0 \lhd S) = Z \lhd S_0)$$

◇

Figure 6.1 illustrates the additional locations that the generalisation permits alterations to.

Theorem 6.6 (Reference Specification Sequential Composition) **Proof on page 182**

Many refinement rules analogous to the classical rules hold for reference specifications. For instance, a rule analogous to Sequential Composition (A.41) is permitted. For *mid* containing no initial variables except a_0, c_0 and $store_0$.

$$
\begin{array}{l}
\vec{a}, \vec{b}\uparrow, \vec{c}^{*} : \left[pre \ , \ post \right]_{\star} \\
\sqsubseteq \\
\quad \left| \left[\mathbf{con}\ STORE, \vec{A}, \vec{C} \bullet \right.\right. \\
\qquad \vec{a}, \vec{b}\uparrow, \vec{c}\uparrow : \left[pre \ , \ mid \right]_{\star} ; \\
\qquad \vec{a}, \vec{b}\uparrow, \vec{c}^{*} : \left[\dfrac{(mid)[store_0, \vec{a}_0, \vec{c}_0 \backslash STORE, \vec{A}, \vec{C}]}{(post)[store_0, \vec{a}_0 \backslash STORE, \vec{A}]} \right]_{\star} \\
\quad \left.\left| \right]\right|
\end{array}
$$

Theorem 6.7 (Invariant Aliasing) **Proof on page 184**

When no reference occurs in the frame, the aliasing constraints of the postcondition can

be weakened using those of the precondition.

$$\vec{a}\!\uparrow\!\!: \left[(\vec{a}_{\neq\vec{c}}^{=\vec{b}}) \wedge \mathit{pre} \ , \ (\vec{a}_{\neq\vec{c}}^{=\vec{b}}) \wedge \mathit{post} \right]_\star$$
$$\sqsubseteq$$
$$\vec{a}\!\uparrow\!\!: \left[(\vec{a}_{\neq\vec{c}}^{=\vec{b}}) \wedge \mathit{pre} \ , \ \mathit{post} \right]_\star$$

6.3 Data-refinement via Inverse Commands

In preparation for subsequent sections a technique discussed by Back and von Wright [BvW90] is introduced. This technique allows the development of a program through a temporary abstraction to an universal join homomorphic environment[5]. By temporarily abstracting the environment, the more mundane aspects of the program can be reasoned about using a simpler semantics/(set of rules).

The technique involves two data-refinements where the second cancels or inverts the first. Consequently, the specification and implementation variables of the first data-refinement are the implementation and specification variables of the second, respectively. In special cases, data-refinement via inverse commands maintains a form of monotonicity that allows a code segment to be developed in isolation and later incorporated in lieu of the original code. To prevent later terminology confusion, the original environment is termed the *initial* and the latter, abstracted environment is termed the *mirror environment*.

To apply this technique, a data-refinement command, *rep*, must be chosen to allow the data-refinement from the initial environment to the mirror environment. Additionally, another command, *rep′*, must be chosen to perform a data-refinement in the reverse direction: from the mirror environment to the initial environment. To ensure that the effects of *rep′* invert those of *rep*, the commands *rep* and *rep′* must be chosen so that:

$$\mathit{rep}'; \ \mathit{rep} \sqsubseteq \mathbf{skip} \tag{6.1}$$

and

$$\mathbf{skip} \sqsubseteq \mathit{rep}; \ \mathit{rep}' \tag{6.2}$$

To allow classical data-refinement rules to be applied, both *rep* and *rep′* are also constrained to be universal-join-homomorphisms. Consequently, given a program segment *P* the following development occurs:

P
$\equiv$
skip; P
$\sqsubseteq$ Equation 6.2
$\mathit{rep}; \ \mathit{rep}'; \ P$

[5]Desideratum 2.18 defines universal join homomorphisms.

Using classical data-refinement techniques, P is now data-refined to P' under rep'. That is, it is assumed that $P \preceq_{DR} P'$. Using the definition of data-refinement, this is equivalent to $rep';\ P \sqsubseteq P';\ rep'$.

$\sqsubseteq$ Definition of Data-Refinement (2.14)
$rep;\ P';\ rep'$

Since rep' maps the initial environment to a more abstract mirror environment, P' is a program segment which is more easily reasoned about than the analogous P program segment. After development of P', a refinement Q', i.e., $P' \sqsubseteq Q'$, is obtained.

$\sqsubseteq$ Monotonicity of refinement.
$rep;\ Q';\ rep'$

Finally, Q' is transformed back to the initial environment by data-refining it under rep to obtain Q'' (hence $Q' \preceq_{DR} Q''$).

$\sqsubseteq$ Definition of Data-Refinement (2.14)
$Q'';\ rep;\ rep'$
$\sqsubseteq$ Equation 6.1
$Q'';$ **skip**
$\equiv$
Q''

In summary, $P \sqsubseteq Q''$ if $P \preceq_{DR} P'$ under rep', $P' \sqsubseteq Q'$ and $Q' \preceq_{DR} Q''$ under rep for universal join homomorphisms rep and rep' where rep' is the inverse of rep. Examples of the use of this technique are provided in Sections 6.4 and 6.5.

6.4 Coalesced Programs

As identified earlier, one form of aliasing difficulty occurs when the alteration of a variable violates a property for an alias. A more detailed example of how a property may be violated through aliasing, and naive reasoning, is now presented. This example is used to motivate and describe a novel technique for avoiding such violations. Consider the following refinement, starting with a reference specification which permits the modification of the store at indices a and b to establish the postcondition under the assumption that a and b are aliased.

$a\uparrow, b\uparrow: \left[a^{=b}\ ,\ a\uparrow \leq 9 \wedge b\uparrow \geq 4 \right]_\star$
$\sqsubseteq$ Reference Specification Sequential Composition (6.6)
$a\uparrow, b\uparrow: \left[a^{=b}\ ,\ a^{=b} \wedge b\uparrow \geq 4 \right]_\star;\ a\uparrow, b\uparrow: \left[a^{=b} \wedge b\uparrow \geq 4\ ,\ a\uparrow \leq 9 \wedge b\uparrow \geq 4 \right]_\star$
$\sqsubseteq$ Invariant Aliasing (6.7)
$a\uparrow, b\uparrow: \left[a^{=b}\ ,\ b\uparrow \geq 4 \right]_\star;\ a\uparrow, b\uparrow: \left[a^{=b} \wedge b\uparrow \geq 4\ ,\ a\uparrow \leq 9 \wedge b\uparrow \geq 4 \right]_\star$

According to classical techniques, it would now seem desirable to use a strengthen postcondition rule similar to Morgan's to remove $b\uparrow \geq 4$ from the second postcondition given

that it is already satisfied by the precondition. Such a rule does not exist. If one did, then the following would be a refinement (after a further weaken precondition).

$$\not\sqsubseteq a\uparrow, b\uparrow: \left[a^{=b} \ , \ b\uparrow \geq 4 \right]_\star ; \ a\uparrow, b\uparrow: \left[a^{=b} \ , \ a\uparrow \leq 9 \right]_\star$$

Using the knowledge of the alias of a with b, this refines to:

$$\sqsubseteq a\uparrow, b\uparrow: \left[a^{=b} \ , \ a\uparrow \geq 4 \right]_\star ; \ a\uparrow, b\uparrow: \left[a^{=b} \ , \ a\uparrow \leq 9 \right]_\star$$

This would then refine to:

$$\sqsubseteq a\uparrow := 10; \ a\uparrow := 3$$

This program violates the conjunct $b\uparrow \geq 4$ of the original postcondition. If however, the original specification were given as:

$$a\uparrow: \left[a^{=b} \ , \ a\uparrow \leq 9 \wedge a\uparrow \geq 4 \right]_\star$$

then the strengthen postcondition rule could not be applied to weaken the postcondition from $a\uparrow \leq 9 \wedge a\uparrow \geq 4$ to $a\uparrow \leq 9$. The systematic replacement of all occurrences of b with its alias a is a solution which would avoid the violation of properties due to aliasing. This observation motivates an original technique termed *coalescing*. Coalescing is the unification of references, in this case a and b. The technique is an instantiation of the data-refinement technique described in Section 6.3. Consequently it involves three steps starting with the transformation of the program to an environment where the two references are coalesced, leaving only one reference in the environment. The program is then refined and finally the resulting code is transformed back to the original environment in which the second reference exists. The final transformation typically requires modifications such as the inclusion of additional assignment statements. Both transformation processes, though, are syntactic, trivial and require no additional proof.

6.4.1 Transforming to a Coalesced Program

Transforming to a coalesced environment involves data-refining a program segment so that two or more previously separate reference variables are unified and hence accessible via a single variable. The following rules are used for coalescing specifications.

Although the coalescing of a specification is a data-refinement, the operator 'coalesces-to' is used here to indicate that unlike other data-refinements, the development of the (sub)program is not complete until the latter, third, step of transforming back to the original environment is performed.

For all rules in this section, the variables $\vec{b}$ are coalesced to the variable a. The variable a is termed the *primary*. Coalescing $\vec{b}$ with a requires the replacement of $\vec{b}$ and $\vec{b_0}$ with a and a_0 respectively.

Theorem 6.8 (Unannotated Coalesced Specification) **Proof on page 185**
For variable set E, disjoint from a and $\vec{b}$:

$$a, \vec{b}, E: \left[pre \wedge a^{=\vec{b}} \ , \ post \right]_\star$$

coalesces-to

$$a, E: \left[pre[\vec{b}\backslash a] \wedge a^{=\varnothing} \ , \ post[\vec{b}, \vec{b_0}\backslash a, a_0] \right]_\star$$

The only restriction on the frame annotations of a and b is that they are the same. Thus there are two other similar rules:

Postulate 6.9 (Dereferenced Coalesced Specification)

$$a\!\uparrow, \vec{b}\!\uparrow, E: \left[pre \wedge a^{=\vec{b}} \ , \ post \right]_\star$$

coalesces-to

$$a\!\uparrow, E: \left[pre[\vec{b}\backslash a] \wedge a^{=\varnothing} \ , \ post[\vec{b}, \vec{b_0}\backslash a, a_0] \right]_\star$$

Example 6.10 (Transforming to a Coalesced Specification) For example, the reference specification

$$a\!\uparrow, b\!\uparrow: \left[a^{=b} \ , \ a\!\uparrow \leq 9 \wedge b\!\uparrow \geq 4 \right]_\star$$

coalesces-to

$$a: \left[a^{=\varnothing} \ , \ a\!\uparrow \leq 9 \wedge a\!\uparrow \geq 4 \right]_\star$$

◇

Postulate 6.11 (Starred Coalesced Specification)

$$a^*, \vec{b}^*, E: \left[pre \wedge a^{=\vec{b}} \ , \ post \right]_\star$$

coalesces-to

$$a^*, E: \left[pre[\vec{b}\backslash a] \wedge a^{=\varnothing} \ , \ post[\vec{b}, \vec{b_0}\backslash a, a_0] \right]_\star$$

6.4.2 Transforming from a Coalesced Program

After refinement of the coalesced program obtained using the data-refinement rules of the previous section, the resulting program must be transformed back into the environment containing the coalesced variable. The transformation is a piecewise process, whereby each construct of the program is transformed using the rule appropriate for that language construct. The rules are syntactic, the process is straightforward and no additional proof is required. The operator 'uncoalesces-to' is used here as a complement to the 'coalesces-to' operator.

The rules for specifications and assignments are given. For most other language constructs the transformation is an identity function and hence leaves the program syntactically the same. For instance, for assignments to variables other than the primary, the transformation is the identity function. For assignments to the primary an additional assignment must be inserted to re-synchronise the coalesced variables to the primary.

Theorem 6.12 (Assignments to the Primary) **Proof on page 187**

For assignments to primary variable a the following transformation is applicable.

$$a := X;$$

uncoalesces-to

$$a := X;$$
$$\vec{b} := a, ..., a$$

Dereferenced assignments to the primary, e.g., $a\uparrow := X$, remain the same as they actually alter the store, not the reference (a).

For specifications, the precondition can be optionally strengthened with the aliasing of a to $\vec{b}$, i.e., $a^{=\vec{b}}$. The postcondition must be strengthened with $a^{=\vec{b}}$ as the code following the specification relies on the aliasing. There are three rules provided for the transformation from a coalesced to an uncoalesced specification; one for each frame element annotation: none, $\uparrow$ and *.

Theorem 6.13 (Unannotated Uncoalesced Specification) **Proof on page 187**

When uncoalescing a specification, the coalesced variables ($\vec{b}$) are returned to the frame and the postcondition is strengthened to ensure that subsequent code can rely on the aliasing of a with $\vec{b}$. For variable set E not containing a:

$$a, E\mathbf{:}\ \left[\ pre\ ,\ post\ \right]_\star$$

uncoalesces-to

$$a, \vec{b}, E\mathbf{:}\ \left[\ a^{=\vec{b}} \wedge pre\ ,\ a^{=\vec{b}} \wedge post\ \right]_\star$$

Theorem 6.14 (Dereferenced Uncoalesced Specification) **Proof on page 188**

Alternatively, for specifications involving a frame with a dereferenced primary:

$$a\uparrow, E\mathbf{:}\ \left[\ pre\ ,\ post\ \right]_\star$$

uncoalesces-to

$$a\uparrow, \vec{b}\uparrow, E\mathbf{:}\ \left[\ a^{=\vec{b}} \wedge pre\ ,\ post\ \right]_\star$$

The strengthening of the postcondition is not required as the frame annotation of a prevents the reference a from being altered.

Postulate 6.15 (Starred Uncoalesced Specification)

$$a^*, E\mathbf{:}\ \left[\ pre\ ,\ post\ \right]_\star$$

uncoalesces-to

$$a^*, \vec{b}^*, E\mathbf{:}\ \left[\ a^{=\vec{b}} \wedge pre\ ,\ a^{=\vec{b}} \wedge post\ \right]_\star$$

Example 6.16 (Coalescing) Starting with a reference specification, the coalescing of a and b is used here to refine in the absence of aliasing.

$$a, b: \left[a^{=b} \wedge c\uparrow = 5 \ , \ a\uparrow \leq 9 \wedge b\uparrow \geq 4 \right]_\star$$

coalesces-to

$$a: \left[c\uparrow = 5 \wedge a^{=\varnothing} \ , \ a\uparrow \leq 9 \wedge a\uparrow \geq 4 \right]_\star$$

This can be refined to the following assignment which aliases c to a:

$$\sqsubseteq$$
$$a := c$$

After transformation back to the original environment, the following code results:

$$a := c;$$
$$b := a$$

In comparison, this code would alternatively have had to have been produced via the following (abbreviated) development:

$$a, b: \left[a^{=b} \wedge c\uparrow = 5 \ , \ a\uparrow \leq 9 \wedge b\uparrow \geq 4 \right]_\star$$

As an aside, notice that the original specification does not require the aliasing of a with b in the postcondition. This constraint was introduced by coalescing the specification.

$\sqsubseteq$ Reference Specification Sequential Composition (6.6)
$$a, b: \left[a^{=b} \wedge c\uparrow = 5 \ , \ a\uparrow \leq 9 \wedge a\uparrow \geq 4 \right]_\star ;$$
$$a, b: \left[a\uparrow \leq 9 \wedge a\uparrow \geq 4 \ , \ a\uparrow \leq 9 \wedge b\uparrow \geq 4 \right]_\star$$

$\sqsubseteq$ Introduce Reference Assignment
$$a := c;$$
$$b := a$$

$\diamond$

6.4.3 Soundness of Coalesced Programming

The development of programs using coalescing temporarily transfers the program to a different environment using the data-refinement by inverse commands technique introduced in Section 6.3. The initial environment variables are the variables that are coalesced, i.e., $\vec{b}$. These are removed and no variables are introduced. That is, there are no mirror environment variables. Using the abstraction invariant

$$AI \ \widehat{=} \ \vec{b} = a$$

the data-refinement command for transforming to a coalesced environment is:

$$rep\ p \ \widehat{=} \ (\exists \vec{b} \bullet p \wedge \vec{b} = a)$$

The data-refinement command for transforming from a coalesced environment is:

$$rep'\ p \ \widehat{=} \ p \wedge \vec{b} = a$$

Theorem 6.17 (Inversed reps) **Proof on page 190**

Showing that coalesced programming is an instantiation of the 'data-refinement via inverse commands' technique requires proving that *rep′* is an inverse of *rep* as shown in Theorem 6.17. That is,

$$rep'; \ rep \equiv \{\vec{b} = a\}$$

and hence, as a corollary:

$$rep'; \ rep \sqsubseteq \textbf{skip}$$

Additionally:

$$\textbf{skip} \equiv rep; \ rep'$$

and hence, as a corollary:

$$\textbf{skip} \sqsubseteq rep; \ rep'$$

The data-refinement rules provided in Sections 6.4.1 and 6.4.2 use *rep* and *rep′*. For the transformation to coalescent programs it is intended that only specifications are to be coalesced[6]. For this purpose, rules 6.8, 6.9 and 6.11, have been provided. Rules must also be provided for all possible language constructs for the transformation back to the initial environment. For assignments and specifications, alterations are typically required, and hence rules 6.12, 6.13, 6.14 and 6.15 are provided. Most constructs, however, are transformed under the identity function; and as such most of these rules are omitted. One such rule, 6.18, is included for instructional purposes. Assignments to variables other than the primary variable are transformed to themselves when they are uncoalesced.

Postulate 6.18 (Assignments to Non-Primaries)
Given the variable *c*, which is not the primary variable (i.e., *a*),

$$c := X$$

uncoalesces-to

$$c := X$$

6.5 Semantics Conversion

This section presents an innovative technique termed *semantics conversion*. Semantics conversion involves temporarily data-refining reference variables to value variables. This

[6]There is no technical reason why other language constructs could not be coalesced.

allows simple reasoning using a semantics for values rather than the complex, array-like reasoning needed for a semantics for references.

A reference semantics code segment is 'abstracted' to a value semantics program by replacing the variable dereferences with analogous value variables. The program is then refined using classical refinement rules (e.g., those of Morgan). The resulting program is then transformed back to a semantics for references. Like coalescing, the soundness of semantics conversion is achieved using the proof technique described in Section 6.3. Section 6.5.1 provides the data-refinement rules for transforming to a value semantics, Section 6.5.2 provides the data-refinement rules for transforming to a reference semantics, and Section 6.5.3 presents the soundness proof.

6.5.1 Transforming to a Value Semantics

This section presents rules for the transformation of reference specifications to value semantics specifications. The rules assume that the variables $\vec{a}$ and $\vec{b}$ are being converted from reference to value semantics variables.

Theorem 6.19 (Transforming Reference Specification) **Proof on page 192**
Transforming a reference specification with unaliased variables $\vec{a}$ and $\vec{b}$ involves replacing all dereferences of $\vec{a}$ and $\vec{b}$ with direct variable accesses.

$$\vec{a}\uparrow\!\!: \left[a^{=\varnothing} \wedge b^{=\varnothing} \wedge pre[\vec{a}, \vec{b} \backslash \vec{a}\uparrow, \vec{b}\uparrow] \ , \ post[\vec{a}, \vec{a}_0, \vec{b}, \vec{b}_0 \backslash \vec{a}\uparrow, \vec{a}\uparrow_0, \vec{b}\uparrow, \vec{b}\uparrow_0] \right]_\star$$

encodes-as

$$\vec{a}\!\!: \left[pre \ , \ post \right]_\star$$

The dereferences of $\vec{a}$ and $\vec{b}$ are replaced with direct variable accesses.

The operator 'encodes-as' is used to indicate that semantics conversion is a two-phase process. The result of the conversion is a reference specification as $\vec{a}$ and $\vec{b}$ may not partition all reference variables in the environment. If they do, then the reference specification statement can be simplified to a classical specification statement. The proof involves the implicit removal of reference variables $\vec{a}$ and the implicit addition of value variables $\vec{a}$. Only the dereferences must be replaced as the variables are unaliased and hence they do not otherwise occur free in either *pre* or *post*.

Example 6.20 (Transforming Reference Specification) The variables in the reference specification

$$a\uparrow\!\!: \left[a^{=\varnothing} \wedge b^{=\varnothing} \wedge a\uparrow \!= b\uparrow \ , \ a\uparrow \!= b\uparrow + 1 \right]_\star$$

has no aliases. By converting the reference a to a value variable, the specification can be transformed to:

$$a\!\!: \left[a = b\uparrow \ , \ a = b\uparrow + 1 \right]_\star$$

By converting both references a and b to value variables, the specification can be transformed to:

$$a\!:\left[\, a = b \;,\; a = b + 1 \,\right]$$

The refinement of this specification is significantly easier. For example, using Simple Specification (A.52) it refines to

$$a := a + 1$$

This code, once transformed back to the reference semantics, is a refinement of the original reference specification.
◇

6.5.2 Transforming to a Reference Semantics

This section provides the rules for transforming from a value semantics to a reference semantics. Although all language constructs must be provided with a transformation rule for the technique to be complete, only several essential rules are presented here. These rules assume that the variables $\vec{z}$ are being converted from value to reference semantics variables.

Theorem 6.21 (Transforming Value Semantics Assignments) **Proof on page 200**
For the transformation of value semantics variables $\vec{z}$ to reference semantics variables, the assignment

$$D := E$$

decodes-as

$$(D := E)[\vec{z}\backslash\vec{z}\uparrow]$$

The operator 'decodes-as' is used to indicate rules for the second phase of the semantics conversion process.

Example 6.22 (Transforming Value Semantics Assignments) For example, the conversion of (integer) variables a and b in

$$a := b$$

will result in the reference semantics program:

$$a\uparrow := b\uparrow$$

◇

Theorem 6.23 (Transforming Value Semantics Specification) **Proof on page 200**

Given value variables $\vec{z'}$ and $\vec{a'}$ such that $\vec{a'} \subseteq \vec{z'}$ and references $\vec{z}$ with subset $\vec{a}$, the specification

$$\vec{a'} : \left[pre \ , \ post \right]_\star$$

decodes-as

$$\vec{a}\uparrow : \left[pre[\vec{z'} \backslash \vec{z}\uparrow] \wedge z^{=\varnothing} \ , \ post[\vec{a'_0}, \vec{z'} \backslash \vec{a}_0\uparrow, \vec{z}\uparrow] \right]_\star$$

Theorem 6.24 (Transforming Value Semantics Specification B) **Proof on page 204**

This rule provides an alternative for the transformation of a value semantics specification into a reference semantics. It provides extra flexibility in the alteration of the references $\vec{a}$ yet still requires them to remain unaliased.

Given value variables $\vec{z'}$ with subset $\vec{a'}$ and references $\vec{z}$ with subset $\vec{a}$ the value semantics specification

$$\vec{a'} : \left[pre \ , \ post \right]$$

decodes-as

$$\vec{a}^* : \left[pre[\vec{z'} \backslash \vec{z}\uparrow] \wedge \vec{z}^{=\varnothing} \ , \ \vec{a}^{=\varnothing} \wedge post[\vec{a'_0}, \vec{z'} \backslash \vec{a}_0\uparrow, \vec{z}\uparrow] \right]_\star$$

6.5.3 Proof of Semantic Conversions

A proof overview is given in this section, showing that the technique and rules presented in the previous section are sound. This soundness is shown using the technique presented in Section 6.3.

The rules of Sections 6.5.1 and 6.5.2 are data-refinements using an initial environment with *store* and reference variables $\vec{z}$ and mirror environment with value variables $\vec{z'}$ and *store*′. They use the following abstraction invariant which equates the dereferences of $\vec{z}$ with the value variables $\vec{z'}$ and ensures that *store* is the same as *store*′ except at indices $\vec{z}$.

$$AI \ \widehat{=} \ \vec{z}\uparrow = \vec{z'} \wedge \vec{z} \lhd store = \vec{z} \lhd store'$$

The following data type invariant is also used to ensure that the references remain unaliased:

$$DTI \ \widehat{=} \ \vec{z}^{=\varnothing}$$

The data-refinement command for transforming to a value semantics is:

$$rep \ p \ \widehat{=} \ \exists \vec{z}, store \bullet p \wedge AI \wedge DTI$$

Similarly, the data-refinement command for transforming to a semantics for references is:

$$rep' \ p \ \widehat{=} \ \exists \vec{z'}, store' \bullet p \wedge AI \wedge DTI$$

Soundness relies on showing that the *rep*s refine to **skip** and vice versa. The following two rules fulfill this requirement.

Theorem 6.25 (Inversed reps for Semantic Conversion) **Proof on page 198**

$\textbf{skip} \sqsubseteq rep;\ rep'$

Theorem 6.26 (Inversed reps for Semantic Conversion B) **Proof on page 197**

$rep';\ rep \sqsubseteq \textbf{skip}$

6.5.4 Simulation

Since the rules presented in Section 6.5.2 are data-refinement rules they can also be used for classical data-refinement simulation as presented in Section 2.3.1. This section presents a rule for simulating a value variable block, via data-refinement, with a reference variable block.

Postulate 6.27 (Simulating a Value Semantics)
Consider the following program in which v' is a value variable. This variable block is refined to use a reference variable v instead.

$$
\begin{array}{l}
|[\ \textbf{var}\ v' : V' \bullet \\
\quad\quad M \\
]|
\end{array}
$$

The rules of Section 6.5.2 assume that a store is in the environment. Consequently, to use those rules, a store must be added to the above program.

$$
\begin{array}{l}
\sqsubseteq \text{ Introduce Local Variable Block (A.55)} \\
|[\ \textbf{var}\ v' : V, store' : Ref\ \mapsto V \bullet \\
\quad\quad N \\
]|
\end{array}
$$

For specifications in M the frame may be expanded with *store* (as is shown in the Introduce Local Variable Block (A.55) rule) to produce analogous versions in N. For simplicity, other language constructs remain the same, simulating the invariance of *store*.

The data-refinement command (rep') for the rules of Section 6.5.2 equates the dereference of v with the value variable v', allows *store* to differ from $store'$ only at indices v, and ensures that v remains unaliased: $rep' \ \widehat{=}\ (\exists\, v', store' \bullet v\uparrow\!= v' \wedge \{v\} \lhd store = \{v\} \lhd store' \wedge v^{=\varnothing}$. The result of data-refining the variable v' with this rep', using the simulation process discussed in Section 2.3.1, is:

$$
\begin{array}{l}
\sqsubseteq \\
|[\ \textbf{var}\ v : Ref\ (V), store : Ref\ (V) \mapsto V \mid v^{=\varnothing} \bullet \\
\quad\quad O \\
]|
\end{array}
$$

where $N \preceq_{DR} O$ under rep' (e.g., by transformation using the data-refinement rules provided in Section 6.5.2). This data-refinement introduces an invariant that constrains v to

be unaliased ($v^{=\varnothing}$). This constraint can be implemented by assigning v to the result of a **new**.

$$
\begin{array}{l}
\sqsubseteq \\
\quad |[\ \textbf{var}\ \mathit{store}, v := \textbf{new}\ (V) \bullet \\
\qquad\quad O \\
\quad]|
\end{array}
$$

In summary, the rules of Section 6.5.2 can be used to simulate a value variable block with a reference variable block.

6.6 Singly Linked List Example

In this section the implementation of a stack as a singly-linked list is presented. This example is based on a data-refinement presented by Bancroft [Ban97]. Two separate data-refinements are used in this example. The first transforms an object with a sequence field into an object with a (value semantics) recursively encapsulated list where each node of the list contains the tail of the list through its *next* field. For instance, the sequence:

$$\langle 1, 2, 3 \rangle$$

is represented by the 'list':

$$
\begin{array}{l}
\textbf{object} \\
\quad \mathit{element} = 1 \\
\quad \mathit{next} = \left(\begin{array}{l}
\textbf{object} \\
\qquad \mathit{element} = 2 \\
\qquad \mathit{next} = \left(\begin{array}{l}
\textbf{object} \\
\qquad\quad \mathit{element} = 3 \\
\qquad\quad \mathit{next} = \mathit{null} \\
\textbf{end}
\end{array} \right) \\
\textbf{end}
\end{array} \right) \\
\textbf{end}
\end{array}
$$

In a semantics for references *null* is a special element of the set of references *Ref*. By convention *null* is used to indicate the lack of a reference. Since the list has a value semantics here, *null* can be considered the empty object. Although the list's representation is somewhat verbose, it captures the idea that, within a semantics for values, objects are encapsulated.

After simplification, through algorithmic refinement, the resulting object is data-refined to use a reference semantics; thereby reducing the recursively encapsulated list to a classical linked list implementation. That is, rather than each object encapsulating the object that represents the list's tail, each object contains a reference to the next element in the list.

First Data-Refinement: The initial specification includes a *sequ* field containing a sequence of natural numbers, a *push* method for prepending elements to the sequence and a *pop* method for removing elements from the queue.

$$
\begin{array}{l}
\textbf{object} \\
\quad \textbf{private } sequ : \mathrm{seq}\,\mathbb{N} := \langle\rangle \\
\quad push(\textbf{value } f : \mathbb{N}) \;\widehat{=}\; sequ\textbf{::} \left[sequ = \langle f \rangle \frown sequ_0 \right] \\
\quad pop(\textbf{result } f : \mathbb{N}) \;\widehat{=}\; sequ, f\textbf{::} \left[sequ \neq \langle\rangle \;,\; sequ_0 = \langle f \rangle \frown sequ \right] \\
\textbf{end}
\end{array}
$$

This specification is data-refined to a linked list that uses recursively encapsulated node (*Node*) objects. Each node of the linked list has two fields; a *next* field holding the tail of the linked list and an *element* field containing the node's data.

$$
\begin{array}{lll}
Node \;\widehat{=}\; & \textbf{object} & \\
 & \quad next : Node & \\
 & \quad element : \mathbb{N} & \\
 & \textbf{end} &
\end{array}
$$

Given the implementation variable *head* which denotes the head node of the linked list, the data-refinement is performed under the following abstraction function.

$$af(n : Node) \;\widehat{=}\; \textbf{if } n = null \textbf{ then } \langle\rangle \textbf{ else } \langle n.element \rangle \frown af(n.next)$$

This function maps *null Nodes* to the empty sequence and proper *Nodes* to the sequence formed by concatenating the *Node*'s element with the sequence returned from the abstraction function applied to the *next* field.

Using the abstraction invariant

$$AI \;\widehat{=}\; sequ = af(head)$$

the specification is data-refined to the following object, replacing the specification variable *sequ* with the implementation variable *head*.

$$
\begin{array}{l}
\preceq_{ODR} \\
\textbf{object} \\
\quad \textbf{private } head : Node := null \\
\quad push(\textbf{value } f : \mathbb{N}) \;\widehat{=}\; \\
\quad\quad head\textbf{::} \left[af(head) = \langle f \rangle \frown af(head_0) \right] \\
\quad pop(\textbf{result } f : \mathbb{N}) \;\widehat{=}\; \\
\quad\quad \{head \neq null\} \\
\quad\quad head\textbf{::} \left[af(head_0) = \langle f \rangle \frown af(head) \right] \\
\textbf{end}
\end{array}
$$

This data-refinement can then be algorithmically refined further. For instance, the *push* operation:

$$head\textbf{::} \left[af(head) = \langle f \rangle \frown af(head_0) \right] \tag{6.3}$$

refines as follows.

Instantiating the **else** clause of the abstraction function (af) with n mapped to *head* for $head \neq null$ gives the predicate:

$$af(head) = \langle head.element \rangle \frown af(head.next)$$

The postcondition can be strengthened using this by equating f with $head.element$ and $head_0$ to $head.next$:

$$head.element = f \wedge head.next = head_0 \wedge$$
$$af(head) = \langle head.element \rangle \frown af(head.next)$$

$\equiv$ Substitutions

$$head.element = f \wedge head.next = head_0 \wedge$$
$$af(head) = \langle f \rangle \frown af(head_0)$$

$\Rrightarrow$

$$af(head) = \langle f \rangle \frown af(head_0)$$

Consequently Specification 6.3 refines to:

$\sqsubseteq$ Strengthen Postcondition (A.54)

$$head :: \left[head.element = f \wedge head.next = head_0 \right] \tag{6.4}$$

This can be further, algorithmically, refined to:

head.**update** *next* **with** *head*;
head.**update** *element* **with** f

or alternatively to the assignment:

head := **object**
next := *head*
element := f
end

Since a value semantics is in use, the assignment of *head* to the *next* field of *head* actually copies the entire linked list into the *next* field (not just a reference).

An analogous refinement may be performed for the *pop* operation thereby producing

the following, semantics for values object:

$\preceq_{ODR}$
object
  private *head* : *Node* := *null*
  push(**value** *f* : $\mathbb{N}$) $\widehat{=}$
    head := **object**
      next := *head*
      element := *f*
    end
  pop(**result** *f* : $\mathbb{N}$) $\widehat{=}$
    {*head* $\neq$ *null*}
    f := *head*.**select** *element*;
    head := *head*.**select** *next*
end

Second Data-Refinement: Alternatively, a program with a reference semantics can be developed. For the *push* operation, the following code is developed.

head := **object**
    next := *head*;
    element := *f*
  end

While this operation is syntactically the same as the value semantics program, in a semantics for references this code segment has the semantics:

$|[$ **var** *tmp* : *Ref Node* := **new** (*Node*) •
  tmp↑.**update** *next* **with** *head*;
  tmp↑.**update** *element* **with** *f*;
  head := *tmp*
$]|$

Consequently, starting at Specification 6.4, a local variable *tmp* is introduced which will temporarily hold the new head of the extended linked list.

$\sqsubseteq$ Introduce Local Variable Block (A.55)
$|[$ **var** *tmp* : *Node* •
  head, *tmp*:: $[$ *head.next* = *head*$_0$ $\wedge$ *head.element* = *f* $]$ $\lhd$
$]|$

A sequential composition is used to split this specification into two. The first is used to modify *tmp* so that it is the (temporary) head of the extended list. The second specification is used to update *head* to the list's head.

$\sqsubseteq$ Introduce Sequential Composition (A.43)
  Contract Frame (A.51)

$$
\begin{array}{l}
\left|\left[\ \textbf{con}\ \mathit{HEAD}, \mathit{TMP} \bullet \right.\right. \\
\quad \mathit{tmp}\textbf{::}\left[\ \mathit{tmp.next} = \mathit{head} \wedge \mathit{tmp.element} = f\ \right]; \qquad (6.5) \\
\quad \mathit{head}, \mathit{tmp}\textbf{::}\left[\begin{array}{ccc} \mathit{tmp.next} = \mathit{HEAD} \wedge & , & \mathit{head.next} = \mathit{HEAD} \wedge \\ \mathit{tmp.element} = f & , & \mathit{head.element} = f \end{array}\right] \qquad (6.6) \\
\left.\left]\right|\right.
\end{array}
$$

Specification 6.5 is refined further using an additional sequential composition.

$$
\begin{array}{l}
\sqsubseteq \\
\mathit{tmp}\textbf{::}\left[\ \mathit{tmp.next} = \mathit{head}_0\ \right]; \\
\mathit{tmp}\textbf{::}\left[\ \mathit{tmp.element} = f\ \right]
\end{array}
$$

Specification 6.6 is refined by strengthening the postcondition to remove references to the logical constant *HEAD*.

$$
\begin{array}{l}
\sqsubseteq \text{ Strengthen Postcondition (A.54)} \\
\quad \text{Weaken Precondition (A.53)} \\
\mathit{head}, \mathit{tmp}\textbf{::}\left[\ \mathit{head} = \mathit{tmp}_0\ \right]
\end{array}
$$

The recollected code for the *push* operation is:

$$
\begin{array}{l}
\left|\left[\ \textbf{var}\ \mathit{tmp} : \mathit{Node} \bullet \right.\right. \\
\quad \mathit{tmp}\textbf{::}\left[\ \mathit{tmp.next} = \mathit{head}_0\ \right]; \\
\quad \mathit{tmp}\textbf{::}\left[\ \mathit{tmp.element} = f\ \right]; \\
\quad \mathit{head}, \mathit{tmp}\textbf{::}\left[\ \mathit{head} = \mathit{tmp}_0\ \right] \\
\left.\left]\right|\right.
\end{array}
$$

Now *tmp* and *head* are data-refined to references using the rules provided in Section 6.5 in conjunction with rule 6.27[7]. Rule 6.27 allows a value variable block to be simulated using a reference variable block.

$$
\begin{array}{l}
\sqsubseteq \\
\left|\left[\ \textbf{var}\ \mathit{tmp} : \mathit{Ref}\ \mathit{Node} := \textbf{new}\ (\mathit{Node}) \right.\right. \\
\quad \mathit{tmp}\uparrow\textbf{:}\left[\ \mathit{tmp}^{=\varnothing}\ ,\ \mathit{tmp}\uparrow.\mathit{next}\uparrow = \mathit{head}_0\uparrow\ \right]_\star; \qquad (6.7) \\
\quad \mathit{tmp}\uparrow\textbf{:}\left[\ \mathit{tmp}^{=\varnothing}\ ,\ \mathit{tmp}\uparrow.\mathit{element} = f\ \right]_\star; \qquad (6.8) \\
\quad \mathit{head}^*, \mathit{tmp}^*\textbf{:}\left[\ \mathit{head}^{=\varnothing}\ ,\ \mathit{head}^{=\varnothing} \wedge \mathit{head}\uparrow = \mathit{tmp}\uparrow_0\ \right]_\star \qquad (6.9) \\
\left.\left]\right|\right.
\end{array}
$$

Specification 6.7 can be refined by noting that aliasing $\mathit{tmp}\uparrow$'s *next* field to head_0 ensures that their dereferences are equal.

$$
\begin{array}{l}
\sqsubseteq \text{ Strengthen Postcondition (A.54)} \\
\quad \text{Weaken Precondition (A.53)} \\
\mathit{tmp}\uparrow\textbf{:}\left[\ \mathit{tmp}\uparrow.\mathit{next} = \mathit{head}_0\ \right]_\star \\
\sqsubseteq \\
\mathit{tmp}\uparrow.\textbf{update}\ \mathit{next}\ \textbf{with}\ \mathit{head}
\end{array}
$$

[7] Omitting the necessary introduction of the store.

Specification 6.8 is refined to:

$tmp\uparrow$.**update** *element* **with** f

Now Specification 6.9 is refined in an analogous manner to Specification 6.7: by strengthening the postcondition to alias *head* with *tmp*. The constraint $head^{=\varnothing}$ forces *tmp* to be reassigned, even though it is about to exit its scope. This is caused by the data-refinement rules being too constrained.

Specification 6.9

$\sqsubseteq$ Strengthen Postcondition (A.54)
$head^*, tmp^* : \left[head^{=\varnothing} \wedge head = tmp \right]_\star$
$\sqsubseteq$
$head, tmp := tmp,$ **new** $(Node)$

After removing the unreferenced logical constants, the collected code for the *push* operation is:

$\left| \left[\right.$ **var** $tmp : Ref\ Node :=$ **new** $(Node) \bullet$
$tmp\uparrow$.**update** *next* **with** *head*;
$tmp\uparrow$.**update** *element* **with** f;
$head, tmp := tmp,$ **new** $(Node)$
$\left. \right] \right|$

With additional reasoning to reduce the final assignment to:

$head := tmp$

this could be alternatively represented by:

$head :=$ **object**
next $:= head$
element $:= f$
end

Chapter 7

Examples

This chapter presents two examples illustrating formal object-oriented development. Section 7.1 presents an example that extends a program adaptation technique for use within an object-oriented refinement calculus. Section 7.2 investigates an example that modifies an object-oriented program's class hierarchy, making it more flexible.

7.1 Object-Oriented Program Adaptation

This section presents an example in which an object-oriented program can be iteratively specified and refined. Section 7.1.1 introduces a program adaptation technique developed for the classical refinement calculus by Groves [Gro98]. This technique involves the incremental enhancement of a program using a simultaneous execution operator. This operator allows a new, enhanced specification to be given as a combination of the original program and the enhancements. The technique is then extended to an object-oriented paradigm and this extension is illustrated by example in Section 7.1.2.

7.1.1 Simultaneous Execution

Mahony [Mah99] presents the definition of a simultaneous execution operator that allows multiple statements to be executed simultaneously. The operator is monotonic and preserves code[1] for statements that modify disjoint sets of variables. Back and von Wright [BvW97, BvW99] provide an alternative presentation of the work involving a similar *product* operator. Back and Butler [BB94, BB95] also note the possibility for application of the product operator in the treatment of inheritance.

Mahony [Mah99] introduces the syntax $\mathbf{MT}_{\vec{v}\to\vec{v}}$ for monotonic predicate transformers on program variables $\vec{v}$ (in lieu of the *Ptrans* $\vec{v}$ $\vec{v}$ syntax introduced in this thesis). When only a subset ($\vec{z}$) of the program variables ($\vec{v}$) are modified, Mahony uses the syntax $\mathbf{MT}_{\vec{v}\overset{\oplus}{\to}\vec{z}}$. To allow a monotonic predicate transformer to modify additional variables (variables other than $\vec{z}$) Mahony introduces $\vec{w}$-opening. Given a monotonic predicate trans-

[1] Given executable code fragments, the composition will also be executable.

former $P : \mathbf{MT}^{\oplus}_{\vec{v}\to\vec{z}}$ the $\vec{w}$-opening allows arbitrary modification of the additional variables in $\vec{w}$.

$$P \overset{\vec{w}}{\oplus} \vec{w} \;\widehat{=}\; P;\ (\vec{w}\backslash\vec{v}\!:\!\left[\, True \,\right])$$

Mahony uses the syntax $\mathbf{A}(P)$ to denote *generalised assumption*. To prevent confusion with Back's [BvW98] use of the nomenclature *assumption*, the term *generalised assertion* is used here instead. Informally, the generalised assertion of a predicate transformer is the precondition that must be guaranteed to prevent the predicate transformer from aborting. Formally, it is defined as the weakest precondition with respect to the postcondition *True*.

$$\mathbf{A}(P) \;\widehat{=}\; P(True)$$

Some common evaluations of generalised assertions are as follows.

$$\mathbf{A}(a := x) \equiv True$$

This assumes that x is well defined.

$$\mathbf{A}(\vec{z}\!:\!\left[\, p \;,\; q \,\right]) \equiv p$$
$$\mathbf{A}(P \sqcap Q) \equiv \mathbf{A}(P) \wedge \mathbf{A}(Q)$$
$$\mathbf{A}(P \sqcup Q) \equiv \mathbf{A}(P) \vee \mathbf{A}(Q)$$
$$\mathbf{A}(P;\ Q) \equiv P(\mathbf{A}(Q))$$

The generalised effect of $P : \mathbf{MT}^{\oplus}_{\vec{v}\to\vec{z}}$ is the strongest predicate guaranteed by all conjunctive refinements. That is, it is the strongest predicate that is guaranteed by all refinements that are expressible as specification statements. For fresh $\vec{z'}$,

$$\mathbf{E}(P) \;\widehat{=}\; \neg\,(P(\neg\,(\vec{z} = \vec{z'})))[\vec{z}, \vec{z'}\backslash\vec{z_0}, \vec{z}]$$

The expression $\neg\ (P(\neg\ (\vec{z} = \vec{z'})))[\vec{z}, \vec{z'}\backslash\vec{z_0}, \vec{z}]$ may be read as 'when starting in a state denoted by $\vec{z_0}$ and executing P, it is not certain that the state $\vec{z}$ cannot be reached,' or alternatively, it is possible that state $\vec{z}$ can be reached. Two common calculations of generalised effects, for assignments and specifications, are as follows.

$$\mathbf{E}(x := e) \;\equiv\; x = (e[\vec{x}\backslash\vec{x_0}])$$
$$\mathbf{E}(\vec{z}\!:\!\left[\, p \;,\; q \,\right]) \;\equiv\; p[\vec{z}\backslash\vec{z_0}] \Rightarrow q$$

However, it is the $\vec{w}$-opened version of the generalised effects that are useful. For assignments and specifications, these are:

Theorem 7.1 (Opened Generalised Effect Basic Properties) **Proof on page 169**

$$\mathbf{E}((x := e) \overset{\vec{w}}{\oplus} \vec{w}) \equiv x = e[\vec{w}, \vec{x}\backslash\vec{w_0}, \vec{x_0}]$$
$$\mathbf{E}(\vec{z}\!:\!\left[\, p \;,\; q \,\right] \overset{\vec{w}}{\oplus} \vec{w}) \equiv p[\vec{z}, \vec{w}\backslash\vec{z_0}, \vec{w_0}] \Rightarrow q[\vec{w}\backslash\vec{w_0}]$$

Law 7.2 (Least Conjunctive Refinement) Using $\mathbf{A}(\_)$ and $\mathbf{E}(\_)$ the least conjunctive refinement of a statement P, denoted by $\Box P$, can be determined. For $P : \mathbf{MT}_{\vec{v}\to\vec{z}}^{\oplus}$ [Mah99, Corollary 3.14]:

$$\Box P \equiv \vec{z}\!:\! \left[\, \mathbf{A}(P) \ , \ \mathbf{E}(P) \,\right]$$

If P is conjunctive then it is its own least conjunctive refinement. Consequently, for conjunctive P:

$$P \equiv \vec{z}\!:\! \left[\, \mathbf{A}(P) \ , \ \mathbf{E}(P) \,\right]$$

The demonic choice of statements P and Q, that is $P \sqcap Q$, has the effect of (demonically) choosing either P or Q and 'executing' that choice. To effect the 'execution' of both P and Q, Mahony introduces *miraculous conjunction*. The *miraculous conjunction* of $P : \mathbf{MT}_{\vec{u}\to\vec{v}}^{\oplus}$ and $Q : \mathbf{MT}_{\vec{u}\to\vec{w}}^{\oplus}$, written as $P \,{}_{\vec{v}}\!\!\textcircled{\scriptsize\wedge}\!\!{}_{\vec{w}}\, Q : \mathbf{MT}_{\vec{u}\to\vec{v}\cup\vec{w}}^{\oplus}$, is defined as:

$$P \,{}_{\vec{v}}\!\!\textcircled{\scriptsize\wedge}\!\!{}_{\vec{w}}\, Q \ \widehat{=}\ \vec{v},\vec{w}\!:\! \left[\, \mathbf{A}(P) \wedge \mathbf{A}(Q) \ , \ \mathbf{E}(P \stackrel{\vec{v}}{\oplus} \vec{w}) \wedge \mathbf{E}(Q \stackrel{\vec{w}}{\oplus} \vec{v}) \,\right]$$

Unfortunately, the miraculous conjunction operator does not preserve code: it may introduce infeasible code (a miracle). For instance, the miraculous conjunction of $a := 1$ and $a := 2$ would establish $a = 1 \wedge a = 2$. If, however, the *frames* ($\vec{v}$ and $\vec{w}$) are disjoint then each variable can only be modified by one side of the miraculous conjunction and consequently code is preserved. This motivates the introduction of the simultaneous execution operator which is formed by constraining the miraculous conjunction operator to disjoint state frames. That is,

$$P \,{}_{\vec{v}}|_{\vec{w}}\, Q \ \widehat{=}\ P \,{}_{\vec{v}}\!\!\textcircled{\scriptsize\wedge}\!\!{}_{\vec{w}}\, Q$$

for $\vec{v} \cap \vec{w} = \varnothing$. For example, the simultaneous execution $a := 1 \mid b := 2$ assigns to a the number 1 and to b the number 2. The frames are omitted when obvious from the context.

Groves [Gro00, Chapter 7][Gro98] presents a technique which, in some situations can reuse previous program developments when the specification of a program changes. This can be achieved by specifying the program in terms of a miraculous conjunction between the original program and a new program representing the changes. Using a collection of refinement laws, the miraculous conjunction is then distributed through the program and eventually removed. This *program evolution* often reuses many of the original development steps[2]. An example of a refinement rule that removes miraculous conjunction is Miraculous Conjunction Serialisation (7.3). This rule allows miraculous conjunction to be serialised.

Theorem 7.3 (Miraculous Conjunction Serialisation)

Groves [Gro98, p152] permits the replacement of a miraculous conjunction with a sequential composition provided that the variables modified in statement S are not accessed

[2] Although not the focus of their work, Back and von Wright [BvW97, BvW99] (especially Theorem 7) have provided some analogous rules.

in statement T.

$$\frac{u \cap sig(T) = \varnothing}{S_u \circledwedge_v T \sqsubseteq S;\ T}$$

The syntax $sig(T)$ denotes the *signature* of, or the variables referenced within, T.

Using this rule, the code $a := 1 \ {}_a|_b \ b := 2$ can be refined to the sequential composition: $a := 1;\ b := 2$. However, the code $a := 1 \ {}_a|_b \ b := a + 2$ cannot be refined to $a := 1;\ b := a + 2$ due to the presence of, as Groves terms it, *interference*. It can, however, be refined as follows.

$a := 1 \ {}_a|_b \ b := a + 2$
$\sqsubseteq$ Commutativity
$b := a + 2 \ {}_b|_a \ a := 1$
$\sqsubseteq$ Miraculous Conjunction Serialisation (7.3)
$b := a + 2;\ a := 1$

When extending simultaneous execution for use within the object-oriented refinement calculus, the constraint that the *frames* are disjoint becomes troublesome. That is, the object frames $o.a$ and $o.b$ are not disjoint as they require the modification of the same variable; for a value semantics this is o. Consequently a simultaneous execution operator which permits object frames must be defined in a similar manner to miraculous conjunction. For a value object o with disjoint fields a and b:

$$P \ {}_{o.a}|_{o.b} \ Q \ \widehat{=}\ o.a, o.b{:}{:}\left[\, \mathbf{A}(P) \wedge \mathbf{A}(Q)\ ,\ \mathbf{E}(P \oplus^{a} o.b) \wedge \mathbf{E}(Q \oplus^{b} o.a) \,\right]$$

given $P \oplus^{a} o.b \ \widehat{=}\ P;\ o.b{:}{:}\left[\, True \,\right]$ and $Q \oplus^{b} o.a \ \widehat{=}\ Q;\ o.a{:}{:}\left[\, True \,\right]$. Hence P is 'opened' so that it can modify $o.b$ and Q is 'opened' so that it can modify $o.a$.

Although both sides of the operator modify o, they are restricted to modifying disjoint parts of o. Thus the two sides do not contradict each other and infeasibility is avoided. Consequently, the preservation of code is maintained by this definition.

Similarly, for a semantics for references, both sides of the simultaneous execution are required to modify the *store* variable (at index o). Care is required, however, as aliasing may allow both sub-statements of a simultaneous execution operator to update the same store location despite the disjointness of the object frames. One solution is to ensure that the variables in the object frames are not aliased.

The simultaneous execution operator integrates with the definition of refinement (4.20) to allow code to be refined by simultaneously executing it in conjunction with a chaotic[3] statement.

[3]Morgan [Mor94] uses the nomenclature **choose** for a demonic choice statement. Back and von Wright [BvW98] use **choose** to indicate an angelic choice statement. To avoid confusion, the specification $\vec{w}{:}{:}\left[\, True \,\right]$ is used in this thesis as a statement that demonically chooses variables $\vec{w}$.

Theorem 7.4 (Introduce Chaotic Simultaneous Execution) **Proof on page 205**
Any conjunctive code can be refined by simultaneously executing it simultaneously with $\vec{w}$**:** $\left[\, True \,\right]$ for variables $\vec{w}$ of types $\vec{W}$. For conjunctive pt defined on environment $\vec{p}$,

$$pt \sqsubseteq pt\ {}_{\vec{p}}|_{\vec{w}}\ \vec{w}\textbf{:}\ \left[\, True \,\right]$$

In the classical (non-object-oriented) refinement calculus, to use the refinement $pt\ {}_{\vec{p}}|_{\vec{w}}\ \vec{w}\textbf{:}\ \left[\, True \,\right]$ in place of pt requires reducing its environment to $\vec{p}$. This can be achieved by encapsulating it within a variable block:

$$\left|\left[\ \textbf{var}\ \vec{w} : \vec{W} \bullet \right.\right.$$
$$\qquad pt\ {}_{\vec{p}}|_{\vec{w}}\ \vec{w}\textbf{:}\ \left[\, True \,\right]$$
$$\left.\left.\ \right]\right|$$

This rule could be used to refine pt to a more efficient implementation using $\vec{w}$.

Postulate 7.5 (Introduce Object Field for Legacy)
When a new field is introduced into an object, the Introduce Chaotic Simultaneous Execution (7.4) rule can be used to enhance the pre-existing methods with the ability to modify the new fields.

Given an object with fields $f_{1..i}$ and methods $m_{1..k}$, adding new fields $f_{i+1..i+j}$ (for $j \geq 0$) permits the original methods to be simultaneously executed with chaos on those fields. For new method labels $m_{k+1..k+l}$ (for $l \geq 0$), field values $fv_{1..i+j}$ of types $F_{1..i+j}$, and methods $mv_{1..k+l}$:

object
  field $f_{h\in 1..i} : F_h := fv_h$
  method $m_{h\in 1..k} = mv_h$
end
$\sqsubseteq_{\varsigma}$
object
  field $f_{h\in 1..i+j} : F_h := fv_h$
  method $m_{h\in 1..k} = mv_h\ {}_{f_{1..i}}|_{f_{i+1..i+j}}\ f_{i+1..i+j}\textbf{::}\ \left[\, True \,\right]$
  method $m_{h\in k+1..k+l} = mv_h$
end

7.1.2 Simultaneous Execution Example

The following is based on an example presented by Mikhajlova and Sekerinski [MS97] in which a banking system is enhanced with new functionality that allows all transactions of an account to be logged. Here the program adaptation techniques of Groves [Gro98] are modified to an object-oriented paradigm and used to present an alternative approach.

The main class of the banking system is the *Account* class. The *Account* class has fields for the account owner's name and the balance of the account. It also has methods

and a constructor for depositing money, withdrawing money, returning the name of the owner and for returning the balance of the account.

class *Account* **is**
  field *owner* : *Name*
  field *balance* : *Currency*
  invariant $\{\ balance \geq 0\ \}$
  method *Account*(**value** *name* : *Name*, **value** *amount* : *Currency*) $\widehat{=}$
    owner := *name* | *balance* := *amount*
  method *Deposit*(**value** *amount* : *Currency*) $\widehat{=}$
    $\{amount > 0\}$; *balance* := *balance* + *amount*
  method *Withdraw*(**value** *amount* : *Currency*) $\widehat{=}$
    $\{amount > 0 \wedge amount \leq balance\}$;
    balance := *balance* − *amount*
  method *Owner*(**result** *name* : *Name*) $\widehat{=}$
    name := *owner*
  method *Balance*(**result** *cur* : *Currency*) $\widehat{=}$
    cur := *balance*
end

The original example extended the functionality of this class with a transaction log. This included an extended interface so that the transaction log can record additional details about the transactions, such as the date it occurred. Mikhajlova and Sekerinski extended the interface by providing the class with a wrapper class which contained an extended interface. An alternative is to provide rules for adding parameters to methods. The addition of result parameters is intuitively a refinement as they can be simply ignored by the callee of the method. Value parameters can be added to a method's interface provided that they are associated with a default value. When a method is called without specifying the value parameter, the default value would be used[4]. Using this approach the *Deposit* and *Withdraw* methods could be extended with a value parameter, *when* : *Date*. The default value for the *when* parameter is omitted from the class yet is assumed to be the given constant *no_date* : *Date*.

Individual transactions are recorded within *Transaction* objects.

class *Transaction* **is**
  field *amount* : *Currency*
  field *date* : *Date*
  method *Transaction*(**value** *a* : *Currency*, **value** *d* : *Date*) =
    amount, *date* := *a*, *d*
end

A field (*transactions*) containing a sequence of *Transaction* objects is added to the *Account* class. This allows the methods to be refined to simultaneous executions that update

[4]To efficiently use this technique, a parameter naming mechanism would provide additional benefits over the positional parameter style employed here. Such a mechanism would allow more flexible control over which parameters are provided.

transactions. In this manner the program is developed incrementally though an iterative process of specification and refinement. Additionally a history constraint is used to ensure that the transaction log can only be appended to: $transactions_0$ **prefix of** *transactions*.

$\sqsubseteq_\varsigma$ Introduce Object Field for Legacy (7.5)
Conjoin Coerced Dynamic Constraint (5.13)

class *Account* **is**
field *owner* : *Name*
field *balance* : *Currency*
field *transactions* : *seq*(*Transaction*)
invariant $\{\ balance \geq 0\ \}$
dynamic $\{\ transactions_0$ **prefix of** $transactions\ \}$
method *Account*(**value** *name* : *Name*, **value** *amount* : *Currency*) =
$(\ owner := name \mid balance := amount\ ) \mid$
$transactions$**::** $[\ transactions = \langle\rangle\ ]_\star$
method *Deposit*(**value** *amount* : *Currency*, **value** *when* : *Date*) =
$(\ \{amount > 0\};\ balance := balance + amount\ ) \mid$
$transactions$**::** $\left[\begin{array}{l} \exists\, t : Transaction \bullet \\ \quad transactions = transactions_0 \frown \langle t \rangle \\ \quad t.amount = amount \wedge t.date = when \end{array}\right]_\star$
method *Withdraw*(**value** *amount* : *Currency*, **value** *when* : *Date*) =
$\left(\begin{array}{l} \{amount > 0 \wedge amount \leq balance\} \\ balance := balance - amount \end{array}\right) \mid$
$transactions$**::** $\left[\begin{array}{l} \exists\, t : Transaction \bullet \\ \quad transactions = transactions_0 \frown \langle t \rangle \\ \quad t.amount = amount \wedge t.date = when \end{array}\right]_\star$
method *Owner*(**result** *name* : *Name*) = *name* := *owner*
method *Balance*(**result** *cur* : *Currency*) = *cur* := *balance*
end

Using Update Object Method (5.6) in conjunction with the rules from Chapter 5, this

class is refined to

$$
\begin{array}{l}
\sqsubseteq_{\varsigma} \\
\textbf{class}\ \mathit{Account}\ \ \textbf{is} \\
\quad \textbf{field}\ \mathit{owner} : \mathit{Name} \\
\quad \textbf{field}\ \mathit{balance} : \mathit{Currency} \\
\quad \textbf{field}\ \mathit{transactions} : \mathit{seq}(\mathit{Transaction}) \\
\quad \textbf{invariant}\ \{\ \mathit{balance} \geq 0\ \} \\
\quad \textbf{dynamic}\ \{\ \mathit{transactions}_0\ \textbf{prefix of}\ \mathit{transactions}\ \} \\
\quad \textbf{method}\ \mathit{Account}(\textbf{value}\ \mathit{name} : \mathit{Name}, \textbf{value}\ \mathit{amount} : \mathit{Currency}) = \\
\qquad (\ \mathit{owner} := \mathit{name} \mid \mathit{balance} := \mathit{amount}\ ) \mid \mathit{transactions} := \langle\rangle \\
\quad \textbf{method}\ \mathit{Deposit}(\textbf{value}\ \mathit{amount} : \mathit{Currency}, \textbf{value}\ \mathit{when} : \mathit{Date}) = \\
\qquad (\ \{\mathit{amount} > 0\};\ \mathit{balance} := \mathit{balance} + \mathit{amount}\ ) \mid \\
\qquad \|[\ \textbf{var}\ t : \mathit{Transaction} \bullet \\
\qquad\quad t := \textbf{new}\ (\mathit{Transaction}); \\
\qquad\quad t.\textbf{call}\ \mathit{Transaction}(\mathit{amount}, \mathit{when}); \\
\qquad\quad \mathit{transactions} := \mathit{transactions} \frown \langle t \rangle \\
\qquad]\| \\
\quad \textbf{method}\ \mathit{Withdraw}(\textbf{value}\ \mathit{amount} : \mathit{Currency}, \textbf{value}\ \mathit{when} : \mathit{Date}) = \\
\qquad \left(\begin{array}{l} \{\mathit{amount} > 0 \wedge \mathit{amount} \leq \mathit{balance}\} \\ \mathit{balance} := \mathit{balance} - \mathit{amount} \end{array}\right) \mid \\
\qquad \|[\ \textbf{var}\ t : \mathit{Transaction} \bullet \\
\qquad\quad t := \textbf{new}\ (\mathit{Transaction}); \\
\qquad\quad t.\textbf{call}\ \mathit{Transaction}(\mathit{amount}, \mathit{when}); \\
\qquad\quad \mathit{transactions} := \mathit{transactions} \frown \langle t \rangle \\
\qquad]\| \\
\quad \textbf{method}\ \mathit{Owner}(\textbf{result}\ \mathit{name} : \mathit{Name}) = \mathit{name} := \mathit{owner} \\
\quad \textbf{method}\ \mathit{Balance}(\textbf{result}\ \mathit{cur} : \mathit{Currency}) = \mathit{cur} := \mathit{balance} \\
\textbf{end}
\end{array}
$$

Using appropriate definitions, the syntax for the construction of the transaction objects can be simplified, e.g.,

$$
\begin{array}{l}
t := \textbf{new}\ \mathit{Transaction}(\mathit{amount}, \mathit{when}) \\
\widehat{=} \\
t := \textbf{new}\ (\mathit{Transaction}); \\
t.\textbf{call}\ \mathit{Transaction}(\mathit{amount}, \mathit{when})
\end{array}
$$

Additionally, if *super* is given a semantics such that it refers to the immediate superclass of a subclass, then the 'original' code can be refined to calls on *super*. Finally, the simultaneous execution operators can be reduced to sequential compositions using Theorem 7.3. This step is not required, however, as the simultaneous executions can be considered

as code.

$\sqsubseteq_\varsigma$ **subclass** *Account′* **of** *Account* **is**
  method *Account*(**value** *name* : *Name*, **value** *amount* : *Currency*) =
    super.**call** *Account*(*name*, *amount*);
    transactions := ⟨⟩
  method *Deposit*(**value** *amount* : *Currency*, **value** *when* : *Date*) =
    super.**call** *Deposit*(*amount*, *from*);
    ||[**var** *t* : *Transaction* •
      t := **new** *Transaction*(*amount*, *when*);
      transactions := *transactions* ⌢ ⟨*t*⟩
    ]||
  method *Withdraw*(**value** *amount* : *Currency*, **value** *when* : *Date*) =
    super.**call** *Withdraw*(*amount*, *to*);
    ||[**var** *t* : *Transaction* •
      t := **new** *Transaction*(*amount*, *when*);
      transactions := *transactions* ⌢ ⟨*t*⟩
    ]||
end

Back and Butler [BB94] briefly discuss providing inheritance with a semantics such that the methods of a subclass are, by definition, simultaneously executed with the methods of the original (super)class. This example could use such an approach as the enhancements do not require alteration of the original class. Such an approach would elicit the following subclass where the super calls have been replaced by the syntax +.

subclass *Account′* **of** *Account* **is**
  method *Account*(**value** *name* : *Name*, **value** *amount* : *Currency*) =
    +*transactions* := ⟨⟩
  method *Deposit*(**value** *amount* : *Currency*, **value** *when* : *Date*) =
    + ||[**var** *t* : *Transaction* •
      t := **new** *Transaction*(*amount*, *when*)
      transactions := *transactions* ⌢ ⟨*t*⟩
    ]||
  method *Withdraw*(**value** *amount* : *Currency*, *to* : *Name*, *when* : *Date*) =
    + ||[**var** *t* : *Transaction* •
      t := **new** *Transaction*(*amount*, *when*)
      transactions := *transactions* ⌢ ⟨*t*⟩
    ]||
end

7.2 Rearranging Class Hierarchies

This section presents an example in which the class hierarchy is rearranged. This rearrangement is guided by an informal design pattern [GHJV96] yet results in a formal

object-refinement. Design patterns encapsulate "recurring patterns of classes and communicating objects" and they solve common design problems. The design pattern that is used for this example is *Abstract Factory*.

An informal requirement may be given that a program is to be able to handle several types of "look and feel" interfaces, e.g., window managers such as KDE and GNOME. The Abstract Factory design pattern is useful in this scenario as it guides the developer (of a program requiring different interfaces) to introduce an *Abstract Factory* class. All methods calls that are specific to a particular window manager, e.g., the instantiation of *widgets* (e.g., windows and scrollbars), are replaced by analogous method calls on the Abstract Factory class. Each window manager has an associated Concrete Factory class which is a concrete subclass of the Abstract Factory class. Each Concrete Factory class implements the methods in the manner required for its own particular window manager. Switching to a different window manager reduces to the simple task of instantiating the Abstract Factory with a different Concrete Factory.

To support this architecture, each widget must have an associated abstract class. For each window manager, corresponding concrete subclasses must be provided for each widget. The Concrete Factory class is responsible for instantiating the appropriate concrete widget classes.

In summary, the Abstract Factory design pattern is used to introduce an interface (the Abstract Factory class) for instantiating related classes (the widgets) without necessarily specifying concrete classes.

To aid readability annotated UML [JBR99, PJ99] class models are used to complement the refinement. The notation of these models is summarised in Figure 7.1. Aggregation denotes the sole containment of one object by another, e.g., a drawing may contain shapes. Access to the aggregate can only be achieved through the aggregator. Inheritance denotes class specialisation, e.g., a circle is a specialisation of a shape. Instantiation denotes a class' responsibility for creating objects of another class, e.g., a drawing package may contain a button (widget) for drawing circles. When the button is pressed, a new circle is created by the class representing the button. Acquaintance denotes the use of an object by another, e.g., a circle may be of a certain colour, however the colour is not contained solely by the circle.

The specification that will be object-refined to incorporate an Abstract Factory is assumed to be 'hard-wired' to directly use the KDE widget classes *KDEScrollbar* and *KDEWindow*. Their particular contents are not crucial to the example. The *KDEScrollbar* class has a field *value* indicating the position of the scrollbar and methods for setting (*setValue*) and returning the value (*getValue*). This field is not important and is used in

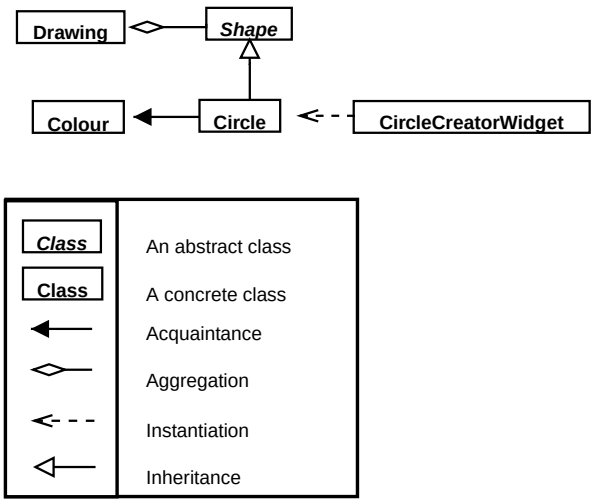

Figure 7.1: Class Diagram Notation

this example for context alone. For given constants V:

class *KDEScrollbar* **is**
private field *value* : $\mathbb{N} := V$
method *getValue*(**result** $v : \mathbb{N}$) $= v := value$
method *setValue*(**value** $v : \mathbb{N}$) $= value := v$
end

The *KDEWindow* class has methods that set the window to its maximum size (*setMaximumSize*) and to its normal size (*setNormalSize*). For given code fragments *SMS* and *SNS*:

class *KDEWindow* **is**
method *setMaximiseSize* $=$ *SMS*
method *setNormalSize* $=$ *SNS*
end

A simple client class is introduced that has fields containing sets of *KDEWindow*s (*windows*) and *KDEScrollbar*s (*scrollbars*) and a method *DoSomething* that instantiates the *KDEWindow* class, calls a method (*setNormalSize*) on the instantiation, and adds the instantiation to *windows*. Subsequently it instantiates the *KDEScrollbar* class, calls a

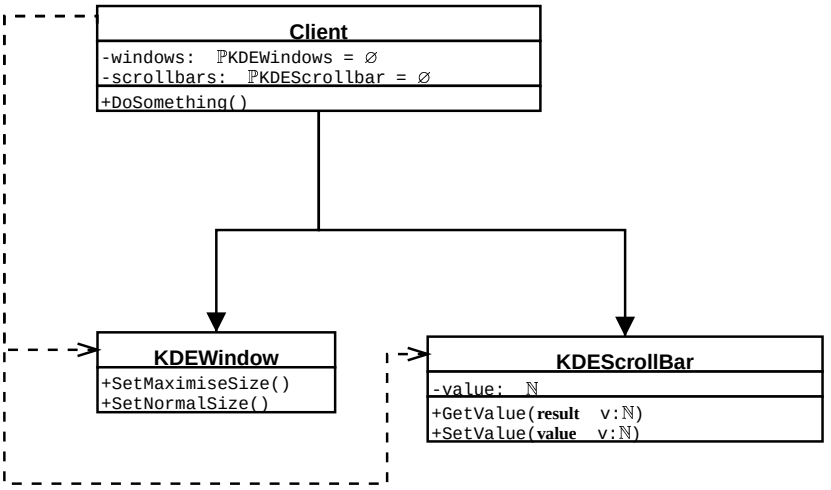

Figure 7.2: Before Abstract Factory Refinement

method (*setValue*) on the instantiation and adds it to *scrollbars*.

class *Client* **is**
  private field *windows* : ℙ *KDEWindow* := ∅
  private field *scrollbars* : ℙ *KDEScrollbar* := ∅
  method *DoSomething* =
  ||[**var** *w* : *KDEWindow* •
    w := **new** *KDEWindow*();
    w.**call** *setNormalSize*();
    windows := *windows* ∪ {*w*}
  ]||;
  ||[**var** *s* : *KDEScrollbar* •
    s := **new** *KDEScrollbar*();
    s.**call** *setValue*(0);
    scrollbars := *scrollbars* ∪ {*s*}
  ]||
end

The class hierarchy is illustrated by Figure 7.2.

By suppressing the details of the classes the program can be represented by the following code.

||[**class** *KDEScrollbar* •
  class *KDEWindow* •
  class *Client* •
    Prog
]||

The code segment *Prog* is used to instantiate the *Client* class and execute the *DoSomething*

method on that instantiation.

$$
\begin{array}{l}
Prog \;\widehat{=}\; \;|[\, \textbf{var}\ c : Client := \textbf{new}\ Client \bullet \\
\qquad\qquad c.DoSomething() \\
\qquad\quad]|
\end{array}
$$

To abstract the client and Abstract Factory from a specific window manager implementation, an abstract class must be introduced for each type of widget. This is achieved by first renaming the existing concrete widget classes. If they are not abstract enough to support the specification of widgets from a different window manager then some redevelopment is required. All development that has been performed so far can be reused for the alternative specification. If the widgets are abstract enough to support an alternative window manager then the class names are replaced according to the rule 7.6.

Postulate 7.6 (Rename Classes)
Renaming class A to B can be achieved by replacing all occurrences of A in the client (*Prog*) with B.

$$
\begin{array}{l}
|[\, \textbf{class}\ A\ \textbf{is} \\
\qquad Attribs \\
\quad \textbf{end} \bullet \\
\qquad Prog \\
]| \\
\sqsubseteq_{\varsigma} \\
|[\, \textbf{class}\ B\ \textbf{is} \\
\qquad Attribs \\
\quad \textbf{end} \bullet \\
\qquad Prog[A \backslash B] \\
]|
\end{array}
$$

Proof

The class reduces to a variable block that introduces A. This variable block data-refines to a variable block that introduces B under an abstraction invariant that equates A with B. As discussed in Section 5.2.1, the abstraction invariant $A = B$ is not monotonic with respect to object-refinement. Consequently, the abstraction invariant $(\exists\, l \bullet A \sqsupseteq_{\varsigma} l \wedge B \sqsupseteq_{\varsigma} l)$ is used instead.
QED

Using rule 7.6 to rename *KDEWindow* to *Window* and *KDEScrollbar* to *Scrollbar*

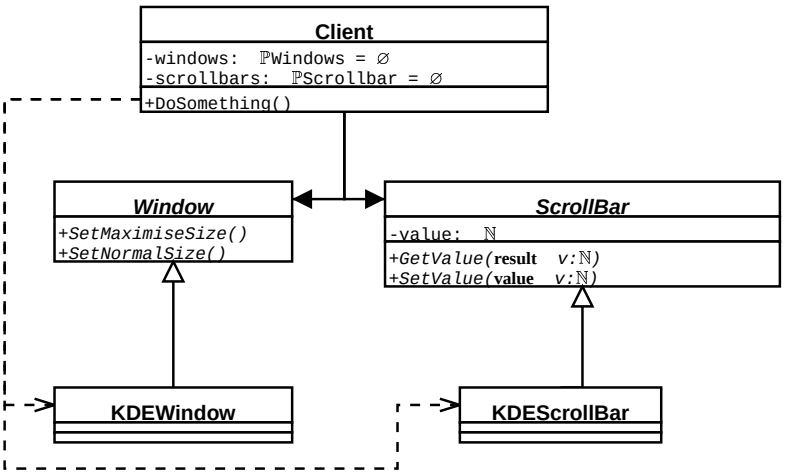

Figure 7.3: Abstract Widgets

elicits the following *Client* class:

class *Client* **is**
private field *windows* : ℙ *Window* := ∅
private field *scrollbars* : ℙ *Scrollbar* := ∅
method *DoSomething* =
[| **var** *w* : *Window* •
w := **new** *Window*;
w.**call** *setNormalSize*()
windows := *windows* ∪ {*w*}
|];
[| **var** *s* : *Scrollbar* •
s := **new** *Scrollbar*;
s.**call** *setValue*(0);
scrollbars := *scrollbars* ∪ {*s*}
|]
end

A subclass *KDEWindow* of *Window* is then introduced using the Class Introduction (5.38) rule. Typically the window widget provided by the KDE window manager will have a differing interface to *Window*. In this situation *KDEWindow* should be used as a wrapper for the actual KDE window widget class. The class hierarchy is illustrated by Figure 7.3.

The instantiation of the *Window* class in *Client* is object-refined to *KDEWindow* using Update Object Field (5.4), and similarly the *Scrollbar* class instantiation is object-refined

to an instantiation of a *KDEScrollbar* class:

```
class Client is
    private field windows : ℙ Window := ∅
    private field scrollbars : ℙ Scrollbar := ∅
    method DoSomething =
      |[ var w : Window •
           w := new KDEWindow;
           w.call setNormalSize()
           windows := windows ∪ {w}
      ]|;
      |[ var s : Scrollbar •
           s := new KDEScrollbar;
           s.call setValue(0);
           scrollbars := scrollbars ∪ {s}
      ]|
end
```

The Abstract Factory class *WidgetFactory* is introduced using Class Introduction (5.38). *WidgetFactory* has abstract methods for instantiating each abstract widget class. Consequently, the instantiations that occur in the client will eventually be refined to the appropriate calls on *WidgetFactory*. However, *WidgetFactory* will never actually be instantiated as it is used merely to constrain the behaviour of the Concrete Factory subclasses.

```
class WidgetFactory is
    method CreateWindow(result window : Window) =
        window := new Window
    method CreateScrollbar(result scrollbar : Scrollbar) =
        scrollbar := new Scrollbar
end
```

A Concrete Factory subclass, *KDEWidgetFactory*, is now introduced using Class Introduction (5.38). The result types are specialised and the instantiated classes are object-refined (Update Object Method (5.6)) to the KDE subclasses.

```
subclass KDEWidgetFactory of WidgetFactory is
    method CreateWindow(result window : MotifWindow) =
        window := new KDEWindow
    method CreateScrollbar(result scrollbar : KDEScrollbar) =
        scrollbar := new KDEScrollbar
end
```

A new *factory* field is added to *Client*, using Introduce Object Attributes (5.8), to hold the Concrete Factory. Using Result Specification (5.26) the object instantiations within

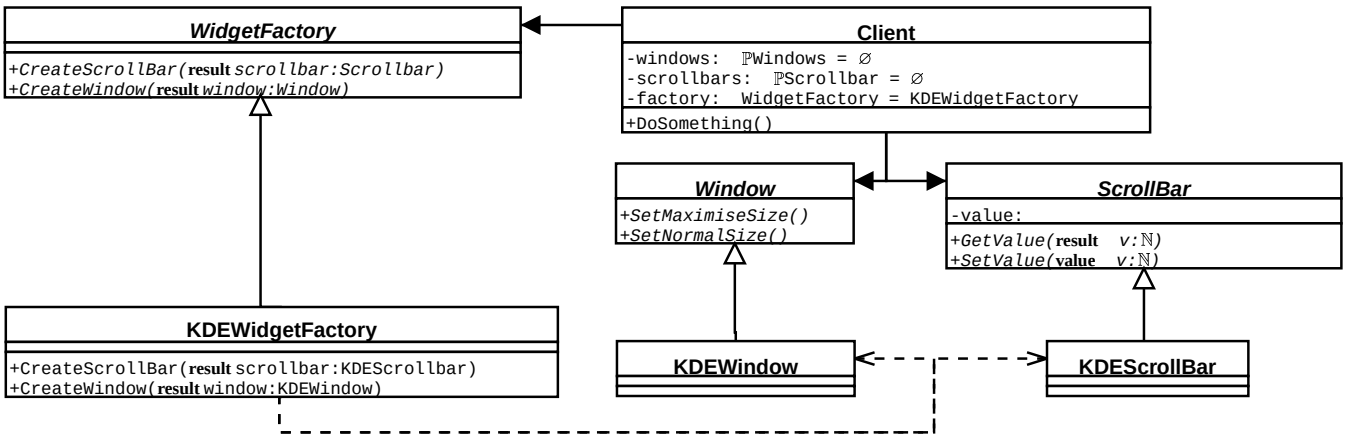

Figure 7.4: Abstract Factory

Client can be delegated to method calls on *factory*. Thus, the resulting code is:

$$
\begin{array}{l}
[\![\ \textbf{class}\ \mathit{Scrollbar}, \mathit{Window}, \mathit{WidgetFactory} \bullet \\
\quad \textbf{subclass}\ \mathit{KDEScrollbar}\ \textbf{of}\ \mathit{Scrollbar} \bullet \\
\quad \textbf{subclass}\ \mathit{KDEWindow}\ \textbf{of}\ \mathit{Window} \bullet \\
\quad \textbf{subclass}\ \mathit{KDEWidgetFactory}\ \textbf{of}\ \mathit{WidgetFactory} \bullet \\
[\![\ \textbf{class}\ \mathit{Client}\ \textbf{is} \\
\qquad \textbf{private field}\ \mathit{windows} : \mathbb{P}\,\mathit{Window} := \varnothing \\
\qquad \textbf{private field}\ \mathit{scrollbars} : \mathbb{P}\,\mathit{Scrollbar} := \varnothing \\
\qquad \textbf{private field}\ \mathit{factory} : \mathit{WidgetFactory} := \mathit{KDEWidgetFactory} \\
\qquad \textbf{method}\ \mathit{DoSomething} = \\
\qquad\quad [\![\ \textbf{var}\ w : \mathit{Window} \bullet \\
\qquad\qquad \mathit{factory}.\textbf{call}\ \mathit{createWindow}(w); \\
\qquad\qquad w.\textbf{call}\ \mathit{setNormalSize}() \\
\qquad\qquad \mathit{windows} := \mathit{windows} \cup \{w\} \\
\qquad\quad]\!]; \\
\qquad\quad [\![\ \textbf{var}\ s : \mathit{Scrollbar} \bullet \\
\qquad\qquad\quad \mathit{factory}.\textbf{call}\ \mathit{createScrollbar}(s); \\
\qquad\qquad\quad s.\textbf{call}\ \mathit{setValue}(0); \\
\qquad\qquad\quad \mathit{scrollbars} := \mathit{scrollbars} \cup \{s\} \\
\qquad\quad]\!] \\
\quad \textbf{end} \bullet \\
\qquad \mathit{Prog} \\
]\!]\ ]\!]
\end{array}
$$

Figure 7.4 presents the class diagram of this code. Now new classes can be introduced using Class Introduction (5.38) to allow for a differing "look and feel." Figure 7.5 shows the result of introducing a GNOME Concrete Factory and associated concrete widget classes. The concrete widget classes may need to act as wrappers to the widget classes provided by the GNOME window manager.

The only change required to use the GNOME interface rather than the KDE interface is to replace the assignment to *factory* of *KDEWidgetFactory* with *GNOMEWidgetFactory*

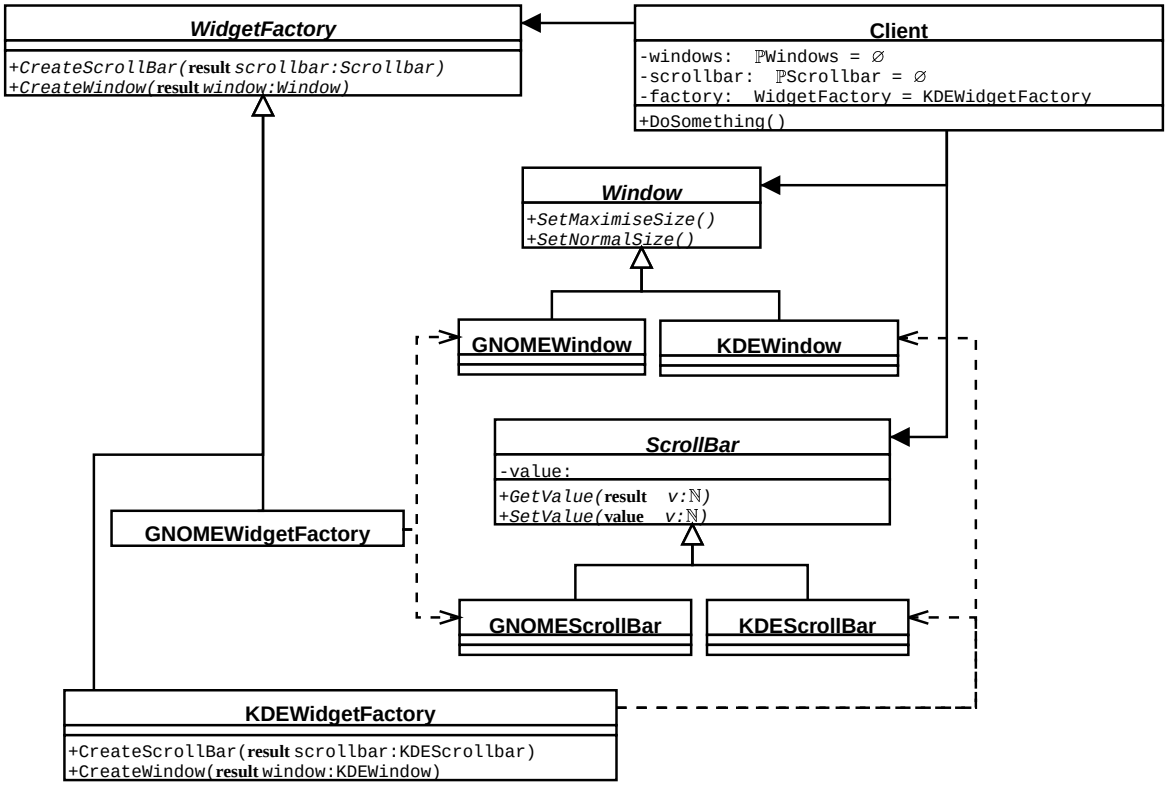

Figure 7.5: Various Looks and Feels

instead. Additional (re)development will be required if the abstract widget classes (e.g., *Window*) are not sufficiently abstract to capture the requirements of the alternative window manager.

Chapter 8

Conclusions

This thesis has covered the important issues regarding the integration of a refinement calculus with the object-oriented programming paradigm. The aim of the thesis has been the development of a scalable, calculational-style development method. Specifically, this is to be achieved through the provision of an object-oriented refinement calculus.

8.1 Summary

After an introduction to the background and literature (Chapters 1,2 and 3), the thesis provides a basis for an object-oriented refinement calculus (Chapter 4). This basis involves the unique definition of an wide-spectrum, object-based language in terms of a predicate transformer semantics which incorporates an object-calculus. Upon the predicate transformer/object-calculus basis, a predicate transformer object abstraction is developed. Constructs to define and use predicate transformer objects are defined and to address practical concerns, a semantics for references is provided. Additional constructs are defined for use with the semantics for references. Since the semantics for references is a definitional extension of the semantics for values, the resulting language could be termed extra-wide-spectrum as it allows the specification and development of both value and reference semantics programs within a single semantic framework.

Chapter 5 builds on the basis to develop an object-based refinement calculus. This is achieved through the provision of an object-refinement relation, object-data-refinement relation and refinement rules to allow the reification of specifications. A class-based refinement calculus is then provided as a definitional extension.

In Chapter 6 the reference semantics is augmented with notations and techniques to help address the extra complexity of developing programs with a semantics for references.

Finally, Chapter 7 illustrates the use of the object-oriented refinement calculus and associated techniques presented by the earlier chapters. One of the examples involves the extension of Grove's [Gro98] program adaptation technique for use within an object-oriented refinement calculus. The second example investigates the formal use of design patterns for the modification of the program's class hierarchy.

8.2 Results and Impacts

One of the main issues for object-oriented refinement calculi is object representation. The research summarised in Chapter 3 has typically provided object representations in which the methods are separated from the fields. This approach has been taken to simplify the associated object types. However, it (at least in theory) breaks the encapsulation of objects and requires an additional dynamic dispatch mechanism to be added to the semantics. The object model presented in Chapter 4 provides the alternative approach in which methods are embedded in objects. Both approaches still deserve attention as it is not yet clear which is better. For instance, the 'method embedded' model has a more succinct approach to modelling dynamic dispatch, yet has difficulties modelling multiple inheritance.

For object representations there appears to be, in general, a tradeoff between complexity and completeness. Simpler models do not provide support for the addition of new methods to subclasses and the models that do are more complex. Additionally, none of the existing models provide support for all desired object-oriented features. For instance, the subtyping of arbitrary binary methods cannot, in general, be handled. Research on object-type theory indicates that this may be an enduring problem. Judicious use of the typecase construct, as seen in Chapter 4, shows that in certain circumstances, the problem can be solved.

A heterogeneously typed framework was developed in Chapter 4. This framework included an innovative *open world* view of refinement. This definition permits refinements such as $a, b := 1, 1 \sqsubseteq a, b, c := 1, 1, 1$ where the environment of the first statement involves only a and b. Opening the refinement relation in this manner is intuitive. The assignment $a, b := 1, 1$ in an environment containing only a and b can be considered to be an assignment that also arbitrarily updates c. Since the environment does not constrain c to a type, it may be assigned, for example, an integer or even a boolean value. By allowing variables outside the environment to be modified, this definition of refinement allows programs to be more easily combined. Invariably this is a step towards a scalable refinement calculus. The 'closed' definition of refinement is part of the reason that some models have difficulty modelling the addition of new methods to subclasses. The 'open' definition is shown (§7.1) to be conducive to the incremental extension of classes through subclassing. Additionally, the 'open world' view allows classes to be replaced by subclasses using algorithmic refinement. Several of the object-oriented refinement calculi summarised in Chapter 3 have required the use of data-refinement to permit the substitution of objects. This is aesthetically displeasing as the subclass instances subsume the type of the superclass and hence no change of type is required. A data-refinement-based substitution is required, however, to allow the client to use the new attributes of the subclass.

Several of the object-oriented refinement calculi summarised have focussed on the provision of a class-based language. This thesis indicates that a better approach is to develop an object-based calculus and definitionally extend it to a class-based calculus.

This provides greater flexibility in the resulting language.

The characterising property of the object-refinement relation(s) presented in this thesis is the ability to substitute an object with an object-refinement. To achieve this monotonicity within statements, a solution inspired by Naumann [Nau94a] is to weaken predicates to those monotonic in object-refinement. Given an object variable o and constant object e, an example of a non-monotonic predicate is $o = e$. Given a state in which this predicate holds, that is, when o is equal to e, it is not possible to replace the value of o with an object-refinement and still maintain the validity of the predicate. Consequently the use of non-monotonic predicates is not conducive to practical development. Instead, a predicate such as $e \sqsubseteq_\varsigma o$ should be used. In this case, replacing o with an object refinement maintains the validity of the predicate.

It was also necessary to prove that the statements do not introduce non-monotonic predicates. This required some restrictions on the predicates used within specification statements. The proofs were originally performed by King [Kin99]. This thesis contributes to this area by proving that the constraints imposed on the specification's predicates can be relaxed. Previously, the postcondition was required to be anti-monotonic for the variables. This prevented specification statements such as

$$o{:}\left[p \sqsubseteq_\varsigma o\right]$$

for object variables o and p as the postcondition is not anti-monotonic in o. We have relaxed this constraint so that the postcondition is required to be anti-monotonic for variables not in the frame.

Fortunately the restriction to monotonic predicates has limited practical impact on the development of programs. In fact, all results of the classical refinement calculus are maintained. A similar restriction is also required in other object-oriented refinement calculi. For example, as summarised on page 29, Utting and Robinson deal with this problem in the context of their calculus.

Another contribution made by the thesis is the unique modelling of private attributes. Rather than use an ancillary mechanism to hide attributes as is done within several of the object-oriented refinement calculi presented in Chapter 3, the core object-oriented data-abstraction mechanism, subsumption, is reused. Using the *private* syntax constructs an object instance of the true (*private*) object type but stores it in a variable that is of the *public* type. Since the client is not permitted access to the private type, subsumption successfully hides the private attributes.

This thesis also furthers the understanding of invariants and history properties. History properties are retermed here as dynamic constraints. This change in nomenclature is desired as a distinction is made between those *specification* constraints that are yet to be implemented and those *implemented* constraints that are existing *properties*. This distinction is made for both invariants and for dynamic constraints. This work also highlights the need to distinguish between subclassing, which is used for the specification of differing

functionality (termed *public refinement*), and *private refinement* which is used to implement an object. The use of data-refinement can alter the reachable states of an object. Private refinement is used to restrict the client from using the additionally reachable states by forcing the client to adhere to the specification of the abstract object. The prevents the client from using the additional scope and hence possibly violating the correctness of the program. Consequently, only private data-refinement is supported. Since data-refinement is used for implementing, rather than specifying altered functionality (as subclassing is), the lack of public data-refinement is not a concern. To the authors knowledge the distinction of subclassing as a public refinement and private refinement for implementation is unique to this object-oriented refinement calculus.

For some of the object-oriented refinement calculi discussed in this thesis, object fields are introduced so that each field is modelled directly by a variable in the state. Consequently, the fields can be used directly in specification statements. For calculi (including the one presented by this thesis) that encapsulate all of an object's fields into a single entity, modifying one field requires the modification of the variable representing the entire object. Additional constraints are needed to prevent the unwanted modification of fields (see Example 3.1). The use of the object specifications introduced in this thesis have proven to be beneficial in making these constraints implicit, thereby improving the clarity of the specifications and allowing fine-grained control of the modification of attributes.

To address practical concerns, a semantics for references was introduced. This semantics involves the use of multiple stores to help curb the problem of 'alias explosion.' As references are supported through definitional extension, the semantics for references in this thesis co-exists and complements the semantics for values. An innovative technique is introduced in Chapter 6 that provides a link between the two semantics. *Semantic conversion* allows the development of a program using the simpler semantics for values. This program can then be converted to a program with a semantics for references. Additionally, a code segment in a reference semantics program can be temporarily converted to a value semantics for easier development. The conversions between the semantics are straightforward but not complete. Proof of the soundness of the technique is provided.

Another innovative technique is presented that removes the superfluous aliasing of a reference program, allowing the developer to concentrate on the program's 'true' aliasing. *Coalesced programming* allows the temporary mapping of aliases onto a single *unique* or *primary* variable. After aliasing has been removed, the semantic conversion technique can be used to allow the program (segment) to be developed using the simpler semantics for values. Like semantic conversion, the conversions to and from coalesced programs is straightforward. Also, proof of the soundness of coalesced programming is provided.

Although a multiple store approach was introduced, little attention has been paid to addressing the movement of aliases between the stores and the resulting impacts on development. Further consolidation of the multiple store approach is required.

In practice, an object-oriented refinement calculus with a semantics for references is of

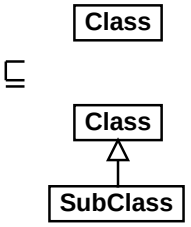

Figure 8.1: Introduce Subclass

more use due to the popularity of object-oriented languages with reference-based semantics. A calculus with an exclusive semantics for references may also be theoretically more elegant. For instance, for the semantics for values the object-refinement relation (and the object type) is required to be recursive due to the nesting of objects. In a semantics for references, the nesting of objects is flattened out into a (one-level) store. Consequently, object-refinement (and the object type) no longer need to be recursive. This observation could possible be used to substantially simplify the object representation and hence the resulting calculus. The ease of development in such a calculus would most likely be more difficult, however.

The thesis illustrated the possibilities for providing formalisations for object-oriented design patterns. Mikhajlova [Mik99b] also illustrates the formal use of the Wrapper design pattern. These examples show that it is relatively straightforward to formally use structural design patterns, though it is still unclear whether other design pattern styles, e.g., behavioural, are as amenable to formalisation.

8.3 Future Work

The example provided in Section 7.2 illustrates the viability of associating object-refinements with a graphical notation (such as UML). It would be desirable to provide a formal link between a graphical notation and the object-oriented refinement calculus, thereby producing graphical refinement rules. For example, one obvious graphical refinement rule would be the introduction of a subclass as shown in Figure 8.1. Another possible rule would be the introduction of an Abstract Factory as shown in Figure 8.2.

8.4 Thesis

In summary, this thesis has addressed the important issues of an object-oriented refinement calculus and has progressed towards the development of a scalable, calculational development method. This thesis also provides a basis for future development of formalised object-oriented refinement calculus techniques.

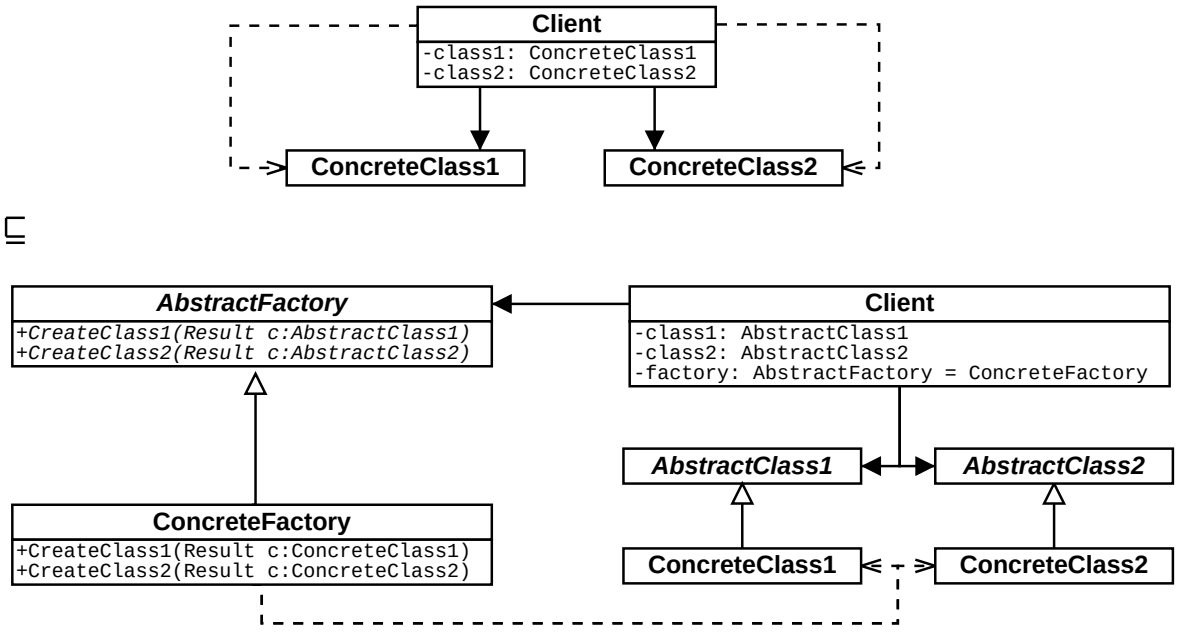

Figure 8.2: Introduce Abstract Factory

Appendix A

A Formal System of Objects

This appendix provides a reference of the additional laws, definitions and theorems used in the thesis. Sections A.1 and A.3 provide miscellaneous object calculus laws and predicate calculus laws, respectively, for use in proofs. Section A.4 presents the semantics of statements (and associated refinement rules) not covered in Chapter 4. A number of the refinement rules have been taken from Morgan's work [Mor94]. Section A.5 provides the classical data-refinement rules used in this thesis. Finally, Section A.6 presents additional object-refinement properties not listed in Chapter 4.

A.1 Object Calculus Laws

Law A.1 (Type Boolean) The boolean type is well-formed.

$$\vdash \mathbb{B}$$

Axiom A.2 (Sub Refl) If A is a well-formed type, then it is a subtype of itself. That is, subtyping is reflexive.

$$\frac{\vdash A}{\vdash A \preceq A}$$

Axiom A.3 (Sub Arrow) If B is a subtype of B' then the function type $A \to B$ which results in an object of type B can also be considered to produce a type B'. This is referred to as covariance. Since the function $A \to B$ accepts objects of type A, any subtype of A can be passed to the function. This is referred to as contravariance.

$$\frac{\vdash A' \preceq A \quad \vdash B \preceq B'}{\vdash A \to B \preceq A' \to B'}$$

A.2 Record Calculus Laws

The following rules are from the Δ_{Rcd} theory as provided by Abadi and Cardelli [AC96, Section 8.6.1,p106].

Axiom A.4 (Type Record) The record type $\{i \in 1..n \bullet (l_i \mapsto B_i)\}_{\mathbb{RT}}$ is well-formed provided each component type is well-formed.

$$\frac{\textit{for all } (i \in 1..n) \bullet \quad E \vdash B_i}{E \vdash \{i \in 1..n \bullet (l_i \mapsto B_i)\}_{\mathbb{RT}}}$$

Axiom A.5 (Sub Record) Two records are subtypes if the supertype is shorter than subtype, the old types vary covariantly, and the new types are well-formed. Given an environment where *rfun* and *rfun*′ are well-formed record types,

$$\frac{\begin{array}{c}\vdash \mathit{rfun}' \subseteq \mathit{rfun} \\ \textit{for all } (i \in \operatorname{dom} \mathit{rfun}' \bullet \vdash \mathit{rfun}(i) \preceq \mathit{rfun}'(i)) \bullet \\ \textit{for all } (i \in \operatorname{dom}(\mathit{rfun}' \lhd \mathit{rfun}) \bullet \vdash \mathit{rfun}(i)) \bullet\end{array}}{\vdash \mathit{rfun} \preceq \mathit{rfun}'}$$

Axiom A.6 (Val Record) The record $\{i \in 1..n \bullet (l_i \mapsto b_i)\}_{\mathbb{R}}$ has the type $\{i \in 1..n \bullet (l_i \mapsto B_i)\}_{\mathbb{RT}}$ provided each component of the record has the appropriate type.

$$\frac{\textit{for all } (i \in 1..n) \bullet \vdash b_i : B_i}{\vdash \{i \in 1..n \bullet (l_i \mapsto b_i)\}_{\mathbb{R}} : \{i \in 1..n \bullet (l_i \mapsto B_i)\}_{\mathbb{RT}}}$$

Axiom A.7 (Val Record Select) The type of a record selection is identified by the label within the record type of the record being selected.

$$\frac{\vdash a : \{i \in 1..n \bullet (l_i \mapsto B_i)\}_{\mathbb{RT}} \quad j \in 1..n}{\vdash a.l_j : B_j}$$

Definition A.8 (Functional Record Update) Functional record update is defined in terms of record construction and record selection.

$$\begin{array}{l} a.l_j := b \ \mathrel{\widehat{=}}\ a \oplus \{l_j \mapsto b\}_{\mathbb{R}} \\ \textit{for } a : \{i \in 1..n \bullet (l_i \mapsto B_i)\}_{\mathbb{RT}} \end{array}$$

◇

Axiom A.9 (Val Record Update) The type of a record update is the type of the record being updated augmented (or overwritten) with the type of the component that is being updated.

$$\frac{\vdash a : \{i \in 1..n \bullet (l_i \mapsto B_i)\}_{\mathbb{RD}} \quad \vdash b : B}{\vdash a.l_j := b : \{i \in 1..n \bullet (l_i \mapsto B_i)\}_{\mathbb{RD}} \oplus \{l_j \mapsto B\}_{\mathbb{RD}}} \; j \in 1..n$$

Axiom A.10 (Eq Record Update) Given two equivalent records, and two equivalent objects, updating one record with the first object is equivalent to updating the second record with the second object.

$$\frac{\vdash a =_{\{i \in 1..n \bullet (l_i \mapsto B_i)\}_{\mathbb{RD}}} a' \quad \vdash b =_B b' \;\; j \in 1..n}{a.l_j := b =_{\{i \in 1..n \bullet (l_i : B_i)\}_{\mathbb{RD}} \oplus \{l_j \mapsto B\}_{\mathbb{RD}}} a'.l := b'}$$

Axiom A.11 (Eval Record Update) Updating an a record is equivalent to constructing a new one by copying most components of the original record and replacing the one to be updated.

$$\frac{(\mathit{where}\ a \equiv \{i \in 1..n \bullet (l_i \mapsto b_i)\}_{\mathbb{R}}) \quad \vdash a : \{i \in 1..n \bullet (l_i \mapsto B_i)\}_{\mathbb{RD}} \quad \vdash b : B \;\; j \in 1..n}{a.l_j := b =_{\{l_j \mapsto B\}_{\mathbb{RD}} \cup \{i \in 1..n \setminus \{j\} \bullet (l_i : B_i)\}_{\mathbb{RD}}} a \oplus \{l_j \mapsto b\}_{\mathbb{R}}}$$

Axiom A.12 (Eq Record) Two records are equivalent if their components are equivalent.

$$\frac{\mathit{for\ all}\ (i \in 1..n) \bullet \vdash b_i =_{B_i} b'_i}{\vdash \{i \in 1..n \bullet (l_i \mapsto b_i)\}_{\mathbb{R}} =_{\{i \in 1..n \bullet (l_i \mapsto B_i)\}_{\mathbb{RD}}} \{i \in 1..n + m \bullet (l_i \mapsto b'_i)\}_{\mathbb{R}}}$$

Axiom A.13 (Eq Record Select) The results of two record selections are equivalent if the records being selected are equivalent as well as the label being selected.

$$\frac{\vdash a =_{\{i \in 1..n \bullet (l_i \mapsto B_i)\}_{\mathbb{RD}}} a' \quad j \in 1..n}{\vdash a.l_j =_{B_j} a'.l_j}$$

Axiom A.14 (Eval Record Select) A record selection is equivalent to the appropriate component from the record.

$$\frac{(\mathit{where}\ a \equiv \{i \in 1..n \bullet (l_i \mapsto b_i)\}_{\mathbb{R}}) \quad \vdash a : \{i \in 1..n \bullet l_i \mapsto B_i\}_{\mathbb{RD}} \;\; j \in 1..n}{\vdash a.l_j =_{B_j} b_j}$$

A.3 Predicate Calculus Laws

The laws in this section are basic predicate calculus laws derived from [Mor94, p258,Appendix A] and/or [Mar67].

Law A.15 (Double Implication)

$$A \Rightarrow (B \Rightarrow C) \equiv (A \wedge B) \Rightarrow C \equiv B \Rightarrow (A \Rightarrow C)$$

Law A.16 (Existential Quantification One Point Rule)

$$A[x \backslash E] \equiv (\exists x \bullet x = E \wedge A)$$

Law A.17 (Universal Quantification One Point Rule)

$$A[x \backslash E] \equiv (\forall x \bullet x = E \Rightarrow A)$$

Law A.18 (Universal de Morgan)

$$\neg\, (\forall x \bullet A) \equiv (\exists x \bullet \neg\, A)$$

Law A.19 (Universal Quantification Weak Distribution)

$$(\forall x \bullet A \Rightarrow B) \Rrightarrow (\forall x \bullet A) \Rightarrow (\forall x \bullet B)$$

The name 'weak' denotes the use of $\Rrightarrow$ rather than $\equiv$.

Law A.20 (Existential Quantification Weak Distribution)

$$(\exists x \bullet A \wedge B) \Rrightarrow (\exists x \bullet A) \wedge (\exists x \bullet B)$$

Law A.21 (Partially Superfluous Universal Quantification)

$$(\forall x \bullet N \wedge B) \equiv N \wedge (\forall x \bullet B)$$

for x not free in N.

Law A.22 (Universal Quantification Elimination) For any term E

$$\forall x \bullet A \Rrightarrow A[x \backslash E]$$

If A is true for all x, then it is true for x being equal to E.

Law A.23 (Existential Quantification Introduction)

$$A[x \backslash E] \Rrightarrow (\exists x \bullet A)$$

If A is true for x being E, then it is true for at least one x.

A.4 Statements

This section provides the typed semantics for the statements used in this thesis that are not covered in Chapter 4. The properties and refinement rules used in this thesis are also reproven for the definitions provided.

The definitions in this section are based in part on the those provided by Sekerinski [Sek96], Morgan [Mor94], and Back and von Wright [BvW98].

Skip

Definition A.24 (Skip) **skip** is the identity predicate transformer. Like **abort**, **skip** is actually a family of predicate transformers.

$$\mathbf{skip}_\alpha \mathrel{\widehat{=}} \lambda\, p : \mathit{Pred}\ \alpha \bullet p$$

The subscript is omitted when obvious from the context.
◇

Postulate A.25 (Val Skip)
Given state type α the $\mathbf{skip}_\alpha$ predicate transformer types as

$$\vdash \mathbf{skip}_\alpha : \mathit{Ptrans}\ \alpha\ \alpha$$

Assertion

Definition A.26 (Assertion) The assertion $\{p\}$ specifies a predicate that can be assumed to hold. The programmer is not responsible for establishing an assertion. Given a predicate $p : \mathit{Pred}\ \alpha$, the assertion $\{p\}$ behaves like **skip** when p is true yet like **abort** when p is false. Morgan [Mor94, p254] uses the terminology *assumption* for assertions. Unfortunately, Back [BvW98, p192] uses *assumption* for what is termed, in this thesis, a guard, or coercion. To avoid confusion, the term *assumption* has no meaning with respect to statements in this thesis. Assertions are defined on a state type β when the greatest lower bound of β and α exists.

$$\frac{\vdash p : \mathit{Pred}\ \alpha \quad E \vdash (\alpha \sqcap \beta)}{\{p\}_\beta \mathrel{\widehat{=}} \lambda\, q : \mathit{Pred}\ \beta \bullet p \wedge q}$$

◇

Postulate A.27 (Val Assertion)
The assertion $\{p\}_\beta$ is typed as follows:

$$\frac{\vdash p : \mathit{Pred}\ \alpha \quad \vdash (\alpha \sqcap \beta)}{\{p\}_\beta : \mathit{Ptrans}\ \beta\ (\alpha \sqcap \beta)}$$

The state type β is chosen as required by the context. If β is chosen to be α then the assertion types as

$$\{p\}_\alpha : Ptrans\ \alpha\ \alpha$$

and the well-formedness of $(\alpha \sqcap \alpha)$ reduces to the well-formedness of α.

Law A.28 (Introduce Assertion)

$$\left[post\right] \sqsubseteq \left[post\right] ;\ \{post\}$$

This law is drawn from [Mor94, p305,17.19].

Theorem A.29 (Weaken Assertion) **Proof on page 171**

Given predicates $p : \mathrm{pred}\, P$ and $q : \mathrm{pred}\, Q$ such that $\mathrm{pred}\, Q \preceq \mathrm{pred}\, P$ (Q has less state elements than P).

$$\frac{p \Rightarrow q}{\{p\} \sqsubseteq \{q\}}$$

Alternatively, an assertion can be removed entirely.

Law A.30 (Remove Assertion) According to Morgan [Mor94, p308], any assertion can be refined by **skip**.

$$\{p\} \sqsubseteq \mathbf{skip}$$

Guard

Definition A.31 (Guard) The guard $\left[p\right]_\beta$, also known as coercion, specifies a predicate p that the programmer is responsible for establishing. After the execution of the guard, the predicate p is true. If p were true beforehand, then the guard **skip**s, if p was not true, then the guard behaves as **magic**. If p is a predicate on state type α, then the guard $\left[p\right]_\beta$ is defined when the greatest lower bound of α and β exists.

$$\frac{\vdash p : Pred\ \alpha \quad \vdash (\alpha \sqcap \beta)}{\left[p\right]_\beta\ \widehat{=}\ \lambda q : Pred\ \beta \bullet p \Rightarrow q}$$

$\diamond$

Postulate A.32 (Val Guard)
The guard $\left[p\right]_\beta$ is typed as follows:

$$\frac{\vdash p : Pred\ \alpha \quad E \vdash (\alpha \sqcap \beta)}{\left[p\right]_\beta : Ptrans\ \beta\ (\alpha \sqcap \beta)}$$

The state type β is chosen as required by the context in an analogous manner to that for assertions.

Law A.33 (Absorb Guard) A guard following a specification can be absorbed into its postcondition.

$$w{:}\left[\, pre \;,\; post \,\right] ;\; \left[\, post' \,\right] = w{:}\left[\, pre \;,\; post \wedge post' \,\right]$$

This law is drawn from [Mor94, p298,17.2].

Update

Definition A.34 (Update) Given a state transformer $st : \alpha \to \beta$ the update operator $\langle st \rangle$ lifts it to a predicate transformer.

$$\langle st \rangle_\beta \;\widehat{=}\; \lambda\, p : Pred\ \beta \bullet (\lambda\, s : \alpha \bullet p(st\ s))$$

◇

Postulate A.35 (Val Update)
Update types as a predicate transformer from the range type of the state transformer function to its domain type.

$$\frac{\vdash st : \alpha \to \beta}{\langle st \rangle_\beta \; : Ptrans\ \beta\ \alpha}$$

The predicate transformers constructed using state transformers are deterministic. Consequently, this is a useful technique for modelling assignments.

Assignment

Definition A.36 (Assignment) Using Sekerinski's [Sek96] work as a basis, assignment can be given a typed definition as follows. Given a state α concerning certain identifiers I, the multiple assignment

$$(i_1, ..., i_m := e_1, ..., e_m)_\beta \;\widehat{=}\; \langle \lambda\, s : \alpha \bullet s \oplus \{j \in 1..m \bullet i_j \mapsto eval\ e_j\ s\}_{\mathbb{S}} \rangle_\beta$$

where $\forall\, a \in 1..m \bullet i_a \in I$ and expressions $e_1, ..., e_m$ only reference variables that are in I. Here *eval* is used to evaluate the expression e in state s.

When α and β are the same, this definition can be more intuitively defined using Morgan's definition [Mor94, p250].

$$(i_1, ..., i_m := e_1, ..., e_m) \;\widehat{=}\; (\lambda\, p \bullet p[i_1, ..., i_m \backslash e_1, ..., e_m])$$

◇

Enter

Definition A.37 (Enter) The enter predicate transformer is a deterministic (functional) state update that uses the precondition state to calculate initial values. It can be thought of intuitively as the opening clause of a local variable block. The enter statement can be provided a typed definition using Sekerinski's [Sek96] definition as a basis. For state types α where the variable being introduced $v : V$ is not in α.

$$\frac{v \; nfi \; \alpha}{enter \; v : V = initv \;\widehat{=}\; \langle \lambda s : \alpha \bullet s \cup \{v \mapsto (eval \; initv \; s)\}_{\mathbb{S}} \rangle}$$

◇

Postulate A.38 (Val Enter)
For state types 'compatible' with state type $\{v \mapsto V\}_{\mathbb{ST}}$, the enter predicate transformer types as follows:

$$\frac{v \; nfi \; \alpha}{(enter \; v : V = initv) : Ptrans \; (\alpha \sqcap \{v \mapsto V\}_{\mathbb{ST}}) \; \alpha}$$

It is known that $(\alpha \sqcap \{v \mapsto V\}_{\mathbb{ST}})$ is well-formed from $v \; nfi \; \alpha$.

Exit

Definition A.39 (Exit) The exit statment is complementary to the enter statement as it intuitively encapsulate the closing brackets of a local variable block. It is a deterministic state update that state cuts a particular state variable. It is only applicable if the variable (v of type V) is present in the state (α): $\alpha \preceq \{v \mapsto V\}_{\mathbb{ST}}$.

$$\frac{\alpha \preceq \{v \mapsto V\}_{\mathbb{ST}}}{\textbf{exit} \; v : V \;\widehat{=}\; \langle \lambda s : (\alpha \sqcap \{v \mapsto V\}_{\mathbb{ST}}) \bullet s \setminus_{\mathbb{S}} \{v\} \rangle}$$

◇

Demonic Choice

Definition A.40 (Generalised demonic choice) Demonic choice can be generalised to an arbitrary number of alternatives. For an index type I the demonic choice is defined as follows.

$$\frac{\vdash pt_i : Ptrans \; \alpha_i \; \beta_i \quad E \vdash (\sqcap \{j \in I \bullet \beta_j\})}{(\sqcap i \in I \bullet pt_i) \;\widehat{=}\; \lambda p \in Pred \; (\sqcup \{j \in I \bullet \alpha_j\}) \bullet \forall i \in I \bullet (pt_i \; p)}$$

◇

Sequential Composition

Definition A.41 (Sequential Composition) Sequential composition is functional composition of predicate transformers. Sequential composition for two predicate transformers is defined when the postcondition state type of the first is a subtype of the precondition state type of the second:

$$\frac{\vdash pt_1 : Ptrans\ \alpha\ \delta \quad \vdash pt_2 : Ptrans\ \beta\ \gamma \quad \vdash \alpha \preceq \gamma}{(pt_1;\ pt_2)\ \widehat{=}\ \lambda p : Pred\ \beta \bullet pt_1\ (pt_2\ p)}$$

◇

Postulate A.42 (Val Sequential Composition)
Sequential composition, when defined, types as a predicate transformer from the postcondition state type of 'second' predicate transformer to the precondition state type of the 'first'.

$$\frac{\vdash pt_1 : Ptrans\ \alpha\ \delta \quad \vdash pt_2 : Ptrans\ \beta\ \gamma \quad \vdash \alpha \preceq \gamma}{\vdash (pt_1;\ pt_2) : Ptrans\ \beta\ \delta}$$

Axiom A.43 (Introduce Sequential Composition) For fresh constants X,

$$\begin{array}{l} w, x{:}\left[\, pre\ ,\ post \,\right] \\ \sqsubseteq \\ \quad |[\ \mathbf{con}\ X \bullet \\ \qquad\quad x{:}\left[\, pre\ ,\ mid \,\right]\ ; \\ \qquad\quad w, x{:}\left[\, mid\ ,\ post \,\right] \\ \quad]| \end{array}$$

for formulae *mid* that do not contain initial variables other than x_0.

This rule is presented by Morgan [Mor94, Law 8.4,p310].

The following is the weakest liberal precondition [Dij76] definition for sequential compositions.

Definition A.44 (Wlp Sequential Composition) For statements P and Q and predicate r:

$$wlp.(P;\ Q).r\ \widehat{=}\ wlp.P.(wlp.Q.r)$$

◇

Local Variable Block A typed definition for the local variable block can be provided using the previously defined enter and exit statements.

Definition A.45 (Local Variable Block)

$$\left|\left[\, \textbf{var}\ a : A = av \bullet P \,\right]\right| \ \widehat{=}\ \textit{enter}\ a : A = av;\ P;\ \textbf{exit}\ a : A$$

This definition is deterministic as it is based on the deterministic enter statement, which in turn, is based on a deterministic state update. An alternative definition for a non-deterministic version is provided by Morgan [Mor94].

$$\left|\left[\, \textbf{var}\ a : A \bullet P \,\right]\right| \ \widehat{=}\ \lambda\, post \bullet (\forall\, a : A \bullet P\ post)$$

provided a is not free in *post*.
◇

Logical Constants Logical constants are magically chosen values. They are used to help refine specifications, yet they are not code and must be removed for the code to be executable.

Definition A.46 (Logical Constant) Morgan [Mor94] defines logical constants as follows.

$$\left|\left[\, \textbf{con}\ lcon \bullet pt \,\right]\right| \ \widehat{=}\ \lambda\, post \bullet (\exists\, lcon \bullet pt\ post)$$

provided *lcon* is not free in *post*.
◇

Law A.47 (Remove Logical Constant) A logical constant may be removed provided it is not used in a program.

$$\left|\left[\, \textbf{con}\ lcon \bullet pt \,\right]\right| \sqsubseteq pt$$

provided *lcon nfi pt*.

Specification

Definition A.48 (Specification Statement) The specification statement $v\!:\!\left[\, pre\ ,\ post \,\right]$ alters identifiers v so that *post* holds, given that *pre* held before the specification statement. If it is not possible for *post* to be established by modifying v (e.g., *post* were *False*) then the specification statement is equivalent to **magic**. Morgan [Mor94] defines specifications as follows.

$$v\!:\!\left[\, pre\ ,\ post \,\right] \ \widehat{=}\ \lambda\, p \bullet pre \wedge (\forall\, v \bullet post \Rightarrow p)[v_0 \backslash v]$$

The predicate (relation) *post* may contain references to the initial values of v using the subscript notation: v_0.

◇

Specifications that use zero-subscripted variables can be reduced to a specification encapsulated by a logical constant [Mor94, Abbreviation 8.2,p69].

Law A.49 (Initial Variable) For fresh X,

$$
\begin{array}{l}
w{:}\left[\, pre \;,\; post \,\right] \\
\widehat{=} \\
\quad |[\, \mathbf{con}\ X \bullet \\
\qquad w{:}\left[\, pre \wedge x = X \;,\; post[x_0 \backslash X] \,\right] \\
\quad]|
\end{array}
$$

The frame can be expanded using an expand frame rule [Mor94, Law 8.3, p69].

Law A.50 (Expand Frame)

$$w{:}\left[\, pre \;,\; post \,\right] \equiv w, x{:}\left[\, pre \;,\; post \wedge x = x_0 \,\right]$$

Law A.51 (Contract Frame) The frame can be contracted using a contract frame law [Mor94, Law 5.4].

$$w, x{:}\left[\, pre \;,\; post \,\right] \sqsubseteq w{:}\left[\, pre \;,\; post[x_0 \backslash x] \,\right]$$

Law A.52 (Simple Specification) An assignment can easily be translated to a specification statement.

$$i{:}\left[\, i = e \,\right] \equiv i := e$$

provided e contains no i [Mor94].

Law A.53 (Weaken Precondition) The precondition of a specification can be weakened.

$$\frac{pre \Rrightarrow pre'}{v{:}\left[\, pre \;,\; b \,\right] \sqsubseteq v{:}\left[\, pre' \;,\; b \,\right]}$$

This law is drawn from [Mor94, p314,1.12].

Law A.54 (Strengthen Postcondition) The postcondition of a specification can be strengthened.

$$\frac{b \Rrightarrow b'}{v{:}\left[\, b' \,\right] \sqsubseteq v{:}\left[\, b \,\right]}$$

The precondition can also be used to strengthen the postcondition [Mor94, Law 5.1].

$$\frac{pre[v \backslash v_0] \wedge b \Rrightarrow b'}{v{:}\left[\, pre \;,\; b' \,\right] \sqsubseteq v{:}\left[\, pre \;,\; b \,\right]}$$

Law A.55 (Introduce Local Variable Block) A variable not in the environment can be introduced.

If x does not occur in w, pre or $post$, then [Mor94]

$$w{:}\left[\, pre \;,\; post \,\right] \sqsubseteq |[\, \mathbf{var}\ x : T \bullet x, w{:}\left[\, pre \;,\; post \,\right] \,]|$$

A.5 Data-Refinement Laws

To data-refine guards, Gardiner and Morgan [GM91] introduce the following function.

$$\overline{rep}\ \psi \ \widehat{=}\ \neg\ (rep \neg\ \psi)$$

When *rep* is of the form

$$rep\ \psi \equiv (\exists\, a \bullet AI \wedge \psi)$$

then $\overline{rep}$ is of the form

$$\overline{rep}\ \psi \equiv (\forall\, a \bullet AI \Rightarrow \psi)$$

Using this function, guards data-refine in the following manner.

Law A.56 (Data-Refine Guard)

$$\left[\, G \,\right] \preceq_{DR} \left[\, \overline{rep}\ G \,\right]$$

A.6 Object Refinement Laws

Postulate A.57 (Object-Refinement Transitivity)
When object *a* object refines to *b*, and *b* object refines to *c*, it can be deduced that *a* object refines to *c*.

$$\frac{a \sqsubseteq_\varsigma b \quad b \sqsubseteq_\varsigma c}{a \sqsubseteq_\varsigma c}$$

Postulate A.58 (Object-Refinement Reflexive)
An object *a* is an object-refinement of itself.

$$\frac{}{a \sqsubseteq_\varsigma a}$$

Appendix B

Proofs

This appendix provides the proofs for the theorems presented in this thesis. To aid the reader, the theorems have been duplicated inside boxes at the beginning of the corresponding proofs.

B.1 State and Predicate Proofs

Proof of 4.4 from p43 (Sub State Type)

Duplicate (Sub State Type) of 4.4 on page 43.
Given an environment including state types α and β:

$$\frac{\vdash \operatorname{dom}\beta \subseteq \operatorname{dom}\alpha \quad \textit{for all } i \in \operatorname{dom}\beta \bullet \vdash \alpha(i) \preceq \beta(i) \quad \textit{for all } i \in \operatorname{dom}(\beta \lhd \alpha) \bullet \vdash \alpha(i)}{\vdash \alpha \preceq \beta}$$

This property is consistent with that used by Sekerinski [Sek96].

Given respective sub- and supertypes $\alpha, \beta \in \textit{Statetypes}$:

$\vdash \operatorname{dom}\beta \subseteq \operatorname{dom}\alpha$
$\vdash \forall i \in \operatorname{dom}\beta \bullet \alpha(i) \preceq \beta(i)$
$\vdash \forall i \in \operatorname{dom}(\beta \lhd \alpha) \bullet \alpha(i)$
$\Rrightarrow$ Sub Record (A.5)
$\vdash \alpha_{\boxed{\text{RD}}} \preceq \beta_{\boxed{\text{RD}}}$
$\Rrightarrow$ Def^n States as Records (4.5)
$\vdash \alpha_{\boxed{\text{ST}}} \preceq \beta_{\boxed{\text{ST}}}$

The antecedents are respectively that subtypes may possess more components than the supertype, the old types vary covariantly, and that the new types are well formed.

QED

Proof of 4.7 from p44 (Sub Predicate)

Duplicate (Sub Predicate) of 4.7 on page 44.
The subtyping rule for predicates is

$$\frac{\vdash \alpha \preceq \beta}{\vdash \mathit{Pred}\ \beta \preceq \mathit{Pred}\ \alpha}$$

That is, predicates subtype when the states on which they are defined vary contravariantly.

$$\frac{}{\vdash \mathbb{B}}\ \text{A.1}$$

$$\frac{\dfrac{}{\vdash \mathbb{B} \preceq \mathbb{B}}\ \text{A.2} \qquad \vdash \alpha \preceq \beta}{\beta \to \mathbb{B} \preceq \alpha \to \mathbb{B}}\ \text{A.3}$$

$$\frac{\beta \to \mathbb{B} \preceq \alpha \to \mathbb{B}}{\mathrm{pred}\,\beta \preceq \mathrm{pred}\,\alpha}\ \text{Def}^n\text{4.6}$$

QED

B.2 Predicate Transformer Proofs

Proof of 4.14 from p46 (Sub Predicate Transformers)

Duplicate (Sub Predicate Transformers) of 4.14 on page 46.
Predicate transformers subtype when the postcondition state varies covariantly and the precondition state varies contravariantly.

$$\frac{\vdash \alpha \preceq \beta \quad \vdash \gamma \preceq \delta}{\vdash \mathit{Ptrans}\ \alpha\ \delta \preceq \mathit{Ptrans}\ \beta\ \gamma}$$

as $\mathit{Pred}\ \beta \preceq \mathit{Pred}\ \alpha$ and $\mathit{Pred}\ \delta \preceq \mathit{Pred}\ \gamma$.

$$\begin{array}{ll} & \mathit{Ptrans}\ \alpha\ \delta \preceq \mathit{Ptrans}\ \beta\ \gamma \\ \equiv & \mathit{Def}^n \text{ Predicate Transformers (4.13)} \\ & \mathrm{pred}\,\alpha \to \mathrm{pred}\,\delta \preceq \mathrm{pred}\,\beta \to \mathrm{pred}\,\gamma \\ \Leftarrow & \text{Sub Arrow (A.3)} \\ & \mathrm{pred}\,\beta \preceq \mathrm{pred}\,\alpha \ \wedge\ \mathrm{pred}\,\delta \preceq \mathrm{pred}\,\gamma \\ \Leftarrow & \text{Sub Predicate (4.7)} \ \times 2 \\ & \alpha \preceq \beta \ \wedge\ \gamma \preceq \delta \end{array}$$

QED

B.3 Statement Proofs

This section provides explicit proofs of several basic properties of the simultaneous execution operator.

Postulate B.1 (Generalised Effect Freedom) **Proof on page 169**

$$w\ \mathit{nfi}\ \mathbf{E}(\mathit{at}\ \mathbin{\hat{\oplus}} w)$$

though w_0 maybe occur free.

Proof of B.1 from p168 (Generalised Effect Freedom)

Informally:

$$\mathbf{E}(at \overset{\$}{\oplus} w) \equiv \neg\ (at(a \neq a'))[aw, aw'\backslash aw_0, aw]$$

All occurrences of w in at are replaced by w_0. The only possible way to introduce w is if w' occurs free, which it does not.

QED

Theorem B.2 (Open Generalised Effect)

Given $P : MT_{\vec{u}\to\vec{v}}$ then

$$\mathbf{E}_{\vec{u}\cup\vec{w}\overset{\oplus}{\to}\vec{v}\cup\vec{w}}(P \overset{\vec{v}}{\oplus}\vec{w}) \equiv \neg\ P(\vec{v} \neq \vec{v'})[\vec{w}, \vec{w'}, \vec{v}, \vec{v'}\backslash \vec{w_0}, \vec{w}, \vec{v_0}, \vec{v}]$$

Proof

Given $P : MT_{\vec{u}\to\vec{v}}$ then

$$\mathbf{E}_{\vec{u}\cup\vec{w}\overset{\oplus}{\to}\vec{v}\cup\vec{w}}(P \overset{\vec{v}}{\oplus}\vec{w})$$

(Notice $v\vec{w} \neq v\vec{w'} \equiv \vec{v} \neq \vec{v'} \vee \vec{w} \neq \vec{w'}$)

$\equiv$ Defn w-opening (see page 133.
$\neg\ (P;\ \vec{w} - \vec{v}\mathbf{:}\ [\ True\ ]\ (\vec{v} \neq \vec{v'} \vee \vec{w} \neq \vec{w'}))[\vec{w}, \vec{w'}, \vec{v}, \vec{v'}\backslash \vec{w_0}, \vec{w}, \vec{v_0}, \vec{v}]$
$\equiv$ Specification Statement (A.48)
$\neg\ (P\ (\forall\ \vec{w} - \vec{v} \bullet (\vec{v} \neq \vec{v'} \vee \vec{w} \neq \vec{w'})[\vec{w_0} - \vec{v_0}\backslash \vec{w} - \vec{v}]))[\vec{w}, \vec{w'}, \vec{v}, \vec{v'}\backslash \vec{w_0}, \vec{w}, \vec{v_0}, \vec{v}]$
$\equiv$ Partially Superfluous Universal Quantification (A.21)
$\neg\ (P\ (\vec{v} \neq \vec{v'} \vee \forall\ \vec{w} \bullet \vec{w} \neq \vec{w'}))[\vec{w}, \vec{w'}, \vec{v}, \vec{v'}\backslash \vec{w_0}, \vec{w}, \vec{v_0}, \vec{v}]$
$\equiv$ Universal de Morgan (A.18)
$\neg\ (P\ (\vec{v} \neq \vec{v'} \vee \neg\ \exists\ \vec{w} \bullet \vec{w} = \vec{w'}))[\vec{w}, \vec{w'}, \vec{v}, \vec{v'}\backslash \vec{w_0}, \vec{w}, \vec{v_0}, \vec{v}]$
$\equiv$ Existential Quantification One Point Rule (A.16)
$\neg\ (P\ (\vec{v} \neq \vec{v'} \vee \neg\ true))[\vec{w}, \vec{w'}, \vec{v}, \vec{v'}\backslash \vec{w_0}, \vec{w}, \vec{v_0}, \vec{v}]$
$\equiv$
$\neg\ P(\vec{v} \neq \vec{v'})[\vec{w}, \vec{w'}, \vec{v}, \vec{v'}\backslash \vec{w_0}, \vec{w}, \vec{v_0}, \vec{v}]$

QED

Proof of 7.1 from p133 (Opened Generalised Effect Basic Properties)

Duplicate (Opened Generalised Effect Basic Properties) of 7.1 on page 133.

$$\mathbf{E}((x := e) \overset{\vec{x}}{\oplus}\vec{w}) \equiv x = e[\vec{w}, \vec{x}\backslash \vec{w_0}, \vec{x_0}]$$
$$\mathbf{E}(\vec{z}\mathbf{:}\ [\ p\ ,\ q\ ] \overset{\vec{z}}{\oplus}\vec{w}) \equiv p[\vec{z}, \vec{w}\backslash \vec{z_0}, \vec{w_0}] \Rightarrow q[\vec{w}\backslash \vec{w_0}]$$

There are several sections here.

Specifications:

$\mathbf{E}(\vec{z}\mathbf{:}\ [\ p\ ,\ q\ ]\ \hat{\oplus}\vec{w})$
$\equiv$ Open Generalised Effect (B.2)
$\neg\ (\vec{z}\mathbf{:}\ [\ p\ ,\ q\ ]\ (\vec{z} \neq \vec{z'}))[\vec{z}, \vec{w}, \vec{z'}, \vec{w'}\backslash\vec{z_0}, \vec{w_0}, \vec{z}, \vec{w}]$
$\equiv$ Specification Statement (A.48)
$\neg\ (p \wedge (\forall\, \vec{z} \bullet q \Rightarrow \vec{z} \neq \vec{z'})[\vec{z_0}\backslash\vec{z}])[\vec{z}, \vec{w}, \vec{z'}, \vec{w'}\backslash\vec{z_0}, \vec{w_0}, \vec{z}, \vec{w}]$
$\equiv$ de Morgan's Law
$(p \Rightarrow \neg\ (\forall\, \vec{z} \bullet q \Rightarrow \vec{z} \neq \vec{z'})[\vec{z_0}\backslash\vec{z}])[\vec{z}, \vec{w}, \vec{z'}, \vec{w'}\backslash\vec{z_0}, \vec{w_0}, \vec{z}, \vec{w}]$
$\equiv$ Universal de Morgan (A.18)
$(p \Rightarrow (\exists\, \vec{z} \bullet \neg\ (q \Rightarrow \vec{z} \neq \vec{z'}))[\vec{z_0}\backslash\vec{z}])[\vec{z}, \vec{w}, \vec{z'}, \vec{w'}\backslash\vec{z_0}, \vec{w_0}, \vec{z}, \vec{w}]$
$\equiv$ Definition of Implication
$(p \Rightarrow (\exists\, \vec{z} \bullet \neg\ (\neg\ q \vee \vec{z} \neq \vec{z'}))[\vec{z_0}\backslash\vec{z}])[\vec{z}, \vec{w}, \vec{z'}, \vec{w'}\backslash\vec{z_0}, \vec{w_0}, \vec{z}, \vec{w}]$
$\equiv$ de Morgan's Law
$(p \Rightarrow (\exists\, \vec{z} \bullet q \wedge \vec{z} = \vec{z'})[\vec{z_0}\backslash\vec{z}])[\vec{z}, \vec{w}, \vec{z'}, \vec{w'}\backslash\vec{z_0}, \vec{w_0}, \vec{z}, \vec{w}]$
$\equiv$ Existential Quantification One Point Rule (A.16)
$(p \Rightarrow (q[\vec{z}\backslash\vec{z'}])[\vec{z_0}\backslash\vec{z}])[\vec{z}, \vec{w}, \vec{z'}, \vec{w'}\backslash\vec{z_0}, \vec{w_0}, \vec{z}, \vec{w}]$
$\equiv$ Distributing Substitutions
$p[\vec{z}, \vec{w}, \vec{z'}, \vec{w'}\backslash\vec{z_0}, \vec{w_0}, \vec{z}, \vec{w}] \Rightarrow$
$q[\vec{z}, \vec{z_0}\backslash\vec{z'}, \vec{z}][\vec{z}, \vec{w}, \vec{z'}, \vec{w'}\backslash\vec{z_0}, \vec{w_0}, \vec{z}, \vec{w}]$
$\equiv$ Simplifying
$p[\vec{z}, \vec{w}, \vec{z'}, \vec{w'}\backslash\vec{z_0}, \vec{w_0}, \vec{z}, \vec{w}] \Rightarrow$
$q[\vec{z}, \vec{z'}, \vec{z_0}, \vec{w}, \vec{w'}\backslash\vec{z}, \vec{z}, \vec{z_0}, \vec{w_0}, \vec{w}]$
$\equiv$ Simplifying
$p[\vec{z}, \vec{w}, \vec{z'}, \vec{w'}\backslash\vec{z_0}, \vec{w_0}, \vec{z}, \vec{w}] \Rightarrow$
$q[\vec{z'}, \vec{w}, \vec{w'}\backslash\vec{z}, \vec{w_0}, \vec{w}]$
$\equiv$ Using $\vec{z'}, \vec{w'}$ nfi p, q
$p[\vec{z}, \vec{w}\backslash\vec{z_0}, \vec{w_0}] \Rightarrow$
$q[\vec{w}\backslash\vec{w_0}]$

Assignments:

$(x := e)\ \hat{\oplus}\vec{w}$
$\equiv$ Defn w-opening (see page 133)
$\neg\ ((\vec{x} := \vec{e})(x' \neq x))[\vec{w}, \vec{x}, \vec{w'}, \vec{x'}\backslash\vec{w_0}, \vec{x_0}, \vec{w}, \vec{x}]$
$\equiv$ Assignment (A.36)
$\neg\ (x' \neq x)[\vec{x}\backslash\vec{e}][\vec{w}, \vec{x}, \vec{w'}, \vec{x'}\backslash\vec{w_0}, \vec{x_0}, \vec{w}, \vec{x}]$
$\equiv$
$(x' = e)[\vec{w}, \vec{x}, \vec{w'}, \vec{x'}\backslash\vec{w_0}, \vec{x_0}, \vec{w}, \vec{x}]$
$\equiv$ Only x' free in x'.
$x = (e)[\vec{w}, \vec{x}, \vec{w'}, \vec{x'}\backslash\vec{w_0}, \vec{x_0}, \vec{w}, \vec{x}]$
$\equiv$ $\vec{w'}, \vec{x'}$ *nfi* e
$x = (e)[\vec{w}, \vec{x}\backslash\vec{w_0}, \vec{x_0}]$

QED

B.4 Statement Refinements

Proof of 4.22 from p49 (Open World Specification)

Duplicate (Open World Specification) of 4.22 on page 49.
Using the open world view of the refinement relation, the frame of a specification can be expanded with variables outside the specification's environment. The specification

$$\vec{z}: \left[pre \ , \ post \right]$$

in an environment with state variables $\vec{z}$, disjoint from $\vec{w}$:

$$\vec{z}: \left[pre \ , \ post \right] \sqsubseteq \vec{z}, \vec{w}: \left[pre \ , \ post \right]$$

$$\vec{z}: \left[pre \ , \ post \right] \sqsubseteq \vec{z}, \vec{w}: \left[pre \ , \ post \right]$$

Applying Definition Refinement (4.20) reduces this to the following. For predicate $p : \mathrm{pred}\,\vec{z}$:

$$\Leftarrow$$
$$\vec{z}: \left[pre \ , \ post \right]_{\vec{z}} \ p$$
$$\Rightarrow$$
$$\vec{z}, \vec{w}: \left[pre \ , \ post \right]_{(\vec{z} \sqcap \vec{w})} \ p$$

$\equiv$ Specification Statement (A.48)
$$pre \wedge (\forall \vec{z} \bullet post \Rightarrow p)[\vec{z_0} \backslash \vec{z}]$$
$$\Rightarrow$$
$$pre \wedge (\forall \vec{z}, \vec{w} \bullet post \Rightarrow p)[\vec{z_0}, \vec{w_0} \backslash \vec{z}, \vec{w}]$$

$\equiv$ Since $\vec{w_0}$ and $\vec{w}$ are not free in either p or $post$.
$$pre \wedge (\forall \vec{z} \bullet post \Rightarrow p)[\vec{z_0} \backslash \vec{z}]$$
$$\Rightarrow$$
$$pre \wedge (\forall \vec{z} \bullet post \Rightarrow p)[\vec{z_0} \backslash \vec{z}]$$

QED

Proof of A.29 from p160 (Weaken Assertion)

Duplicate (Weaken Assertion) of A.29 on page 160.
Given predicates $p : \mathrm{pred}\,P$ and $q : \mathrm{pred}\,Q$ such that $\mathrm{pred}\,Q \preceq \mathrm{pred}\,P$ (Q has less state elements than P).

$$\frac{p \Rightarrow q}{\{p\} \sqsubseteq \{q\}}$$

For any postcondition predicate *post*.

$\{p\}$ *post*
$\equiv$ Assertion (A.26)
$\lambda r \bullet p \wedge r$ *post*
$\equiv$ Function Application
$p \wedge$ *post*
$\Rrightarrow$ Antecedent
$q \wedge$ *post*
$\equiv$ Function Abstraction
$\lambda r \bullet q \wedge r$ *post*
$\equiv$ Assertion (A.26)
$\{q\}$ *post*

QED

B.5 Client Constructs Pre Object-Refinement

Proof of 4.37 from p72 (Introduce Field Update (Semantics for Values))

Duplicate (Introduce Field Update (Semantics for Values)) of 4.37 on page 72.

Given object value variable o

$$o.f :: \left[o.\textbf{select}\ f = e \right] \equiv o.\textbf{update}\ f\ \ \textbf{with}\ e$$

$o.f :: \left[o.\textbf{select}\ f = e \right]$
$\equiv$ Object Specifications (Semantics for Values) (4.34)
$$o: \left[\begin{array}{c} o.\textbf{select}\ f = e \wedge \\ \forall\, a \in \{poc(o)\} \setminus (poc(o.f) \cup prec(o.f)) \bullet \\ a = a_0 \end{array} \right]$$
$\equiv$ Eval Update (2.11)
$o: \left[o = o_{0 \odot} f \leftleftarrows e \right]$
$\equiv$ Simple Specification (A.52)
$o := o_{\odot} f \leftleftarrows e$
$\equiv$ Object Field Update (Semantics for Values) (4.36)
$o.\textbf{update}\ f\ \ \textbf{with}\ e$

QED

Proof of 4.47 from p81 (Introduce Field Update (Semantics For References))

Duplicate (Introduce Field Update (Semantics For References)) of 4.47 on page 81.
An object specification where the attribute f of object reference o may be altered such that its final value is p is equivalent to updating field f of o with p.

$$o\uparrow.f :: \left[o\uparrow.\textbf{select}\ f = p \right] \equiv o\uparrow.\textbf{update}\ f\ \ \textbf{with}\ p$$

$o\uparrow.f :: \left[\, o\uparrow.\textbf{select}\ f = p \,\right]_\alpha$
$\equiv$ Object Specifications (Semantics for References) (4.44)
$$store: \left[\begin{array}{c} o\uparrow.\textbf{select}\ f = p \wedge \\ \forall\, n \in \mathrm{dom}(store)\backslash\{o\} \bullet store(n) = store_0(n) \wedge \\ \forall\, m \in attributes(store(o))\backslash\{f\} \bullet \\ store(o).\textbf{select}\ m = store_0(o).\textbf{select}\ m \end{array}\right]_\alpha$$
$\equiv$ Object Field Selection (4.32)
$$store: \left[\begin{array}{c} store(o)_\odot f = p \wedge \\ \forall\, n \in \mathrm{dom}(store)\backslash\{o\} \bullet store(n) = store_0(n) \wedge \\ \forall\, m \in attributes(store(o))\backslash\{f\} \bullet store(o)_\odot m = store_0(o)_\odot m \end{array}\right]_\alpha$$
$\equiv$ Eval Select (2.8)
Eval Update (2.11)
$$store: \left[\begin{array}{c} store(o) = store_0(o)_\odot f \leftleftarrows p \wedge \\ \forall\, n \in \mathrm{dom}(store)\backslash\{o\} \bullet store(n) = store_0(n) \end{array}\right]_\alpha$$
$\equiv$ Property of function override
$store: \left[\, store = store_0 \oplus \{o \mapsto store_0(o)_\odot f \leftleftarrows p\} \,\right]_\alpha$
$\equiv$ Simple Specification (A.52)
$store := store \oplus \{o \mapsto store(o)_\odot f \leftleftarrows p\}$
$\equiv$ Accessed Function Assignment (4.40)
$store(o) := store(o)_\odot f \leftleftarrows p$
$\equiv$ Field Update (Semantics for References) (4.46)
$o\uparrow.\textbf{update}\ f\ \ \textbf{with}\ p$

QED

Proof of 4.35 from p71 (Expanding the Frame with a Postfix Closure)

Duplicate (Expanding the Frame with a Postfix Closure) of 4.35 on page 71.
The object specification frame may be extended with an attribute path (p) that is in the postfix closure of an attribute path (o) that is already in the frame.

$$o :: \left[\, post \,\right] \equiv o, p :: \left[\, post \,\right]$$

provided $p \in poc(o)$.

o:: $\lceil post \rceil$
$\equiv$ Object Specifications (Semantics for Values) (4.34)

$$head(\!| o |\!)\text{:} \left[\begin{array}{c} post \wedge \\ \forall a \in \left(\begin{array}{c} \bigcup\{s \in head(\!| o |\!) \bullet poc(s)\} \setminus \\ \bigcup\{s \in o \bullet poc(s) \cup prec(s)\} \end{array} \right) \bullet a = a_0 \end{array} \right]$$

$\equiv$ If $p \in poc(o)$ then $head(\!| o |\!) \equiv head(\!| o \cup p |\!)$

$$head(\!| o \cup p |\!)\text{:} \left[\begin{array}{c} post \wedge \\ \forall a \in \left(\begin{array}{c} \bigcup\{s \in head(\!| o \cup p |\!) \bullet poc(s)\} \setminus \\ \bigcup\{s \in o \bullet poc(s) \cup prec(s)\} \end{array} \right) \bullet a = a_0 \end{array} \right]$$

$\equiv$ If $p \in poc(o)$ then

$$prec(p) \subseteq (poc(o) \cup prec(o) \wedge poc(p) \subseteq poc(o)$$

and

$$\bigcup\{s \in (o \cup p) \bullet poc(s) \cup prec(s)\}$$
$$\equiv$$
$$\bigcup\{s \in o \bullet poc(s) \cup prec(s)\} \cup poc(p) \cup prec(p)$$

$$head(\!| o \cup p |\!)\text{:} \left[\begin{array}{c} post \wedge \\ \forall a \in \left(\begin{array}{c} \bigcup\{s \in head(\!| o \cup p |\!) \bullet poc(s)\} \setminus \\ \bigcup\{s \in o \cup p \bullet poc(s) \cup prec(s)\} \end{array} \right) \bullet a = a_0 \end{array} \right]$$

$\equiv$ Object Specifications (Semantics for Values) (4.34)
o, p:: $\lceil post \rceil$

QED

B.6 Object-Refinement Proofs

Proof of 5.4 from p84 (Update Object Field)

Duplicate (Update Object Field) of 5.4 on page 84.
Replacing an object's field with an object-refinement is an object-refinement. If object o has field l and $o_{\odot} l \sqsubseteq_{\varsigma} f$, then o's l field can be replaced by f.

$$\frac{o_{\odot} l \sqsubseteq_{\varsigma} f}{o \sqsubseteq_{\varsigma} (o_{\odot} l \Leftarrow f)}$$

The construct $o_{\odot} l \Leftarrow f$[1] is object calculus syntax representing the replacement of o's field l with f.

Straightforward application of Definition Object-Refinement (5.1) and Eval Update (2.11).

QED

Proof of 5.6 from p85 (Update Object Method)

Duplicate (Update Object Method) of 5.6 on page 85.
This refinement rule permits the refinement of an object's method, resulting in an object-refinement. Given an object o with a method l, and a method m that is a refinement of $o_{\odot}l$, then the replacement of $o_{\odot}l$ with m is a valid object-refinement.

$$\frac{o_{\odot}l \sqsubseteq m}{o \sqsubseteq_{\varsigma} (o_{\odot}l \leftleftarrows m)}$$

Straightforward application of Definitions Object-Refinement (5.1), Object-Refinement Predicate Transformers (5.3) and Eval Update (2.11).

QED

Proof of 5.8 from p86 (Introduce Object Attributes)

Duplicate (Introduce Object Attributes) of 5.8 on page 86.
Given an object with fields $f_{1..i}$ and methods $m_{1..k}$, adding new fields $f_{i+1..i+j}$ (for $j \geq 0$) or methods $m_{k+1..k+l}$ (for $l \geq 0$) to an object produces an object-refinement. For field values $fv_{1..i+j}$ of types $F_{1..i+j}$, and methods $mv_{1..k+l}$:

object
  field $f_{h\in 1..i} : F_h := fv_h$
  method $m_{h\in 1..k} = mv_h$
end
$\sqsubseteq_{\varsigma}$
object
  field $f_{h\in 1..i+j} : F_h := fv_h$
  method $m_{h\in 1..k+l} = mv_h$
end

Straightforward application of Definitions Object-Refinement (5.1), and Object-Refinement Reflexive (A.58).
QED

B.7 Client Constructs Re Object-Refinement Proofs

Proof of 5.17 from p94 (Object-Refine in Assignment (Semantics for Values))

Duplicate (Object-Refine in Assignment (Semantics for Values)) of 5.17 on page 94.
For object variable o and (object) expressions e and f:

$$\frac{e \sqsubseteq_\varsigma f}{o := e \sqsubseteq o := f}$$

That is, substituting e for f in the assignment $o := e$ produces the refined statement $o := f$.

Straightforward application of the definitions of refinement, assignment and the monotonicity of predicates.

RHS
$\equiv$ Refinement (4.20)
$\forall A \bullet (o := a\, A) \Rrightarrow (o := c\, A)$
$\equiv$ Assignment (A.36)
$\forall A \bullet A[o\backslash a] \Rrightarrow A[o\backslash c]$
$\Lleftarrow$ Upwards closure of A
$a \sqsubseteq_\varsigma c$

QED

Proof of 5.18 from p94 (Object-Refine in Specification (Semantics for Values))

Duplicate (Object-Refine in Specification (Semantics for Values)) of 5.18 on page 94.
For object variable o and (object) expressions e and f:

$$\frac{e \sqsubseteq_\varsigma f}{o{:}\left[\, e \sqsubseteq_\varsigma o \,\right] \sqsubseteq o{:}\left[\, f \sqsubseteq_\varsigma o \,\right]}$$

Substituting f for e in the specification $o{:}\left[\, e \sqsubseteq_\varsigma o \,\right]$ produces the refined statement $o{:}\left[\, f \sqsubseteq_\varsigma o \,\right]$.

The proof is achieved in two separate ways. The first uses the monotonicity of predicates. The second uses the transitivity of object-refinement.

$o{:}\left[\, o \sqsupseteq_\varsigma e \,\right] \sqsubseteq o{:}\left[\, o \sqsupseteq_\varsigma f \,\right]$
$\equiv$ Refinement (4.20)
$\forall A \bullet o{:}\left[\, o \sqsupseteq_\varsigma e \,\right]\ A \Rrightarrow o{:}\left[\, o \sqsupseteq_\varsigma f \,\right]\ A$
$\equiv$ Specification Statement (A.48)
$\forall A \bullet (\forall o \bullet o \sqsupseteq_\varsigma e \Rightarrow A) \Rrightarrow (\forall o \bullet o \sqsupseteq_\varsigma f \Rightarrow A)$

From this point on, the proof can be achieved using two different methods. Either is sufficient proof. The first uses object-refinement transitivity.

$\forall A \bullet (\forall o \bullet o \sqsupseteq_\varsigma e \Rightarrow A) \Rrightarrow (\forall o \bullet \underline{o \sqsupseteq_\varsigma f \Rightarrow A})$

Opening on the underlined portion, gaining the antecedents from the assumption and derived from the hypothesis. The goal is to close the underlined window with the expression *true*.

$(o \sqsupseteq_\varsigma e \Rightarrow A) \wedge e \sqsubseteq_\varsigma f \Rightarrow (o \sqsupseteq_\varsigma f \Rightarrow A)$
$\equiv$ Double Implication (A.15)
$((o \sqsupseteq_\varsigma e \Rightarrow A) \wedge e \sqsubseteq_\varsigma f \wedge o \sqsupseteq_\varsigma f) \Rightarrow A$
$\equiv$ Using $e \sqsubseteq_\varsigma f \wedge o \sqsupseteq_\varsigma f \Rightarrow o \sqsupseteq_\varsigma e$
That is, Object-Refinement Transitivity (A.57)
$(A \wedge e \sqsubseteq_\varsigma f \wedge o \sqsupseteq_\varsigma f) \Rightarrow A$
$\equiv$ Close with expression *true*
true

The second proof method uses the upward closure of predicates.

$\forall A \bullet (\forall o \bullet o \sqsupseteq_\varsigma e \Rightarrow A) \Rrightarrow (\forall o \bullet o \sqsupseteq_\varsigma f \Rightarrow A)$
$\equiv$ Substitution (using inspiration for choice) with fresh variable d
$\forall A \bullet (\forall o \bullet o \sqsupseteq_\varsigma d \Rightarrow A)[d\backslash e] \Rrightarrow (\forall o \bullet o \sqsupseteq_\varsigma d \Rightarrow A)[d\backslash f]$
$\Leftarrow$ Upwards closure of predicates
$e \sqsubseteq_\varsigma f$

QED

Proof of 5.19 from p94 (Object-Refinement Specification)

Duplicate (Object-Refinement Specification) of 5.19 on page 94.
For constant expression e:

o**:** $\left[e \sqsubseteq_\varsigma o \right] \sqsubseteq o := e$

o**:** $\left[o \sqsupseteq_\varsigma e \right] \sqsubseteq o := e$

$\equiv$ Refinement (4.20)
Specification Statement (A.48)
Assignment (A.36)
$\forall A \bullet \underline{(\forall o \bullet o \sqsupseteq_\varsigma e \Rightarrow A)} \Rrightarrow A[o\backslash e]$

Open window on the underlined portion with the goal being $A[o\backslash e]$ and the window transformation relation being entailment.

$\underline{(\forall o \bullet o \sqsupseteq_\varsigma e \Rightarrow A)}$
$\Rrightarrow$ Universal Quantification Elimination (A.22)
$(o \sqsupseteq_\varsigma e \Rightarrow A)[o\backslash e]$
$\equiv$ Substitution
$e \sqsupseteq_\varsigma e \Rightarrow A[o\backslash e]$
$\equiv$ Object-Refinement Reflexive (A.58)
$A[o\backslash e]$

Close the window.

QED

B.8 Data-refinement for Objects Proofs

Proof of 5.35 from p106 (Data-Refine New)

Duplicate (Data-Refine New) of 5.35 on page 106.

$o := \textbf{new}\ spec \preceq_{DR} o := \textbf{new}\ impl$

Using

$$rep \mathrel{\widehat=} \lambda p \bullet (\exists store_s \bullet o\!\uparrow_{store_s} \preceq_{ODR} o\!\uparrow_{store_i} \land (\{o\} \lhd store_s) \sqsubseteq_\varsigma (\{o\} \lhd store_i) \land p)$$

and $spec \preceq_{ODR} impl$ under rep_{si}:

$o\!\uparrow_s := \textbf{new}\ spec$
$\equiv$ New Operator (5.22)
$o\!:\!\left[\, o = \sqcap x \bullet (\forall j \bullet x \notin \operatorname{dom} store_{s\,j}) \,\right]$;
$o\!\uparrow_s := spec$
$\equiv$ Accessed Function Assignment (4.40)
$o\!:\!\left[\, o = \sqcap x \bullet (\forall j \bullet x \notin \operatorname{dom} store_{s\,j}) \,\right]$;
$store_s := store_s \oplus \{o \mapsto spec\}$

Using Data-refine Sequential Composition (2.16) this can be broken into the data-refinement of each statement. The first data-refines, using Data-refine Specifications (2.15) to

$$o\!:\!\left[\, o = \sqcap x \bullet (\forall j \bullet x \notin \operatorname{dom} store_{i\,j}) \,\right];$$

The second data-refines as follows:

$store_s := store_s \oplus \{o \mapsto spec\}$
$\equiv$ Simple Specification (A.52)
$store_s\!:\!\left[\, store_s = store_{s\,0} \oplus \{o \mapsto spec\} \,\right]$
$\equiv$ Property of function override
$$store_s\!:\!\left[\begin{array}{l} \forall k \in \operatorname{dom} store_{s\,0} \backslash \{o\} \bullet k\!\uparrow_{store_s} \sqsupseteq_\varsigma k\!\uparrow_{store_{s0}} \land \\ \qquad o\!\uparrow_{store_s} \sqsupseteq_\varsigma spec \end{array}\right]$$
$\preceq_{DR}$ Data-refine Specifications (2.15)
$$store_i\!:\!\left[\begin{array}{l} \exists store_s \bullet o\!\uparrow_{store_s} \preceq_{ODR} o\!\uparrow_{store_i} \land \{o\} \lhd store_s \sqsubseteq_\varsigma \{o\} \lhd store_i \land \\ \qquad \forall k \in \operatorname{dom} store_{s\,0} \backslash \{o\} \bullet k\!\uparrow_{store_s} \sqsupseteq_\varsigma k\!\uparrow_{store_{s0}} \land \\ \qquad o\!\uparrow_{store_s} \sqsupseteq_\varsigma spec \end{array}\right]$$
$\sqsubseteq$ Strengthen Postcondition (A.54)
$$store_i\!:\!\left[\begin{array}{l} \exists store_s \bullet store_s = store_i \oplus \{o \mapsto spec\} \\ o\!\uparrow_{store_s} \preceq_{ODR} o\!\uparrow_{store_i} \land \{o\} \lhd store_s \sqsubseteq_\varsigma \{o\} \lhd store_i \land \\ \qquad \forall k \in \operatorname{dom} store_{s\,0} \backslash \{o\} \bullet k\!\uparrow_{store_s} \sqsupseteq_\varsigma k\!\uparrow_{store_{s0}} \land \\ \qquad o\!\uparrow_{store_s} \sqsupseteq_\varsigma spec \end{array}\right]$$

$\equiv$ Existential Quantification One Point Rule (A.16)

$$store_i: \left[\begin{array}{c} o\uparrow_{store_i \oplus \{o \mapsto spec\}} \preceq_{ODR} o\uparrow_{store_i} \wedge \\ \{o\} \lhd (store_i \oplus \{o \mapsto spec\}) \sqsubseteq_\varsigma \{o\} \lhd store_i \wedge \\ \forall k \in \mathrm{dom}(store_{i\ 0} \oplus \{o \mapsto spec\}) \backslash \{o\} \bullet \\ k\uparrow_{(store_i \oplus \{o \mapsto spec\})} \sqsupseteq_\varsigma k\uparrow_{(store_i \oplus \{o \mapsto spec\})\ 0} \wedge \\ o\uparrow_{(store_i \oplus \{o \mapsto spec\})} \sqsupseteq_\varsigma spec \end{array} \right]$$

$\equiv$ Object Dereference (4.41)

$$store_i: \left[\begin{array}{c} spec \preceq_{ODR} o\uparrow_{store_i} \wedge \{o\} \lhd store_i \sqsubseteq_\varsigma \{o\} \lhd store_i \wedge \\ \forall k \in \mathrm{dom}\, store_{i\ 0} \backslash \{o\} \bullet k\uparrow_{store_i} \sqsupseteq_\varsigma k\uparrow_{store_i\ 0} \wedge \\ spec \sqsupseteq_\varsigma spec \end{array} \right]$$

$\equiv$ Simplifying

$$store_i: \left[\begin{array}{c} spec \preceq_{ODR} o\uparrow_{store_i} \wedge \\ \forall k \in \mathrm{dom}\, store_{i\ 0} \backslash \{o\} \bullet k\uparrow_{store_i} \sqsupseteq_\varsigma k\uparrow_{store_i\ 0} \end{array} \right]$$

$\equiv$ Strengthen Postcondition (A.54)

$$store_i: \left[\begin{array}{c} spec \preceq_{ODR} impl \wedge impl \sqsubseteq_\varsigma o\uparrow_{store_i} \wedge \\ \forall k \in \mathrm{dom}\, store_{i\ 0} \backslash \{o\} \bullet k\uparrow_{store_i} \sqsupseteq_\varsigma k\uparrow_{store_i\ 0} \end{array} \right]$$

$\equiv$ Strengthen Postcondition (A.54) using the assumption.

$$store_i: \left[\begin{array}{c} impl \sqsubseteq_\varsigma o\uparrow_{store_i} \wedge \\ \forall k \in \mathrm{dom}\, store_{i\ 0} \backslash \{o\} \bullet k\uparrow_{store_i} \sqsupseteq_\varsigma k\uparrow_{store_i\ 0} \end{array} \right]$$

$\equiv$ Property of function override.

$$store_i: \left[store_i = store_{i\ 0} \oplus \{o \mapsto impl\} \right]$$

$\equiv$ Simple Specification (A.52)
Accessed Function Assignment (4.40)

$$o\uparrow := impl$$

QED

Lemma B.3 (Square Object-Data-Refinement)

$$\frac{spec \preceq_{ODR} impl \wedge impl \sqsubseteq_\varsigma oi}{\exists os \bullet spec \sqsubseteq_\varsigma os \wedge os \preceq_{ODR} oi}$$

B.9 Class Proofs

Proof of 5.38 from p107 (Class Introduction)

Duplicate (Class Introduction) of 5.38 on page 107.
This refinement rule can be used to introduce a new class into an existing class hierarchy. Since a class definition is merely a scoped variable introduction with a specific object initialisation similar rules apply to class introduction as to variable introduction. Namely, a class with class name *Classname* can be introduced into a program *P* provided *Classname* is not free in *P* and *P* is conjunctive.

$$\frac{\textit{Classname}\ \textit{nfi}\ P}{P \sqsubseteq \ |[\ \textbf{class}\ \textit{Classname}\ \textbf{is}\ \textit{Attribs}\ \textbf{end}\ \bullet\ P\ ]|}$$

where *Attribs* is a list of attributes.

Using *Attribs* implicitly as both the object containing the list of attributes *Attribs* and the list of attributes:

P
$\equiv$ Conjunctivity
$w{:}\ [\ \mathbf{A}(P)\ ,\ \mathbf{E}(P)\ ]$
$\sqsubseteq$ Introduce Local Variable Block (A.55)
$|[\ \textbf{var}\ \textit{Classname} : \tau(\textit{Classname})\ \bullet$
$\quad w, \textit{Classname}{:}\ [\ \mathbf{A}(P)\ ,\ \mathbf{E}(P)\ ]$
$]|$
$\sqsubseteq$ Leading assignment and *Classname nfi P*
$|[\ \textbf{var}\ \textit{Classname} : \tau(\textit{Classname})\ \bullet$
$\quad \textit{Classname} := \textit{Attribs};$
$\quad w, \textit{Classname}{:}\ [\ \mathbf{A}(P)\ ,\ \mathbf{E}(P)\ ]$
$]|$
$\equiv$ Scoped Object Definition (Semantics for Values) (4.25)
Contract Frame (A.51)
Conjunctivity
$|[\ \textbf{class}\ \textit{Classname}\ \textbf{is}\ \textit{Attribs}\ \textbf{end}\ \bullet\ P\ ]|$

QED

B.10 Semantics for References Proofs

Proof of 6.6 from p114 (Reference Specification Sequential Composition)

Duplicate (Reference Specification Sequential Composition) of 6.6 on page 114.

Many refinement rules analogous to the classical rules hold for reference specifications. For instance, a rule analogous to Sequential Composition (A.41) is permitted. For *mid* containing no initial variables except a_0, c_0 and $store_0$.

$$
\begin{array}{l}
\vec{a}, \vec{b}\uparrow, \vec{c}^{*}\mathbf{:}\ \left[\, pre\ ,\ post \,\right]_{\star} \\
\sqsubseteq \\
\quad \left|\left[\ \mathbf{con}\ STORE, \vec{A}, \vec{C} \bullet \right.\right. \\
\qquad \vec{a}, \vec{b}\uparrow, \vec{c}\uparrow\mathbf{:}\ \left[\, pre\ ,\ mid \,\right]_{\star}\ ; \\
\qquad \vec{a}, \vec{b}\uparrow, \vec{c}^{*}\mathbf{:}\ \left[\ \dfrac{(mid)[store_0, \vec{a}_0, \vec{c}_0 \backslash STORE, \vec{A}, \vec{C}]}{(post)[store_0, \vec{a}_0 \backslash STORE, \vec{A}]}\ \right]_{\star} \\
\quad \left.\left]\right|\right.
\end{array}
$$

$\vec{a}, \vec{b}\uparrow, \vec{c}^{*}\mathbf{:}\ \left[\, pre\ ,\ post \,\right]_{\star}$

$\widehat{=}$ Reference Specification (6.4)

Constrained (6.5)

$store, \vec{a}, \vec{c}\mathbf{:}\ \left[\, pre\ ,\ post \wedge (\vec{b_0} \cup \vec{c_0}) \lhd (\mathrm{dom}\, store_0 \lhd store) = (\vec{b_0} \cup \vec{c_0}) \lhd store_0 \,\right]$

$\sqsubseteq$ Introduce Sequential Composition (A.43)

b not in frame.

$$
\begin{array}{l}
\left|\left[\ \mathbf{con}\ STORE, \vec{A}, \vec{C} \bullet \right.\right. \\
\qquad store, \vec{a}, \vec{c}\mathbf{:}\ \left[\ pre\ ,\ \begin{array}{c} mid \wedge \vec{c} = \vec{c_0} \wedge \\ (\vec{b} \cup \vec{c_0}) \lhd (\mathrm{dom}\, store_0 \lhd store) = (\vec{b} \cup \vec{c_0}) \lhd store_0 \end{array}\ \right]\ ; \\
\qquad store, \vec{a}, \vec{c}\mathbf{:}\ \left[\ \dfrac{\begin{array}{c} (mid \wedge \vec{c} = \vec{c_0} \wedge \\ (\vec{b} \cup \vec{c_0}) \lhd (\mathrm{dom}\, store_0 \lhd store) = (\vec{b} \cup \vec{c_0}) \lhd store_0) \\ [store_0, \vec{a}_0, \vec{c}_0 \backslash STORE, \vec{A}, \vec{C}] \end{array}}{\begin{array}{c} (post \wedge \\ (\vec{b} \cup \vec{c_0}) \lhd (\mathrm{dom}\, store_0 \lhd store) = (\vec{b} \cup \vec{c_0}) \lhd store_0) \\ [store_0, \vec{a}_0, \vec{c}_0 \backslash STORE, \vec{A}, \vec{C}] \end{array}}\ \right] \\
\quad \left.\left]\right|\right.
\end{array}
$$

$\equiv$ Substitutions

$$
\begin{array}{l}
\left|\left[\ \mathbf{con}\ STORE, \vec{A}, \vec{C} \bullet \right.\right. \\
\qquad store, \vec{a}, \vec{c}\mathbf{:}\ \left[\ pre\ ,\ \begin{array}{c} mid \wedge \vec{c} = \vec{c_0} \wedge \\ (\vec{b} \cup \vec{c}) \lhd (\mathrm{dom}\, store_0 \lhd store) = (\vec{b} \cup \vec{c}) \lhd store_0 \end{array}\ \right]\ ; \\
\qquad store, \vec{a}, \vec{c}\mathbf{:}\ \left[\ \dfrac{\begin{array}{c} (mid)[store_0, \vec{a}_0, \vec{c}_0 \backslash STORE, \vec{A}, \vec{C}] \wedge \\ \vec{c} = \vec{C} \wedge (\vec{b} \cup \vec{c}) \lhd (\mathrm{dom}\, STORE \lhd store) = (\vec{b} \cup \vec{c}) \lhd STORE \end{array}}{\begin{array}{c} (post)[store_0, \vec{a}_0, \vec{c}_0 \backslash STORE, \vec{A}, \vec{C}] \wedge \\ (\vec{b} \cup \vec{C}) \lhd (\mathrm{dom}\, STORE \lhd store) = (\vec{b} \cup \vec{C}) \lhd STORE \end{array}}\ \right] \\
\quad \left.\left]\right|\right.
\end{array}
$$

$\sqsubseteq$ Contract Frame (A.51)
Reference Specification (6.4)

$$
\begin{array}{l}
\big|\big[\ \textbf{con}\ STORE, \vec{A}, \vec{C} \bullet \\
\quad \vec{a}, \vec{b}\uparrow, \vec{c}\uparrow\!\!: \left[\, pre \ ,\ mid \,\right]_\star ; \\
\quad store, \vec{a}, \vec{c}\!: \left[\begin{array}{c} (mid)[store_0, \vec{a}_0, \vec{c}_0 \backslash STORE, \vec{A}, \vec{C}] \wedge \\ \vec{c} = \vec{C} \wedge (\vec{b} \cup \vec{c}) \lhd (\mathrm{dom}\, STORE \lhd store) = (\vec{b} \cup \vec{c}) \lhd STORE \\ \hline (post)[store_0, \vec{a}_0, \vec{c}_0 \backslash STORE, \vec{A}, \vec{C}] \wedge \\ (\vec{b} \cup \vec{C}) \lhd (\mathrm{dom}\, STORE \lhd store) = (\vec{b} \cup \vec{C}) \lhd STORE \end{array}\right] \\
\big]\big|
\end{array}
$$

$\sqsubseteq$ Strengthen Postcondition (A.54)
Using $\vec{c_0} = \vec{C}$ and Lemma B.4.
Weaken Precondition (A.53)

$$
\begin{array}{l}
\big|\big[\ \textbf{con}\ STORE, \vec{A}, \vec{C} \bullet \\
\quad \vec{a}, \vec{b}\uparrow, \vec{c}\uparrow\!\!: \left[\, pre \ ,\ mid \,\right]_\star ; \\
\quad store, \vec{a}, \vec{c}\!: \left[\begin{array}{c} (mid)[store_0, \vec{a}_0, \vec{c}_0 \backslash STORE, \vec{A}, \vec{C}] \\ \hline (post)[store_0, \vec{a}_0 \backslash STORE, \vec{A}] \wedge \\ (\vec{b} \cup \vec{c_0}) \lhd (\mathrm{dom}\, store_0 \lhd store) = (\vec{b} \cup \vec{c_0}) \lhd store_0 \end{array}\right] \\
\big]\big|
\end{array}
$$

$\equiv$ Reference Specification (6.4)

$$
\begin{array}{l}
\big|\big[\ \textbf{con}\ STORE, \vec{A}, \vec{C} \bullet \\
\quad \vec{a}, \vec{b}\uparrow, \vec{c}\uparrow\!\!: \left[\, pre \ ,\ mid \,\right]_\star ; \\
\quad \vec{a}, \vec{b}\uparrow, \vec{c}^{*}\!: \left[\begin{array}{c} (mid)[store_0, \vec{a}_0, \vec{c}_0 \backslash STORE, \vec{A}, \vec{C}] \\ \hline (post)[store_0, \vec{a}_0 \backslash STORE, \vec{A}] \end{array}\right]_\star \\
\big]\big|
\end{array}
$$

QED

Lemma B.4 (Reference Specification Sequential Composition Lemma) Informally, *store* must be shown to be *STORE* possibly modified only at b and c_0 and possibly extended. It is known that $store_0$ is such a modification of *STORE*. One choice then would be to strengthen *store* to $store_0$. Another choice is to allow *store* to be a modified and extended version of $store_0$.

$$
\begin{array}{l}
(\vec{b} \cup \vec{c_0}) \lhd (\mathrm{dom}\, STORE \lhd store_0) = (\vec{b} \cup \vec{c_0}) \lhd STORE) \\
\Rightarrow \\
(\vec{b} \cup \vec{c_0}) \lhd (\mathrm{dom}\, STORE \lhd store) = (\vec{b} \cup \vec{c_0}) \lhd STORE) \\
\Leftarrow \\
(\vec{b} \cup \vec{c_0}) \lhd (\mathrm{dom}\, store_0 \lhd store) = (\vec{b} \cup \vec{c_0}) \lhd store_0)
\end{array}
$$

Proof of 6.7 from p114 (Invariant Aliasing)

Duplicate (Invariant Aliasing) of 6.7 on page 114.
When no reference occurs in the frame, the aliasing constraints of the postcondition can be weakened using those of the precondition.

$$\vec{a}\!\uparrow\!: \left[(\vec{a}^{\,=\vec{b}}_{\neq\vec{c}}) \wedge pre \;,\; (\vec{a}^{\,=\vec{b}}_{\neq\vec{c}}) \wedge post \right]_\star$$
$$\sqsubseteq$$
$$\vec{a}\!\uparrow\!: \left[(\vec{a}^{\,=\vec{b}}_{\neq\vec{c}}) \wedge pre \;,\; post \right]_\star$$

When no reference occurs in the frame, the aliasing constraints of the postcondition can be weakened using those of the precondition.

The theorem is a specialisation of the following proof where $\vec{d} = \vec{b}$ and $\vec{c} = \vec{e}$. The proof is essentially a strengthen postcondition.

$$\vec{a}\!\uparrow\!: \left[(\vec{a}^{\,=\vec{b}}_{\neq\vec{c}}) \wedge pre \;,\; (\vec{a}^{\,=\vec{d}}_{\neq\vec{e}}) \wedge post \right]_\star$$
$\equiv$ Reference Specification (6.4)
$$store\!: \left[(\vec{a}^{\,=\vec{b}}_{\neq\vec{c}}) \wedge pre \;,\; (\vec{a}^{\,=\vec{d}}_{\neq\vec{e}}) \wedge post \wedge \vec{a_0}|\!\lhd store \right]$$
$\sqsubseteq$ Strengthen Postcondition (A.54)
$$store\!: \left[(\vec{a}^{\,=\vec{b}}_{\neq\vec{c}}) \wedge pre \;,\; (\vec{a}^{\,=\vec{b}}_{\neq\vec{c}}) \Rightarrow (\vec{a}^{\,=\vec{d}}_{\neq\vec{e}}) \wedge post \wedge \vec{a_0}|\!\lhd store \right]$$
$\equiv$ Vector Aliased (6.2)
Aliased (6.1)
$$store\!: \left[(\vec{a}^{\,=\vec{b}}_{\neq\vec{c}}) \wedge pre \;,\; (\vec{a}^{\,=\vec{d}\setminus\vec{b}}_{\neq\vec{e}\setminus\vec{c}}) \wedge post \wedge \vec{a_0}|\!\lhd store \right]$$
$\equiv$ Reference Specification (6.4)
$$\vec{a}\!\uparrow\!: \left[(\vec{a}^{\,=\vec{b}}_{\neq\vec{c}}) \wedge pre \;,\; (\vec{a}^{\,=\vec{d}\setminus\vec{b}}_{\neq\vec{e}\setminus\vec{c}}) \wedge post \right]_\star$$

QED

Proof of 6.8 from p118 (Unannotated Coalesced Specification)

Duplicate (Unannotated Coalesced Specification) of 6.8 on page 118.
For variable set E, disjoint from a and $\vec{b}$:

$$a, \vec{b}, E\mathbf{:}\ \left[\ pre \wedge a^{=\vec{b}}\ ,\ post\ \right]_\star$$

coalesces-to

$$a, E\mathbf{:}\ \left[\ pre[\vec{b}\backslash a] \wedge a^{=\varnothing}\ ,\ post[\vec{b}, \vec{b_0}\backslash a, a_0]\ \right]_\star$$

Using $rep\ p\ \widehat{=}\ (\exists\, \vec{b} \bullet \vec{b} = a \wedge p)$ as specified in Section 6.4.3 and $E = F{\uparrow} \cup G^* \cup H$:

$$a, \vec{b}, F{\uparrow}, G^*, H\mathbf{:}\ \left[\ pre \wedge a^{=\vec{b}}\ ,\ post\ \right]_\star$$

$\equiv$ Reference Specification (6.4)

$$store, a, \vec{b}, G, H\mathbf{:}\ \left[\ pre \wedge a^{=\vec{b}}\ ,\ post \wedge (F_0 \cup G_0) \lhd\!\!| store\ \right]$$

$\equiv$ Initial Variable (A.49)

$$\left|\left[\ \mathbf{con}\ STORE, Fcon, Gcon, Hcon, A, \vec{B} \bullet \right.\right.$$
$$store, a, \vec{b}, G, H\mathbf{:}\left[\begin{array}{c} pre \wedge a^{=\vec{b}} \wedge STORE = store \wedge A = a \wedge Fcon = F \wedge \\ Gcon = G \wedge Hcon = H \wedge \vec{B} = \vec{b} \\ \hline (post \wedge (F_0 \cup G_0) \lhd\!\!| store) \\ [store_0, F_0, G_0, H_0, a_0, \vec{b_0}\backslash STORE, Fcon, Gcon, Hcon, A, \vec{B}] \end{array}\right]$$
$$\left.\left]\right|\right.$$

$\preceq_{DR}$ [GM91, Lemma 8]
[MG90, Lemma 3-Validity]
See Section 2.15 for details.

$$\left|\left[\ \mathbf{con}\ STORE, Fcon, Gcon, Hcon, A, \vec{B} \bullet \right.\right.$$
$$store, a, G, H\mathbf{:}\left[\begin{array}{c} (\exists\, \vec{b} \bullet \vec{b} = a \wedge pre \wedge a^{=\vec{b}} \wedge STORE = store \wedge \\ A = a \wedge Fcon = F \wedge Gcon = G \wedge Hcon = H \wedge \\ \vec{B} = \vec{b}) \\ \hline (\exists\, \vec{b} \bullet \vec{b} = a \wedge (post \wedge (F_0 \cup G_0) \lhd\!\!| store) \\ [store_0, F_0, G_0, H_0, a_0, \vec{b_0}\backslash STORE, Fcon, Gcon, Hcon, A, \vec{B}]) \end{array}\right]$$
$$\left.\left]\right|\right.$$

$\equiv$ Existential Quantification One Point Rule (A.16)

$$\left|\left[\ \mathbf{con}\ STORE, Fcon, Gcon, Hcon, A, \vec{B} \bullet \right.\right.$$
$$store, a, G, H\mathbf{:}\left[\begin{array}{c} pre[\vec{b}\backslash a] \wedge a^{=a} \wedge STORE = store \wedge \\ A = a \wedge Fcon = F \wedge Gcon = G \wedge Hcon = H \wedge \\ \vec{B} = a \\ \hline (post \wedge (F_0 \cup G_0) \lhd\!\!| store) \\ [store_0, F_0, G_0, H_0, a_0, \vec{b_0}\backslash STORE, Fcon, Gcon, Hcon, A, \vec{B}][\vec{b}\backslash a] \end{array}\right]$$
$$\left.\left]\right|\right.$$

$\sqsubseteq$ Strengthen Postcondition (A.54)
Weaken Precondition (A.53)

$$\left|\left[\ \textbf{con}\ STORE, Fcon, Gcon, Hcon, A, \vec{B} \bullet \right.\right.$$
$$store, a, G, H : \left[\frac{pre[\vec{b}\backslash a] \wedge a^{=\varnothing}}{\begin{array}{c}(post \wedge (F_0 \cup G_0) \lhd\!\lhd store) \\ [store_0, F_0, G_0, H_0, a_0, \vec{b}_0 \backslash STORE, Fcon, Gcon, Hcon, A, \vec{B}] \\ [\vec{b}\backslash a][STORE, A, Fcon, Gcon, Hcon, \vec{B} \backslash store_0, a_0, F_0, G_0, H_0, a_0]\end{array}}\right]$$
$$\left.\left]\right|\right.$$

$\sqsubseteq$ Substitutions
Remove Logical Constant (A.47)

$$store, a, G, H : \left[pre[\vec{b}\backslash a] \wedge a^{=\varnothing}\ ,\ \begin{array}{c}(post \wedge (F_0 \cup G_0) \lhd\!\lhd store) \\ [\vec{b}_0 \backslash a_0][\vec{b}\backslash a]\end{array} \right]$$

$\equiv$ Reference Specification (6.4)

$$a, E : \left[\ pre[\vec{b}\backslash a] \wedge a^{=\varnothing}\ ,\ post[\vec{b}_0, \vec{b}\backslash a_0, a]\ \right]_\star$$

QED

Proof of 6.12 from p119 (Assignments to the Primary)

Duplicate (Assignments to the Primary) of 6.12 on page 119.
For assignments to primary variable a the following transformation is applicable.

$a := X;$

uncoalesces-to

$a := X;$
$\vec{b} := a, ..., a$

This proof uses the $rep'q \;\widehat{=}\; q \wedge \vec{b} = a$ as presented in Section 6.4.3. This law is similar to the augment assignment rule presented by Morgan [Mor94, Law 17.8]. Given p in the coalesced environment:

$(rep';\ a := X)\ p$

$\equiv$ Law 1.3 Assignment Morgan
$rep'\ (p[a\backslash X])$

$\equiv$ Defn rep'
$p[a\backslash X] \wedge \vec{b} = a$

$\Rrightarrow$
$p[a\backslash X]$

It is recommended that the reader now start from the bottom and read to this point as the development was constructed in such a manner.

$\equiv$ Assignment
$a := X\ (p)$

$\equiv$
$a := X\ (p \wedge a = a)$

$\equiv$ No $\vec{b}$ in coalesced environment.
$a := X\ ((p \wedge \vec{b} = a)[\vec{b}\backslash a])$

$\equiv$ Sequential Composition and Assignment
$(a := X;\ \vec{b} := a, ..., a)\ (p \wedge \vec{b} = a)$

$\equiv$ Defn rep'
$(a := X;\ \vec{b} := a, ..., a;\ rep')\ p$

QED

Proof of 6.13 from p119 (Unannotated Uncoalesced Specification)

Duplicate (Unannotated Uncoalesced Specification) of 6.13 on page 119.
When uncoalescing a specification, the coalesced variables ($\vec{b}$) are returned to the frame and the postcondition is strengthened to ensure that subsequent code can rely on the aliasing of a with $\vec{b}$. For variable set E not containing a:

$$a, E\mathbf{:}\ \left[\ pre\ ,\ post\ \right]_\star$$

uncoalesces-to

$$a, \vec{b}, E\mathbf{:}\ \left[\ a^{=\vec{b}} \wedge pre\ ,\ a^{=\vec{b}} \wedge post\ \right]_\star$$

The theorem proven is simplified by omission of additional frame elements. This proof uses the $rep'q \mathrel{\widehat{=}} q \wedge \vec{b} = a$ as presented in Section 6.4.3.

$$a\mathbf{:}\ \left[\ pre\ ,\ post\ \right]_\star$$

$\equiv$ Reference Specification (6.4)
$$a\mathbf{:}\ \left[\ pre\ ,\ post\ \right]$$

$\equiv$ Initial Variable (A.49)
$$\left|\left[\ \mathbf{con}\ A \bullet \right.\right.$$
$$\quad a\mathbf{:}\ \left[\ pre \wedge a = A\ ,\ post[a_0 \backslash A]\ \right]$$
$$\left.\left]\right|\right.$$

$\preceq_{DR}$ Data-refine Specifications (2.15)
$$\left|\left[\ \mathbf{con}\ A \bullet \right.\right.$$
$$\quad a, \vec{b}\mathbf{:}\ \left[\ \vec{b} = a \wedge pre \wedge a = A\ ,\ \vec{b} = a \wedge post[a_0 \backslash A]\ \right]$$
$$\left.\left]\right|\right.$$

$\sqsubseteq$ Strengthen Postcondition (A.54) with $a_0 = A$
Weaken Precondition (A.53)
Remove Logical Constant (A.47)
$$a, \vec{b}\mathbf{:}\ \left[\ \vec{b} = a \wedge pre\ ,\ \vec{b} = a \wedge post\ \right]$$

$\equiv$ Reference Specification (6.4)
$$a, \vec{b}\mathbf{:}\ \left[\ a^{=\vec{b}} \wedge pre\ ,\ a^{=\vec{b}} \wedge post\ \right]_\star$$

The full proof requires additional reasoning about the omitted frame elements.
QED

Proof of 6.14 from p119 (Dereferenced Uncoalesced Specification)

Duplicate (Dereferenced Uncoalesced Specification) of 6.14 on page 119.
Alternatively, for specifications involving a frame with a dereferenced primary:

$$a\uparrow, E\colon \left[\, pre \;,\; post \,\right]_\star$$

uncoalesces-to

$$a\uparrow, \vec{b}\uparrow, E\colon \left[\, a^{=\vec{b}} \wedge pre \;,\; post \,\right]_\star$$

The strengthening of the postcondition is not required as the frame annotation of a prevents the reference a from being altered.

The theorem proven is simplified by omission of additional frame elements. This proof uses the $rep'q \;\widehat{=}\; q \wedge \vec{b} = a$ as presented in Section 6.4.3.

$$a\uparrow\colon \left[\, pre \;,\; post \,\right]_\star$$

$\equiv$ Reference Specification (6.4)

$$store\colon \left[\, pre \;,\; post \wedge \{a_0\} \lhd\!\mid store \,\right]$$

Using no specification variables (α), and hence no specification variables in the frame (δ), common frame variables $\gamma = \{store\}$ and implementation variables $\beta = \{\vec{b}\}$:

$\preceq_{DR}$ Data-refine Specifications (2.15)

$$\left|\left[\, \textbf{con } STORE \bullet \; \vec{b}, store\colon \left[\begin{array}{ll} \vec{b} = a \wedge pre \wedge & \vec{b} = a \wedge \\ STORE = store \;, & (post \wedge \{a_0\} \lhd\!\mid store)[store_0 \backslash STORE] \end{array}\right] \,\right]\right|$$

$\sqsubseteq$ Strengthen Postcondition (A.54) with
$STORE = store_0$ and using
$\vec{b}_0 = \vec{b} \wedge \vec{b}_0 = a \Rightarrow \vec{b} = a$
Weaken Precondition (A.53)
Remove Logical Constant (A.47)

$$\vec{b}, store\colon \left[\, \vec{b} = a \wedge pre \;,\; \vec{b}_0 = \vec{b} \wedge post \wedge \{a_0\} \lhd\!\mid store \,\right]$$

$\equiv$ Expand Frame (A.50)

$$\vec{b}, store, a\colon \left[\, \vec{b} = a \wedge pre \;,\; a = a_0 \wedge \vec{b}_0 = \vec{b} \wedge post \wedge \{a_0\} \lhd\!\mid store \,\right]$$

$\sqsubseteq$ Strengthen Postcondition (A.54)
Using $\vec{b}_0 = a_0 \Rightarrow \{a_0\} = \{a_0, \vec{b}_0\}$

$$\vec{b}, store, a\colon \left[\, \vec{b} = a \wedge pre \;,\; a = a_0 \wedge \vec{b}_0 = \vec{b} \wedge post \wedge \{a_0, \vec{b}_0\} \lhd\!\mid store \,\right]$$

$\equiv$ Expand Frame (A.50)

$$store\colon \left[\, a^{=\vec{b}} \wedge pre \;,\; post \wedge \{a_0, \vec{b}_0\} \lhd\!\mid store \,\right]$$

$\equiv$ Reference Specification (6.4)

$$a\uparrow, b\uparrow\colon \left[\, a^{=\vec{b}} \wedge pre \;,\; post \,\right]_\star$$

QED

Proof of 6.17 from p121 (Inversed reps)

Duplicate (Inversed reps) of 6.17 on page 121.
Showing that coalesced programming is an instantiation of the 'data-refinement via inverse commands' technique requires proving that rep' is an inverse of rep as shown in Theorem 6.17. That is,

$$rep';\ rep \equiv \{\vec{b} = a\}$$

and hence, as a corollary:

$$rep';\ rep \sqsubseteq \mathbf{skip}$$

Additionally:

$$\mathbf{skip} \equiv rep;\ rep'$$

and hence, as a corollary:

$$\mathbf{skip} \sqsubseteq rep;\ rep'$$

The proof is divided into two sections, showing that $rep';\ rep$ refines to **skip** and that **skip** refines to $rep;\ rep'$.

Given p in uncoalesced (initial) environment with variables $\vec{b}$ with

$$rep\ p\ \widehat{=}\ (\exists \vec{b} \bullet p \wedge \vec{b} = a)$$

and

$$rep'\ p\ \widehat{=}\ p \wedge \vec{b} = a$$

$(rep';\ rep)\ p$

$\equiv$ Defn *rep*
$rep'\ (\exists \vec{b} \bullet p \wedge \vec{b} = a)$

$\equiv$ Existential Quantification One Point Rule (A.16)
$rep'\ (p[\vec{b}\backslash a])$

$\equiv$ Defn rep'
$(p[\vec{b}\backslash a]) \wedge \vec{b} = a$

$\equiv$
$p \wedge \vec{b} = a$

$\equiv$
$\{\vec{b} = a\}\ p$

The second part of the proof uses p in the coalesced environment (no $\vec{b}$ in p). This proof was actually performed from the bottom up and should be read as such.

skip p

$\equiv$

p

$\equiv$ No $\vec{b}$ in p.

$p[\vec{b}\backslash a]$

$\equiv$ Existential Quantification One Point Rule (A.16)

$(\exists\, \vec{b} \bullet p \wedge \vec{b} = a)$

$\equiv$ Choice of *rep*

rep $(p \wedge \vec{b} = a)$

$\equiv$ Choice of *rep*$'$

$(rep;\ rep')p$

QED

Proof of 6.19 from p122 (Transforming Reference Specification)

Duplicate (Transforming Reference Specification) of 6.19 on page 122.
Transforming a reference specification with unaliased variables $\vec{a}$ and $\vec{b}$ involves replacing all dereferences of $\vec{a}$ and $\vec{b}$ with direct variable accesses.

$$\vec{a}\uparrow\!\!: \left[a^{=\varnothing} \wedge b^{=\varnothing} \wedge pre[\vec{a}, \vec{b} \backslash \vec{a}\uparrow, \vec{b}\uparrow] \ , \ post[\vec{a}, \vec{a}_0, \vec{b}, \vec{b}_0 \backslash \vec{a}\uparrow, \vec{a}\uparrow_0, \vec{b}\uparrow, \vec{b}\uparrow_0] \right]_\star$$

encodes-as

$$\vec{a}\!: \left[pre \ , \ post \right]_\star$$

The dereferences of $\vec{a}$ and $\vec{b}$ are replaced with direct variable accesses.

$$\vec{a}\uparrow\!\!: \left[\vec{a}^{=\varnothing} \wedge \vec{b}^{=\varnothing} \wedge pre[\vec{a}, \vec{b} \backslash \vec{a}\uparrow, \vec{b}\uparrow] \ , \ post[\vec{a}, \vec{a}_0, \vec{b}, \vec{b}_0 \backslash \vec{a}\uparrow, \vec{a}\uparrow_0, \vec{b}\uparrow, \vec{b}\uparrow_0] \right]_\star$$

$\equiv$ Reference Specification (6.4)

$$store\!: \left[\vec{a}^{=\varnothing} \wedge \vec{b}^{=\varnothing} \wedge pre[\vec{a}, \vec{b} \backslash \vec{a}\uparrow, \vec{b}\uparrow] \ , \ post[\vec{a}, \vec{a}_0, \vec{b}, \vec{b}_0 \backslash \vec{a}\uparrow, \vec{a}\uparrow_0, \vec{b}\uparrow, \vec{b}\uparrow_0] \wedge \vec{a_0} \lhd\!\!| store \right]$$

$\equiv$ Strengthen Postcondition (A.54)
$\vec{a}$ not in frame.

$$store\!: \left[\vec{a}^{=\varnothing} \wedge \vec{b}^{=\varnothing} \wedge pre[\vec{a}, \vec{b} \backslash \vec{a}\uparrow, \vec{b}\uparrow] \ , \ post[\vec{a}, \vec{a}_0, \vec{b}, \vec{b}_0 \backslash \vec{a}\uparrow, \vec{a}\uparrow_0, \vec{b}\uparrow, \vec{b}\uparrow_0] \wedge \vec{a} \lhd\!\!| store \right]$$

Using the abbreviations $\vec{z} = \vec{a} \cup \vec{b}$ and $\vec{z'} = \vec{a'} \cup \vec{b'}$ this data-refines using the initial environment variables $\alpha = \vec{a} \cup \vec{b} \cup \{store\}$, frame elements that are initial variables $\delta = store$, common frame elements $\gamma = \varnothing$, and mirror environment variables $\beta = \vec{a'} \cup \vec{b'} \cup store'$ and with *rep* chosen as discussed in Section 6.5.3:

$$rep\ p \ \widehat{=} \ \exists \vec{z}, store \bullet p \wedge \vec{z}\uparrow = \vec{z'} \wedge \vec{z} \lhd store = store' \wedge \vec{z}^{=\varnothing}$$

That is:

$\preceq_{DR}$ Data-refine Specifications (2.15)

$$\left| \left[\mathbf{con}\ STORE, \vec{a}, \vec{b}, store \bullet \right.\right.$$
$$\vec{a'}, \vec{b'}, store'\!: \left[\begin{array}{c} \vec{z}\uparrow = \vec{z'} \wedge \\ \vec{z} \lhd store = store' \wedge \\ \hline (\exists\, store \bullet \vec{z}\uparrow = \vec{z'} \wedge \\ \vec{z} \lhd store = store' \wedge \vec{z}^{=\varnothing} \wedge \\ (post[\vec{z}, \vec{z}_0 \backslash \vec{z}\uparrow, \vec{z}\uparrow_0] \wedge \vec{a} \lhd\!\!| store)[store_0 \backslash STORE]) \end{array} \right]$$
$$\left.\left| \right] \right|$$

$\equiv$ Definitions, substitutions and simplifications

$$\left|\left[\ \mathbf{con}\ STORE, \vec{a}, \vec{b}, store \bullet \right.\right.$$
$$\vec{a}', \vec{b}', store' : \left[\begin{array}{c} \vec{z}\upharpoonright = \vec{z'} \wedge \\ \vec{z} \lhd store = store' \wedge \\ \vec{z}^{=\varnothing} \wedge pre[\vec{z}\backslash\vec{z}\upharpoonright] \wedge STORE = store \\ \hline (\exists\, store \bullet \vec{z}\upharpoonright = \vec{z'} \wedge \\ \vec{z} \lhd store = store' \wedge \\ \vec{z}^{=\varnothing} \wedge (post[\vec{z}, \vec{z}_0\backslash\vec{z}\upharpoonright, \vec{z}\upharpoonright_0])[store_0\backslash STORE] \wedge \\ \vec{a} \lhd (\mathrm{dom}\, STORE \lhd store) = \vec{a} \lhd STORE) \end{array}\right]$$
$$\left.\left|\ \right]\right|$$

The current goal is now to strengthen the postcondition so that $store'$ only alters at new indices.

To make the proof more readable, the *store* logical constant is renamed $store_0$ to signify it represents the original value of the initial environment variable *store*.

$\equiv$

$$\left|\left[\ \mathbf{con}\ STORE, \vec{a}, \vec{b}, store_0 \bullet \right.\right.$$
$$\vec{a}', \vec{b}', store' : \left[\begin{array}{c} \vec{z}\upharpoonright_0 = \vec{z'} \wedge \\ \vec{z} \lhd store_0 = store' \wedge \\ \vec{z}^{=\varnothing} \wedge pre[\vec{z}\backslash\vec{z}\upharpoonright][store\backslash store_0] \wedge STORE = store_0 \\ \hline (\exists\, store \bullet \vec{z}\upharpoonright = \vec{z'} \wedge \\ \vec{z} \lhd store = store' \wedge \\ \vec{z}^{=\varnothing} \wedge (post[\vec{z}, \vec{z}_0\backslash\vec{z}\upharpoonright, \vec{z}\upharpoonright_0])[store_0\backslash STORE] \wedge \\ \vec{a} \lhd (\mathrm{dom}\, STORE \lhd store) = \vec{a} \lhd STORE) \end{array}\right]$$
$$\left.\left|\ \right]\right|$$

The postcondition is strengthened using $STORE = store_0$ from the precondition. Subsequently, *STORE* is removed from the precondition and the logical constant frame.

$\sqsubseteq$ Strengthen Postcondition (A.54)
Weaken Precondition (A.53)
Remove Logical Constant (A.47)

$$\left|\left[\ \mathbf{con}\ \vec{a}, \vec{b}, store_0 \bullet \right.\right.$$
$$\vec{a}', \vec{b}', store' : \left[\begin{array}{c} \vec{z}\upharpoonright_0 = \vec{z'} \wedge \\ \vec{z} \lhd store_0 = store' \wedge \\ \vec{z}^{=\varnothing} \wedge pre[\vec{a}, \vec{b}\backslash\vec{a}\upharpoonright, \vec{b}\upharpoonright][store\backslash store_0] \\ \hline (\exists\, store \bullet \vec{z}\upharpoonright = \vec{z'} \wedge \\ \vec{z} \lhd store = store' \wedge \\ \vec{z}^{=\varnothing} \wedge post[\vec{z}, \vec{z}_0\backslash\vec{z}\upharpoonright, \vec{z}\upharpoonright_0] \wedge \\ \vec{a} \lhd (\mathrm{dom}\, store_0 \lhd store) = \vec{a} \lhd store_0) \end{array}\right]$$
$$\left.\left|\ \right]\right|$$

Using $\vec{z} \lhd store_0 = store'_0$ from the precondition, and $\vec{z} \lhd store = store'$ the postcondition is again strengthened considering the following entailment.

Strengthened postcondition:

$(\mathrm{dom}\, store'_0 \lhd store') = store'_0 \wedge \vec{b} \lhd store = \vec{b} \lhd store_0$

$\equiv$ Using $\vec{z} \lhd store_0 = store'_0$.
$(\mathrm{dom}(\vec{z} \lhd store_0) \lhd store') = store'_0 \wedge \vec{b} \lhd store = \vec{b} \lhd store_0$

$\equiv$ Since $\mathrm{dom}\, c \backslash b \equiv \mathrm{dom}(b \lhd c)$.
$((\mathrm{dom}\, store_0 \backslash \vec{z}) \lhd store') = store'_0 \wedge \vec{b} \lhd store = \vec{b} \lhd store_0$

$\equiv$ Since $(b \backslash c) \lhd d = b \lhd (c \lhd d)$.
$(\mathrm{dom}\, store_0 \lhd (\vec{z} \lhd store')) = store'_0 \wedge \vec{b} \lhd store = \vec{b} \lhd store_0$

$\equiv$ Using $store' = \vec{z} \lhd store$.
$(\mathrm{dom}\, store_0 \lhd (\vec{z} \lhd store)) = store'_0 \wedge \vec{b} \lhd store = \vec{b} \lhd store_0$

$\equiv$ Using $d \lhd (b \lhd c) \equiv b \lhd (d \lhd c)$.
$\vec{z} \lhd (\mathrm{dom}\, store_0 \lhd store) = store'_0 \wedge \vec{b} \lhd store = \vec{b} \lhd store_0$

$\equiv$ Using $store'_0 = \vec{z} \lhd store_0$.
$\vec{z} \lhd (\mathrm{dom}\, store_0 \lhd store) = \vec{z} \lhd store_0 \wedge \vec{b} \lhd store = \vec{b} \lhd store_0$

$\equiv$
$(\mathrm{dom}\, store_0 \backslash \vec{z}) \lhd store = (\mathrm{dom}\, store_0 \backslash \vec{z}) \lhd store_0 \wedge \vec{b} \lhd store = \vec{b} \lhd store_0$

Using $(b \lhd w = b \lhd u \wedge c \lhd w = c \lhd u) \equiv (b \cup c) \lhd w = (b \cup c) \lhd u$

$\equiv$
$((\mathrm{dom}\, store_0 \backslash \vec{z}) \cup \vec{b}) \lhd store = ((\mathrm{dom}\, store_0 \backslash \vec{z}) \cup \vec{b}) \lhd store_0$

$\equiv$ Using $((c \backslash d) \backslash e) \cup e = c \backslash d$.
$(\mathrm{dom}\, store_0 \backslash \vec{a}) \lhd store = (\mathrm{dom}\, store_0 \backslash \vec{a}) \lhd store_0$

$\equiv$ Simplifying
$\vec{a} \lhd (\mathrm{dom}\, store_0 \lhd store) = \vec{a} \lhd store_0$

Consequently the previous specification refines as follows.

$\sqsubseteq$ Strengthen Postcondition (A.54)

$$\left|\left[\mathbf{con}\ \vec{a}, \vec{b}, store_0 \bullet \vec{a}', \vec{b}', store' : \left[\begin{array}{c} \vec{z}\uparrow_0 = \vec{z'} \wedge \\ \vec{z} \lhd store_0 = store' \wedge \\ \vec{z}^{=\varnothing} \wedge pre[\vec{z} \backslash \vec{z}\uparrow][store \backslash store_0] \end{array} , \begin{array}{c} (\exists\, store \bullet \vec{z}\uparrow = \vec{z'} \wedge \\ \vec{z} \lhd store = store' \wedge \\ \vec{z}^{=\varnothing} \wedge post[\vec{z}, \vec{z}_0 \backslash \vec{z}\uparrow, \vec{z}\uparrow_0] \wedge \\ \mathrm{dom}\, store'_0 \lhd store' = store'_0 \wedge \\ \vec{b} \lhd store = \vec{b} \lhd store_0) \end{array} \right] \right]\right|$$

To transform the postcondition closer to that desired, substitutions are performed on *post* using $\vec{z}\!\uparrow\!\models \vec{z'}$.

$$
\begin{array}{l}
\equiv \\
\quad \lvert\!\lvert [\, \mathbf{con}\ \vec{a}, \vec{b}, store_0 \bullet \\
\qquad \vec{a'}, \vec{b'}, store' : \left[\begin{array}{c|c} \vec{z}\!\uparrow_0 = \vec{z'} \wedge & (\exists\, store \bullet \vec{z}\!\uparrow\!\models \vec{z'} \wedge \\ \vec{z} \lhd\!\!\cdot\, store_0 = store' \wedge & \vec{z} \lhd\!\!\cdot\, store = store' \wedge \\ \vec{z}^{=\varnothing} \wedge pre[\vec{z}\backslash\vec{z}\!\uparrow][store\backslash store_0] & \vec{z}^{=\varnothing} \wedge post[\vec{z}, \vec{z}_0\backslash\vec{z'}, \vec{z}\!\uparrow_0] \wedge \\ & \mathrm{dom}\, store'_0 \lhd store' = store'_0 \wedge \\ & \vec{b} \lhd store = \vec{b} \lhd store_0) \end{array} \right] \\
\quad]\!\rvert\!\rvert
\end{array}
$$

Goal is now to ensure that $\vec{b}\!\uparrow_0 = \vec{b}\!\uparrow$ and hence $\vec{b'} = \vec{b'_0}$.

$$
\begin{array}{l}
\equiv \text{ Splitting conjuncts} \\
\quad \lvert\!\lvert [\, \mathbf{con}\ \vec{a}, \vec{b}, store_0 \bullet \\
\qquad \vec{a'}, \vec{b'}, store' : \left[\begin{array}{c|c} \vec{z}\!\uparrow_0 = \vec{z'} \wedge & (\exists\, store \bullet \vec{a}\!\uparrow\!\models \vec{a'} \wedge \\ & \vec{b}\!\uparrow\!\models \vec{b'} \wedge \vec{z} \lhd\!\!\cdot\, store = store' \wedge \\ \vec{z} \lhd\!\!\cdot\, store_0 = store' \wedge & \vec{z}^{=\varnothing} \wedge post[\vec{z}, \vec{z}_0\backslash\vec{z'}, \vec{z}\!\uparrow_0] \wedge \\ \vec{z}^{=\varnothing} \wedge pre[\vec{z}\backslash\vec{z}\!\uparrow][store\backslash store_0] & \mathrm{dom}\, store'_0 \lhd store' = store'_0 \wedge \\ & \vec{b} \lhd store = \vec{b} \lhd store_0) \end{array} \right] \\
\quad]\!\rvert\!\rvert
\end{array}
$$

From the precondition it is known that $\vec{b}\!\uparrow_0 = \vec{b'}_0$. Using the postcondition conjunct, $\vec{b} \lhd store = \vec{b} \lhd store_0$ the conjunct $\vec{b}\!\uparrow\!\models \vec{b'}$ is strengthened to $\vec{b'}_0 = \vec{b'}$:

$$
\begin{array}{l}
\vec{b}\!\uparrow_0 = \vec{b'}_0 \wedge \\
\vec{b} \lhd store = \vec{b} \lhd store_0 \wedge \\
\vec{b'}_0 = \vec{b'}
\end{array}
$$

$$
\begin{array}{l}
\equiv \text{ Joining first and third conjuncts.} \\
\vec{b}\!\uparrow_0 = \vec{b'}_0 \wedge \\
\vec{b} \lhd store = \vec{b} \lhd store_0 \wedge \\
\vec{b}\!\uparrow_0 = \vec{b'}
\end{array}
$$

$$
\begin{array}{l}
\Rrightarrow \text{ Second conjunct entails } \vec{b}\!\uparrow_0 = \vec{b}\!\uparrow \\
\vec{b}\!\uparrow_0 = \vec{b'}_0 \wedge \\
\vec{b} \lhd store = \vec{b} \lhd store_0 \wedge \\
\vec{b}\!\uparrow\!\models \vec{b'}
\end{array}
$$

Hence the specification above is developed as follows.

$\sqsubseteq$ Strengthen Postcondition (A.54)

$$\left|\left[\ \mathbf{con}\ \vec{a}, \vec{b}, store_0 \bullet \right.\right.$$
$$\vec{a'}, \vec{b'}, store' : \left[\begin{array}{c} \vec{z}\uparrow_0 = \vec{z'} \wedge \\ \vec{z} \lhd store_0 = store' \wedge \\ \vec{z}^{=\varnothing} \wedge pre[\vec{z}\backslash\vec{z}\uparrow][store\backslash store_0] \end{array} , \begin{array}{c} (\exists\, store \bullet \vec{a}\uparrow = \vec{a'} \wedge \\ \vec{b'}_0 = \vec{b'} \wedge \\ \vec{z} \lhd store = store' \wedge \\ \vec{z}^{=\varnothing} \wedge post[\vec{z}, \vec{z}_0\backslash\vec{z'}, \vec{z}\uparrow_0] \wedge \\ \mathrm{dom}\, store'_0 \lhd store' = store'_0 \wedge \\ \vec{b} \lhd store = \vec{b} \lhd store_0) \end{array} \right]$$
$$\left.\left]\right|\right|$$

Remove $\vec{b'}$ from the frame, strengthen postcondition to remove $\vec{z}^{=\varnothing}$ and choose *store* such that $store \equiv (\vec{a} \lhd A) \cup ((\vec{b} \lhd store_0) \cup store')$ where A is a function that maps $\vec{a}$ to $\vec{a'}$.

$\sqsubseteq$ Expand Frame (A.50)
Strengthen Postcondition (A.54)
Existential Quantification One Point Rule (A.16)

$$\left|\left[\ \mathbf{con}\ \vec{a}, \vec{b}, store_0 \bullet \right.\right.$$
$$\vec{a'}, store' : \left[\begin{array}{c} \vec{z}\uparrow_0 = \vec{z'} \wedge \\ \vec{z} \lhd store_0 = store' \wedge \\ \vec{z}^{=\varnothing} \wedge \\ pre[store\backslash store_0][\vec{z}\backslash\vec{z}\uparrow_0] \end{array} , \begin{array}{c} A(\vec{a}) = \vec{a'} \wedge \\ store' = store' \wedge \\ (post[\vec{z}, \vec{z}_0\backslash\vec{z'}, \vec{z}\uparrow_0])[store\backslash store'] \wedge \\ \mathrm{dom}\, store'_0 \lhd store' = store'_0 \wedge \\ \vec{b} \lhd store_0 = \vec{b} \lhd store_0 \end{array} \right]$$
$$\left.\left]\right|\right|$$

Using the precondition information $\vec{z} \lhd store_0 = store'_0$. Also using $\vec{z}\uparrow_0 = \vec{z'}_0$ in the postcondition.

$\sqsubseteq$ Simplification and substitutions
Weaken Precondition (A.53)
Constrained (6.5)
Strengthen Postcondition (A.54)

$$\left|\left[\ \mathbf{con}\ \vec{a}, \vec{b}, store_0 \bullet \right.\right.$$
$$\vec{a'}, store' : \left[\begin{array}{c} \vec{z} \lhd store_0 = store' \wedge \\ pre[store\backslash store_0][\vec{z}\backslash\vec{z'}] \\ \hline (post[\vec{z}, \vec{z}_0\backslash\vec{z'}, \vec{z'}_0])[store, store_0\backslash store', store'_0] \wedge \\ \varnothing \lhd store' \end{array} \right]$$
$$\left.\left]\right|\right|$$

The only uses of *store* in *pre* are dereferences. Dereferences of references other than $\vec{z}$ are equal to those in *store'*.

$\sqsubseteq$ Simplify
Weaken Precondition (A.53)
Remove Logical Constant (A.47)

$$\vec{a'}, store' : \left[pre[store, \vec{z}\backslash store', \vec{z'}] \ , \begin{array}{c} (post[\vec{z}, \vec{z}_0\backslash\vec{z'}, \vec{z'}_0])[store, store_0\backslash store', store'_0] \wedge \\ \varnothing \lhd store' \end{array} \right]$$

Ignoring dashed annotations:

$$\equiv$$
$$\vec{a}, store\mathbf{:}\left[pre\ ,\ \begin{array}{c} post \wedge \\ \varnothing \lhd\!\!\mid store \end{array} \right]$$

$\equiv$ Reference Specification (6.4)
$\vec{a}\mathbf{:}\left[pre\ ,\ post \right]_{\star}$

QED

Proof of 6.26 from p125 (Inversed reps for Semantic Conversion B)

Duplicate (Inversed reps for Semantic Conversion B) of 6.26 on page 125.

$rep'; rep \sqsubseteq$ **skip**

This proof uses an initial environment with *store* and reference variables $\vec{z}$ and mirror environment with value variables $\vec{z'}$ and *store*. They use the following abstraction invariant which equates the dereferences of $\vec{z}$ with the value variables $\vec{z'}$ and ensures that *store* is the same as $store'$ except at indices $\vec{z}$.

$$AI \ \widehat{=}\ \vec{z} \upharpoonright = \vec{z'} \wedge \vec{z} \lhd store = store'$$

The data type invariant is also used:

$$DTI \ \widehat{=}\ \vec{z}^{=\varnothing}$$

Hence:

$$rep\ p \ \widehat{=}\ \exists \vec{z}, store \bullet p \wedge AI \wedge DTI$$

$$rep'\ p \ \widehat{=}\ \exists \vec{z'}, store' \bullet p \wedge AI \wedge DTI$$

Given *post* on the initial (reference) environment $\{\vec{z}, store\}$:

$$(rep'; rep)\ post$$

$\equiv$ Definitions and function application
$(\lambda q \bullet \exists \vec{z'}, store' \bullet q \wedge AI \wedge DTI)$
$(\exists \vec{z}, store \bullet post \wedge AI \wedge DTI)$

$\equiv$ Definition of *AI* and *DTI*
$(\lambda q \bullet \exists \vec{z'}, store' \bullet q \wedge AI \wedge DTI)$
$(\exists \vec{z}, store \bullet post \wedge \vec{z} \upharpoonright = \vec{z'} \wedge$
$\quad \vec{z}^{=\varnothing} \wedge \vec{z} \lhd store = store')$

$\equiv$ Definition of *AI* and *DTI* and function application
$\exists \vec{z'}, store' \bullet (\exists \vec{Z}, STORE \bullet post[z, store\backslash Z, STORE] \wedge \vec{Z}\uparrow_{STORE} = \vec{z'} \wedge$
$\quad \vec{Z}^{=\varnothing} \wedge \vec{Z} \lhd STORE = store') \wedge \vec{z}\restriction = \vec{z'} \wedge \vec{z} \lhd store = store' \wedge \vec{z}^{=\varnothing}$

The only uses of Z in $post[z, store\backslash Z, STORE]$ are within dereferences. Since the dereferences of z equal those of Z (through z'), the substitution of Z with z is valid. Since *STORE* equals *store* (through *store*$'$) at all other locations, that subsitution is also valid.

$\equiv$ Definition of *AI* and *DTI* and function application
$\exists \vec{z'}, store' \bullet (\exists \vec{Z}, STORE \bullet post \wedge \vec{Z}\uparrow_{STORE} = \vec{z'} \wedge$
$\quad \vec{Z}^{=\varnothing} \wedge \vec{Z} \lhd STORE = store') \wedge \vec{z}\restriction = \vec{z'} \wedge \vec{z} \lhd store = store' \wedge \vec{z}^{=\varnothing}$

Since the existentially quantified variables do no occur in *post* it can be extracted from the scope of the quantifications.

$\equiv$
$post \wedge$
$\exists \vec{z'}, store' \bullet (\exists \vec{Z}, STORE \bullet \vec{Z}\uparrow_{STORE} = \vec{z'} \wedge$
$\quad \vec{Z}^{=\varnothing} \wedge \vec{Z} \lhd STORE = store') \wedge \vec{z}\restriction = \vec{z'} \wedge \vec{z} \lhd store = store' \wedge \vec{z}^{=\varnothing}$

$\Rrightarrow$
$post$

$\equiv$
skip $post$

QED

Proof of 6.25 from p125 (Inversed reps for Semantic Conversion)

Duplicate (Inversed reps for Semantic Conversion) of 6.25 on page 125.
skip $\sqsubseteq$ $rep;\ rep'$

$rep;\ rep'$

$\equiv$ Reps defn
$(\lambda p \bullet \exists \vec{z} \bullet p \wedge AI);$
$(\lambda p \bullet \exists \vec{z'} \bullet p \wedge AI)$

Given *post* on the mirrored (value) environment $\vec{z'}$, an initial environment including the references $\vec{z}$, the rule is shown as a corollary of the following development.

$(rep;\ rep')\ post$

$\equiv$
$(\lambda p \bullet \exists \vec{z} \bullet p \wedge AI)$
$(\exists \vec{z'} \bullet post \wedge AI)$

$\equiv$ Definition of AI
$\lambda\, p \bullet \exists \vec{z} \bullet p \wedge AI$
$(\exists \vec{z'} \bullet post \wedge \vec{z} \upharpoonright = \vec{z'})$

$\equiv$ Existential Quantification One Point Rule (A.16)
$\lambda\, p \bullet \exists \vec{z} \bullet p \wedge AI$
$(post[\vec{z'} \backslash \vec{z} \upharpoonright])$

$\equiv$ Existential Quantification One Point Rule (A.16)
$\exists \vec{z} \bullet (post[\vec{z'} \backslash \vec{z} \upharpoonright]) \wedge AI$

$\equiv$ AI
$\exists \vec{z} \bullet (post[\vec{z'} \backslash \vec{z} \upharpoonright]) \wedge \vec{z} \upharpoonright = \vec{z'})$

(Non-standard application)

$\equiv$ Existential Quantification One Point Rule (A.16)
post

$\equiv$
skip *post*

QED

Proof of 6.21 from p123 (Transforming Value Semantics Assignments)

Duplicate (Transforming Value Semantics Assignments) of 6.21 on page 123.
For the transformation of value semantics variables $\vec{z}$ to reference semantics variables, the assignment

$$D := E$$

decodes-as

$$(D := E)[\vec{z}\backslash\vec{z}\uparrow]$$

Proof is straightforward application of Transforming Value Semantics Specification (6.23) using Simple Specification (A.52).
QED

Proof of 6.23 from p124 (Transforming Value Semantics Specification)

Duplicate (Transforming Value Semantics Specification) of 6.23 on page 124.
Given value variables $\vec{z'}$ and $\vec{a'}$ such that $\vec{a'} \subseteq \vec{z'}$ and references $\vec{z}$ with subset $\vec{a}$, the specification

$$\vec{a'}\mathbf{:}\ \left[\ pre\ ,\ post\ \right]_\star$$

decodes-as

$$\vec{a}\uparrow\mathbf{:}\ \left[\ pre[\vec{z'}\backslash\vec{z}\uparrow] \wedge z^{=\varnothing}\ ,\ post[\vec{a'_0}, \vec{z'}\backslash\vec{a}_0\uparrow, \vec{z}\uparrow]\ \right]_\star$$

The proof uses value semantics variables $\vec{z'}$ partitioned by $\vec{a'}$ and $\vec{b'}$, reference semantics variables $\vec{z}$ partitioned by $\vec{a}$ and $\vec{b}$, initial environment variables $\{\vec{z'}, store'\}$, mirror environment variables $\{\vec{z}, store\}$ and the following abstraction invariant.

$$AI\ \widehat{=}\ \vec{z}\uparrow_{store} = \vec{z'} \wedge \vec{z}^{=\varnothing} \wedge \vec{z} \lhd store = store'$$

Proof:

$\vec{a'}\mathbf{:}\ \left[\ pre\ ,\ post\ \right]_\star$
$\equiv$ Reference Specification (6.4)
$\vec{a'}, store'\mathbf{:}\ \left[\ pre\ ,\ post \wedge \varnothing \lhd\!| store'\ \right]$
$\preceq_{DR}$ Data-refine Specifications (2.15)
Using $\alpha\ \widehat{=}\ \{\vec{z'}, store'\}$
$\delta\ \widehat{=}\ \{\vec{a'}, store'\}$
$\gamma\ \widehat{=}\ \{\}$
$\beta\ \widehat{=}\ \{\vec{z}, store\}$

$$\left|\left[\ \mathbf{con}\ \vec{A}', STORE', \vec{z}', store' \bullet \right.\right.$$
$$\vec{z}, store: \left[\begin{array}{cc} \vec{z}\upharpoonright = \vec{z}' \wedge & \\ \vec{z}^{=\varnothing} \wedge & (\exists\, \vec{a}', store' \bullet \vec{z}\upharpoonright = \vec{z}' \wedge \vec{z}^{=\varnothing} \wedge \\ \vec{z} \lhd store = store' \wedge\ , & \vec{z} \lhd store = store' \wedge \\ pre \wedge \vec{A}' = \vec{a}' \wedge & (post)[\vec{a}'_0, store'_0 \backslash \vec{A}', STORE']) \\ STORE' = store' & \end{array}\right]$$
$$\left.\left.\ \right]\right|$$

Open window on the specification.

The immediate goals are to remove the existential quantification in the postcondition and to rearrange to precondition to the precondition in the desired specification.

$\equiv$ Equivalence substitution in precondition on *pre*
using $\vec{z}\upharpoonright = \vec{z}'$
Split conjunct in postcondition using assumption.

$$\vec{z}, store: \left[\begin{array}{c} \vec{z}\upharpoonright = \vec{z}' \wedge \vec{z}^{=\varnothing} \wedge \\ \vec{z} \lhd store = store' \wedge \\ pre[\vec{z}' \backslash \vec{z}\upharpoonright] \wedge \vec{A}' = \vec{a}' \wedge STORE' = store' \end{array}\ ,\ \begin{array}{c} (\exists\, \vec{a}', store' \bullet \vec{a}\upharpoonright = \vec{a}' \wedge \\ \vec{b}\upharpoonright = \vec{b}' \wedge \vec{z}^{=\varnothing} \wedge \\ \vec{z} \lhd store = store' \wedge \\ (post)[\vec{a}'_0, store'_0 \backslash \vec{A}', STORE']) \end{array}\right]$$

$\equiv$ Existential Quantification One Point Rule (A.16)
Using $store'$ is not free in *post*

$$\vec{z}, store: \left[\begin{array}{c} \vec{z}\upharpoonright = \vec{z}' \wedge \vec{z}^{=\varnothing} \wedge \\ \vec{z} \lhd store = store' \wedge \\ pre[\vec{z}' \backslash \vec{z}\upharpoonright] \wedge \\ \vec{A}' = \vec{a}' \wedge STORE' = store' \end{array}\ ,\ \begin{array}{c} \vec{b}\upharpoonright = \vec{b}' \wedge \vec{z}^{=\varnothing} \wedge \\ (post)[\vec{a}'_0, store'_0, \vec{a}' \backslash \vec{A}', STORE', \vec{a}\upharpoonright] \end{array}\right]$$

The aliases $\vec{z}$ can be altered, however this is not needed by the rule that is being proven. Consequently, the frame can be contracted.

$\sqsubseteq$ Contract Frame (A.51)
Knowing that $\vec{z}_0$ is not free in *post*

$$store: \left[\begin{array}{c} \vec{z}\upharpoonright = \vec{z}' \wedge \vec{z}^{=\varnothing} \wedge \\ \vec{z} \lhd store = store' \wedge \\ pre[\vec{z}' \backslash \vec{z}\upharpoonright] \wedge \\ \vec{A}' = \vec{a}' \wedge STORE' = store' \end{array}\ ,\ \begin{array}{c} \vec{b}\upharpoonright = \vec{b}' \wedge \vec{z}^{=\varnothing} \wedge \\ (post)[\vec{a}'_0, store'_0, \vec{a}' \backslash \vec{A}', STORE', \vec{a}\upharpoonright] \end{array}\right]$$

$\equiv$ Equivalence substitutions on *post* using $\vec{b}\upharpoonright = \vec{b}'$

$$store: \left[\begin{array}{c} \vec{z}\upharpoonright = \vec{z}' \wedge \vec{z}^{=\varnothing} \wedge \\ \vec{z} \lhd store = store' \wedge \\ pre[\vec{z}' \backslash \vec{z}\upharpoonright] \wedge \\ \vec{A}' = \vec{a}' \wedge STORE' = store' \end{array}\ ,\ \begin{array}{c} \vec{b}\upharpoonright = \vec{b}' \wedge \vec{z}^{=\varnothing} \wedge \\ (post)[\vec{a}'_0, store'_0, \vec{z}' \backslash \vec{A}', STORE', \vec{z}\upharpoonright] \end{array}\right]$$

$\sqsubseteq$ Strengthen Postcondition (A.54)
Using $z^{=\varnothing}[store \backslash store_0] \Rrightarrow z^{=\varnothing}$ to eliminate $z^{=\varnothing}$

$$store: \left[\begin{array}{c} \vec{z}\upharpoonright = \vec{z}' \wedge \vec{z}^{=\varnothing} \wedge \\ \vec{z} \lhd store = store' \wedge \\ pre[\vec{z}' \backslash \vec{z}\upharpoonright] \wedge \\ \vec{A}' = \vec{a}' \wedge STORE' = store' \end{array}\ ,\ \begin{array}{c} \vec{b}\upharpoonright = \vec{b}' \wedge \\ (post)[\vec{a}'_0, store'_0, \vec{z}' \backslash \vec{A}', STORE', \vec{z}\upharpoonright] \end{array}\right]$$

$\sqsubseteq$ Strengthen Postcondition (A.54)

Using the following to exchange $\vec{b}' = \vec{b}\uparrow$ with $\vec{b}\uparrow = \vec{b}\uparrow_0$

$(\vec{z}' = \vec{z}\uparrow)[store \backslash store_0] \wedge \vec{b}\uparrow = \vec{b}\uparrow_0$

$\Rrightarrow$

$\vec{b}' = \vec{b}\uparrow_0 \wedge \vec{b}\uparrow = \vec{b}\uparrow_0$

$\Rrightarrow$

$\vec{b}' = \vec{b}\uparrow$

to get

$$store\mathbf{:}\left[\begin{array}{cc} \vec{z}\uparrow = \vec{z'} \wedge \vec{z} = \varnothing \wedge & \\ \vec{z} \lhd store = store' \wedge & \vec{b}\uparrow = \vec{b}\uparrow_0 \wedge \\ pre[\vec{z'} \backslash \vec{z}\uparrow] \wedge & \quad , \quad (post)[\vec{a'}_0, store'_0, \vec{z}' \backslash \vec{A}', STORE', \vec{z}\uparrow] \\ \vec{A}' = \vec{a}' \wedge STORE' = store' & \end{array}\right]$$

From the precondition it is known that $\vec{z} \lhd store_0 = store'$ and $STORE' = store'$ and hence $\vec{z} \lhd store_0 = STORE'$. The only uses of $store'_0$ in *post* are possible dereferences of variables other than $\vec{z}$. Given the equivalence of $STORE'$ and $store_0$ on variables other than $\vec{z}$, $store_0$ can be substituted for $STORE'$ in *post*.

$\sqsubseteq$ Strengthen Postcondition (A.54)

$$store\mathbf{:}\left[\begin{array}{cc} \vec{z}\uparrow = \vec{z'} \wedge \vec{z} = \varnothing \wedge & \\ \vec{z} \lhd store = store' \wedge & \vec{b}\uparrow = \vec{b}\uparrow_0 \wedge \\ pre[\vec{z'} \backslash \vec{z}\uparrow] \wedge & \quad , \quad (post)[\vec{a'}_0, store'_0, \vec{z}' \backslash \vec{A}', store_0, \vec{z}\uparrow] \\ \vec{A}' = \vec{a}' \wedge STORE' = store' & \end{array}\right]$$

The only constraint on *store* is the dereferences of indices $\vec{b}$ remain stable. This is strengthened to all indices except $\vec{a}$ (and new indices).

$\sqsubseteq$ Strengthen Postcondition (A.54)

Using $\vec{a}\uparrow \lhd store \Rrightarrow \vec{b}\uparrow = \vec{b}\uparrow_0$

to get

$$store\mathbf{:}\left[\begin{array}{cc} \vec{z}\uparrow = \vec{z'} \wedge \vec{z} = \varnothing \wedge & \\ \vec{z} \lhd store = store' \wedge & \vec{a}\uparrow \lhd store \wedge \\ pre[\vec{z'} \backslash \vec{z}\uparrow] \wedge & \quad , \quad (post)[\vec{a'}_0, store'_0, \vec{z}' \backslash \vec{A}', store_0, \vec{z}\uparrow] \\ \vec{A}' = \vec{a}' \wedge STORE' = store' & \end{array}\right]$$

$\sqsubseteq$ Strengthen Postcondition (A.54)

Using

$$(\vec{z}\restriction = \vec{z'} \wedge \vec{A'} = \vec{a'})[store\backslash store_0] \wedge (post)[\vec{a'}_0, store'_0, \vec{z'}\backslash \vec{a}_0, store_0, \vec{z}\restriction]$$

$\equiv$ Substitutions

$$(\vec{z}\restriction = \vec{z'} \wedge \vec{A'} = \vec{a'})[store\backslash store_0] \wedge (post)[\vec{a'}_0, store'_0, \vec{z'}\backslash \vec{A'}, store_0, \vec{z}\restriction][\vec{A'}\backslash \vec{a}_0]$$

$\Rrightarrow$ Splitting first conjunct and substitutions

$$\vec{a}\restriction_0 = \vec{a'} \wedge \vec{A'} = \vec{a'} \wedge (post)[\vec{a'}_0, store'_0, \vec{z'}\backslash \vec{A'}, store_0, \vec{z}\restriction][\vec{A'}\backslash \vec{a}_0]$$

$\Rrightarrow$

$$\vec{a}\restriction_0 = \vec{A'} \wedge (post)[\vec{a'}_0, store'_0, \vec{z'}\backslash \vec{A'}, store_0, \vec{z}\restriction][\vec{A'}\backslash \vec{a}_0]$$

$\Rrightarrow$

$$(post)[\vec{a'}_0, store'_0, \vec{z'}\backslash \vec{A'}, store_0, \vec{z}\restriction]$$

to get

$$store\mathbf{:}\left[\begin{array}{c} \vec{z}\restriction = \vec{z'} \wedge \vec{z}^{=\varnothing} \wedge \\ \vec{z} \lhd store = store' \wedge \\ pre[\vec{z'}\backslash \vec{z}\restriction] \wedge \\ \vec{A'} = \vec{a'} \wedge STORE' = store' \end{array} \;,\; \begin{array}{c} \vec{a}\restriction \lhd store \wedge \\ (post)[\vec{a'}_0, store'_0, \vec{z'}\backslash \vec{a}_0, store_0, \vec{z}\restriction] \end{array}\right]$$

Substitution of $store'$ with $store$ on *pre* since it only contains dereferences of variables other than $\vec{z}$.

$\equiv$

$$store\mathbf{:}\left[\begin{array}{c} \vec{z}\restriction = \vec{z'} \wedge \vec{z}^{=\varnothing} \wedge \\ \vec{z} \lhd store = store' \wedge \\ pre[\vec{z'}, store'\backslash \vec{z}\restriction, store] \wedge \\ \vec{A'} = \vec{a'} \wedge STORE' = store' \end{array} \;,\; \begin{array}{c} \vec{a}\restriction \lhd store \wedge \\ (post)[\vec{a'}_0, store'_0, \vec{z'}\backslash \vec{a}_0, store_0, \vec{z}\restriction] \end{array}\right]$$

$\sqsubseteq$ Weaken Precondition (A.53)

$$store\mathbf{:}\left[\vec{z}^{=\varnothing} \wedge pre[\vec{z'}, store'\backslash \vec{z}\restriction, store] \;,\; \begin{array}{c} \vec{a}\restriction \lhd store \wedge \\ (post)[\vec{a'}_0, store'_0, \vec{z'}\backslash \vec{a}_0, store_0, \vec{z}\restriction] \end{array}\right]$$

Ignoring the implicit annotations which were added for proof to distinguish the variables syntactically.

$\equiv$ Reference Specification (6.4)

$$\vec{a}\restriction\mathbf{:}\left[\vec{z}^{=\varnothing} \wedge pre[\vec{z'}\backslash \vec{z}\restriction] \;,\; (post)[\vec{a'}_0, \vec{z'}\backslash \vec{a}_0, \vec{z}\restriction]\right]_\star$$

Close window to obtain:

$$\equiv \quad \left|\left[\, \mathbf{con}\ \vec{A'}, STORE', \vec{z'}, store' \bullet \right.\right.$$
$$\left.\left. \vec{a}\restriction\mathbf{:}\left[\vec{z}^{=\varnothing} \wedge pre[\vec{z'}\backslash \vec{z}\restriction] \;,\; (post)[\vec{a'}_0, \vec{z'}\backslash \vec{a}_0, \vec{z}\restriction]\right]_\star \,\right]\right|$$

$\sqsubseteq$ Remove Logical Constant (A.47)

$$\vec{a}\restriction\mathbf{:}\left[\vec{z}^{=\varnothing} \wedge pre[\vec{z'}\backslash \vec{z}\restriction] \;,\; (post)[\vec{a'}_0, \vec{z'}\backslash \vec{a}_0, \vec{z}\restriction]\right]_\star$$

QED

Proof of 6.24 from p124 (Transforming Value Semantics Specification B)

Duplicate (Transforming Value Semantics Specification B) of 6.24 on page 124.

This rule provides an alternative for the transformation of a value semantics specification into a reference semantics. It provides extra flexibility in the alteration of the references $\vec{a}$ yet still requires them to remain unaliased.
Given value variables $\vec{z'}$ with subset $\vec{a'}$ and references $\vec{z}$ with subset $\vec{a}$ the value semantics specification

$$\vec{a'}\mathbf{:}\ [\, pre\ ,\ post\,]$$

decodes-as

$$\vec{a}^{*}\mathbf{:}\ \Big[\, pre[\vec{z'}\backslash\vec{z}\!\uparrow] \wedge \vec{z}^{=\varnothing}\ ,\ \vec{a}^{=\varnothing} \wedge post[\vec{a'_0}, \vec{z'}\backslash\vec{a}_0\!\uparrow, \vec{z}\!\uparrow]\,\Big]_{\star}$$

The proof is similar to that of 6.23, except that rather than contracting the frame by $\vec{z}$, it is contracted by $\vec{b}$ and the postcondition is strengthened with a predicate that ensures the remaining variables $\vec{a}$ remain unaliased: $\vec{a}^{=\varnothing}$.

QED

B.11 Simultaneous Execution Statement Proofs

Proof of 7.4 from p136 (Introduce Chaotic Simultaneous Execution)

Duplicate (Introduce Chaotic Simultaneous Execution) of 7.4 on page 136.
Any conjunctive code can be refined by simultaneously executing it simultaneously with $\vec{w}$**:** $\left[\, True \,\right]$ for variables $\vec{w}$ of types $\vec{W}$. For conjunctive pt defined on environment $\vec{p}$,

$$pt \sqsubseteq pt\ {}_{\vec{p}}|_{\vec{w}}\ \vec{w}\textbf{:}\ \left[\, True \,\right]$$

This proof was constructed from the bottom up and should be read in that fashion.
For conjunctive pt in environment $\vec{p}$:

pt

$\equiv$ Least Conjunctive Refinement (7.2)
$\vec{p}\textbf{:}\ \left[\, \mathbf{A}(pt)\ ,\ \mathbf{E}(pt) \,\right]$

$\sqsubseteq$ Open World Specification (4.22)
For $\vec{q}$ outside the environment.
$\vec{p}, \vec{q}\textbf{:}\ \left[\, \mathbf{A}(pt)\ ,\ \mathbf{E}(pt) \,\right]$

$\sqsubseteq$ Strengthen Postcondition (A.54)
$\vec{p}, \vec{q}\textbf{:}\ \left[\, \mathbf{A}(pt)\ ,\ \mathbf{A}(pt)[p\backslash p_0] \Rightarrow \mathbf{E}(pt) \,\right]$

$\equiv$ Since $\vec{q}$ is not free in pt.
$\vec{p}, \vec{q}\textbf{:}\ \left[\, \mathbf{A}(pt)\ ,\ \mathbf{A}(pt)[p, q\backslash p_0, q_0] \Rightarrow \mathbf{E}(pt)[q\backslash q_0] \,\right]$

$\equiv$ Opened Generalised Effect Basic Properties (7.1)
$\vec{p}, \vec{q}\textbf{:}\ \left[\, \mathbf{A}(pt)\ ,\ \mathbf{E}(\vec{p}\textbf{:}\ \left[\, \mathbf{A}(pt)\ ,\ \mathbf{E}(pt) \,\right] \,\vec{\oplus}\, \vec{q}) \,\right]$

Heuristically choosing the conjunct. Also using $pt \equiv \vec{p}\textbf{:}\ \left[\, \mathbf{A}(pt)\ ,\ \mathbf{E}(pt) \,\right]$

$\sqsubseteq$ Strengthen Postcondition (A.54)
$\vec{p}, \vec{q}\textbf{:}\ \left[\, \mathbf{A}(pt) \wedge True\ ,\ \mathbf{E}(pt \,\vec{\oplus}\, \vec{q}) \wedge \mathbf{E}(\vec{q}\textbf{:}\ \left[\, True\ ,\ True \,\right] \,\vec{\oplus}\, \vec{p}) \,\right]$

Using $True \equiv \mathbf{A}(\vec{q}\textbf{:}\ \left[\, True\ ,\ True \,\right])$.

$\equiv$
$\vec{p}, \vec{q}\textbf{:}\ \left[\, \mathbf{A}(pt) \wedge \mathbf{A}(\vec{q}\textbf{:}\ \left[\, True\ ,\ True \,\right])\ ,\ \mathbf{E}(pt \,\vec{\oplus}\, \vec{q}) \wedge \mathbf{E}(\vec{q}\textbf{:}\ \left[\, True\ ,\ True \,\right] \,\vec{\oplus}\, \vec{p}) \,\right]$

$\equiv$ Definition Simultaneous Execution
$pt\ {}_{\vec{p}}|_{\vec{q}}\ \vec{q}\textbf{:}\ \left[\, True\ ,\ True \,\right]$

QED

Appendix C

Symbol Glossary

This appendix provides a glossary and index for the syntax used in this thesis. The page reference provides the definition of the symbol.

$pt_p \textcircled{\wedge}_q qt$	Miraculous conjunction of *pt* and *qt* predicate transformers.	p134
$pt \,{}_p\|_q\, qt$	Simultaneous execution of *pt* and *qt* predicate transformers.	p132
$\mathbf{A}(pt)$	The generalised assertion of statement *pt*.	p133
$\mathbf{E}_{u \overset{\oplus}{\rightarrow} v}(pt)$	The generalised effect of statement *pt*.	p133
$pt \overset{''}{\oplus} v$	Enlarges statement state space of *pt*.	p133
false	Boolean falsity.	p39
true	Boolean truth.	p39
$\wedge$	Boolean conjunction.	
$\Rightarrow$	Boolean implication.	
$\neg$	Boolean logical negation.	
$\Leftrightarrow$	Boolean equivalence.	
$\vee$	Boolean disjunction.	
$(OT_1 \sqcap OT_2)$	Greatest lower bound of object types.	p40
$(OT_1 \sqcup OT_2)$	Least upper bound of object types.	p41
$(\sqcap i \in I \bullet OT_i)$	Generalised greatest lower bound of object types.	p41

$(\bigsqcup i \in I \bullet OT_i)$ p42
Generalised least upper bound of object types.

$\{l : \mathbb{B}\}_{\boxed{\text{ST}}}$ The type representing the state which has a variable l of type $\mathbb{B}$. p42

$\top_{\boxed{\text{ST}}}$ The empty state type. p42

$\{l : \mathbb{B}\}_{\boxed{\text{S}}}$ A state with one variable l of type $\mathbb{B}$ p42

$\alpha \setminus_{\boxed{\text{S}}} \{l\}$ State α has identifer l cut from it. p43

$\alpha \cup_{\boxed{\text{S}}} \beta$ States α and β are combined. p43

True Predicate truth.

False Predicate falsity. p45

$\Rightarrow$ Predicate implication. p45

$\wedge$ Predicate conjunction.

$\vee$ Predicate disjunction.

$\Leftrightarrow$ Predicate equivalence.

$A \sqcap B$ The demonic choice of statements A and B. p47

$A \sqcup B$ The angelic choice of statements A and B.

$(\sqcap i \in I \bullet S_i)$ p162
Generalised demonic choice.

$(\ ^{\circ}_{9}\ i \in I \bullet S_i)$
Generalised sequential composition.

$\langle st \rangle$ Lifts the state transoformer *st* to a predicate transformer. p161

$wp(Prog, post,)$ p15
The weakest precondition function that returns the weakest precondition that establishes *post* after executing *Prog*.

$\lceil p \rceil$ Lifts the predicate p to a guard. p160

$\{p\}$ Lifts the predicate p to an assertion. p159

$\lceil p \rceil_{\alpha}$ Validity of predicate p for state type α. p45

Ptrans α β p46
Predicate Transformer type from a postcondition on state α to a precondition state β.

$pt_1 \Rrightarrow pt_2$ p46

Predicate transformer entailment.

$A \sqsubseteq B$ Statement A refines to statement B. p48

$A \sqsupseteq B$ Statement A is a refinement of statement B.

$A \not\sqsubseteq B$ Statement A does not refine to statement B.

$A \preceq_{DR} B$ Statement A data-refines to statement B. p20

Ref Reference type p74

$\Cup$ Special function used in definition of semantics for references object specifications to union the ranges of bags. p79

$a\uparrow$ The dereference value of reference variable a. p76

$f :: [\, pre \;,\; post \,]$ p70

An object specification is a specification statement with additional constraints specifically designed for restricting the alteration of object attributes.

Object
field $f : F$
method m
end p58

A predicate transformer object-type. The host object is accessed through *self*.

object
field $f : F := fv$
method $m = mv$
end p58

A predicate transformer object. The host object is accessed through *self*.

$A \sqsubseteq_{\varsigma} B$ Object A object-refines to object B. p83

$A \preceq_{ODR} B$ p101

Object A object-data-refines to object B.

class *classname* **is**
field $f : F := fv$
method $m = mv$
end p107

A predicate transformer class.

$a \lhd store$ *store* is constrained so that all variables except those in a are equivalent to the their initial values in $store_0$. New locations can also be added. p113

$a \lhd (S, T)$ Store S is the same as T except at new locations and those in a. p113

$(a^{=\vec{b}}_{\neq\vec{c}})$	The variable a is aliased with those in $\vec{b}$, and they are not aliased with those in $\vec{c}$.	p111
$(a^{=B}_{\neq C})$	Vector aliased function.	p111
$a\!:\left[\,pre\;,\;post\,\right]_\star$	Reference specifications allow constraints about the store to be left implicit.	p113
$\#$	Bag size.	
$=_\alpha$	Equality on type α.	p13
$\perp$	The bottom type.	
$\alpha \preceq \beta$	α is a subtype of β.	p6
$\mathbb{Z}$	The set of integer numbers.	
$\mathbb{R}$	The set of real numbers.	
$\varnothing$	The empty environment.	
$E \vdash f$	Value f is well formed in environment E.	p9
$E \vdash T$	Type T is well formed in environment E.	p9
$\vdash f$	Value f is well formed in an irrelevant environment.	p9
$\vdash T$	Type T is well formed in an irrelevant environment.	p9
A *nfi* B	Variable A is not free in term B.	p11
$o :_\perp O$	Object o has least type O.	p11
$x_\odot l \leftleftarrows b$	Object calculus method update.	p14
$\varsigma(x : X)\ m$	Object calculus method construction.	p12
$[x\backslash v]$	Object calculus syntactic substitution of x with v.	p11
$x_\odot l$	Object calculus method invocation.	p13
Objtype $\{\ l : \mathbb{B}\ \}$	Object calculus object-types. An introduction is provided in Section 2.2.	p11
object $\{\ l = 6\ \}$	Object calculus object. See object calculus object types.	

$\mu(X)\ [\ C\{X\}\ ]$ p52

Recursive object type.

typecase $a\ \mid\ (x : A)d_1\ \mid\ d_2$ p53

Object calculus dynamic type checking.

Appendix D

Glossary

This glossary provides concise definitions and standardisations of terminology used in this thesis. It also acts as an index. The first page reference provides the definition, following which are the pages at which the concept is used in a significant fashion.

Abstract Data Type, 4 An abstract data type is a data structure and a set of operations that are used to access and manipulated the data structure. The operations provided are the only operations that may access and manipulate the data structure. An object is an example of an abstract data type in which the data is encapsulated with its operations.

Alexandrov topology, 91

Aliasing, 108 Aliasing occurs when two or more variables refer to the same entity.

Attributes, 5 The combination of an object's fields and methods.

Axiomatic Semantics, 37 An axiomatic semantics defines the meaning of a program in an implicit manner. An example is the provision of a programming language semantics via Hoare triples.

Binary Methods, 25 A binary method is a method which has a value parameter and result parameter which is of the same type as that of the host.

Boolean Lattice, 39 A boolean lattice is a lattice in which every element has a unique complement (also in the lattice).

Class, 5 A class is a template from which object instances can be created.

Class-Based, 82 "Class-based languages form the mainstream of object-oriented programming." e.g., Simula, Smalltalk, C++ [AC96, p11]. The actual properties of class-based languages are controversial. As such, this thesis purports to provide many fundamental features, leaving it to the language designer to include those features deemed appropriate.

Class-based languages are object-oriented languages which have classes.

Client Enhancement, 98

Cloning, 9 Cloning is the process of copying an object.

Coalescing, 117 The merging of two aliases into one variable.

Complete Lattice, 39 A complete lattice is a lattice which has a top and bottom element.

Construction Monotonicity, 82, 96 The ability to substitute a class with a subclass in a **new** statement.

Contravariance, 28 See also Covariance. An object's component type is contravariant if it is permitted to vary anti-monotonically.

Covariance, 28, 34 Covariance is object theory terminology for the monotonicity of a component (type) within an object type with respect to the subtyping relationship.

Data type, 17 A data type is an abstract data type. In the context of data-refinement, the 'abstract' is dropped to avoid confusion.

Data-refinement, 17 Data-refinement is a relationship between two data types; typically between an abstract, mathematically clear data type and a concrete, implementation-like data type. Simulation is the replacement of the abstract data type with the concrete data type in client code. Typically it is performed to increase the efficiency and/or produce implementable code. This relationship may take the form of a predicate, or as in this thesis, a predicate transformer.

In refinement calculi, when all components of a variable block (or module or class) have been data-refined, the variable block introducing the abstract data type is replaced with one which introduces the concrete data type. The 'new', concrete variable block is said to simulate the original. In fact, the concrete variable block is a refinement of the abstract.

Dynamic Constraint, 61, 86, 102

Dynamic Dispatch, 6, 27 The technique used to decide the code to execute for a method invocation of an object.

Fields, 5 Non-invocable, mutable, data attributes of objects.

Host, 5, 68 The host of an attribute (or object) is the object in which the attribute (object) is contained.

Inheritance, 6 Inheritance is the sharing of attributes between a class and its subclass.

Initial Environment, 115

Interference, 135 in the context of aliasing.

Lambda Calculus, 9

Lattice, 39 A partially ordered set in which the meet and join exist for all pairs of elements.

Method, 5 An invocable, procedural portion of an object.

Miraculous Conjunction, 134 The execution of two statements such that the effects of both statements are achieved.

Mirror Environment, 115

Modular Reasoning, 26 The behavioural conformance of polymorphic objects.

Multiple Dispatch, 27 Multiple dispatch occurs when more than one object (or class) is used to determine the method to execute.

Nested Objects, 84

Object Calculus, 9 Object calculi are used to provide the foundations for object-oriented language in a fashion analogous to manner in which λ-calculi provide the foundations for procedural languages.

Object Identity, 108

Object Invariant, 61, 63, 86, 102

Object-Based, 82 "Object-based languages simplify and generalise class-based languages by reducing classes to more primitive notions." [AC96, p1]

Object-Data-Refinement, 98

Object-Refinement, 83

Operational Semantics, 37 Operational semantics are based on implementable models, e.g., stacks and stores (also known as closures, stores are functions mapping locations to values). They provide a machine oriented description of how the language works.

Polymorphism, 6 Typically used to refer to subtyping polymorphism which is the use of an object as an instance of a supertype.

Primary Variable, 117 When two variables are coalesced, the one remaining is termed the *primary*.

Product Types, 31 Product types are better known under the name Cartesian products.

Protected System, 64, 87 A protected system is one in which an object's (private) attributes can only be modified through method calls.

Prototype, 9 A prototype is the analogy of a class in an object-based language. A prototype is an object used as a template for cloning.

Reference Assignment, 95

Reference cloning, 96 The copying of an object in a semantics for references.

Reference Specification, 110 An enhanced specification that implicitly encapsulates the store constraints in a semantics for references.

Simulation, 17, 22 see Data-refinement. In the context of data-refinement, simulation, (actually a special case refinement) is the replacement of an abstract data type by its data-refinement in client code.

The process starts with two data types where it has been shown that they exist in a data-refinement relationship. Simulation is the proof of refinement between the abstract client code and the concrete client code.

The behaviour of the abstract client code has been 'simulated' by the concrete client code—even though they work on different state spaces.

Store, 108 A store is a function mapping references (pointers to objects) to their values.

Subclassing, 6 The word 'subclassing' has various meanings depending upon its context. In the literature review, its definition is provided within the context it is used. For class based refinement as in Section 5.5, a subclass denotes a class refinement that is also a subtype. Hence subclass instances can be used in place of the superclass instances.

Subsumption, 6 The recognition of an object as an instance of a supertype of its own type.

Sum Types, 31 Sum types are the composition of multiple distinct base types. The base types are projected into the summation. No two projected elements of the base types map to the same element within the summation.

Upwards Closed, 91

Weakest Liberal Precondition, 16

Bibliography

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996. {2,4,7,9,11,36,43,50,51,52,53,55,156,211,213}

[AL97] M. Abadi and K. Rustan M. Leino. A Logic of Object-oriented Programs. In *Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 682—696. Springer, April 1997. {36}

[AM91] S. N. Ahmed and J. M. Morris. Constructing and refining modules in a type theory. In J. M. Morris and R. C. Shaw, editors, *Fourth Refinement Workshop*, BCS Workshops in Computing series, pages 411–440. Springer-Verlag, 1991. {37}

[Bac80] R.J.R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*. Tract 131, Mathematisch Centrum, Amsterdam, 1980. {15}

[Bac88] R. J. R. Back. Data refinement in the refinement calculus. Series A 68, Inst. för Informationsbehandling, Åbo Akademi, Turku, Finland, 1988. {17}

[Ban97] P. G. Bancroft. *Module Refinement Refined (for the calculation of Oberon programs)*. PhD thesis, Department of Computer Science, University of Queensland, July 1997. {1,40,74,109,126}

[Bav81] H. Bavendregt. *The lambda calculus: its syntax and semantics*. North-Holland, 1981. {2,9}

[BB94] R. J. R. Back and M. J. Butler. Exploring summation and product operators in the refinement calculus. Technical Report A152, Åbo Akademi, 1994. A94-152. {132,140}

[BB95] R. J. Back and M. Butler. Exploring summation and product operators in the refinement calculus. In B. Möller, editor, *Mathematics of Program Construction*, number 947 in Lecture Notes in Computer Science, pages 128–158. Springer-Verlag, 1995. {31,38,132}

[BCC+95] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3), 1995. {25}

[BMvW97] R.-J. Back, A. Mikhajlova, and J. von Wright. Class refinement as semantics of correct subclassing. Technical Report TUCS Technical Report No. 147, Turku Centre for Computer Science (TUCS), Finland, December 1997. {1,82,102}

[But97] M. Butler. Calculational derivation of pointer algorithms from tree operations. Technical Report DSSE-TR-97-5, Declarative Systems and Software Engineering Group, Department of Electronics and Computer Science, University of Southampton, December 1997. Trees using pointers. {109}

[BvW90] R. J. R. Back and J. von Wright. Command lattices, variable environments and data refinement. Series A 102, Åbo Akademi – Departments of Computer Science and Mathematics, 1990. {17,115}

[BvW97] R. J. Back and J. von Wright. Programs on product spaces. Technical Report 143, Turku Centre for Computer Science, November 1997. {132,134}

[BvW98] R. J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. {1,15,22,23,39,79,87,133,135,159}

[BvW99] R.-J. Back and J. von Wright. Products in the refinement calculus. Technical Report 235, Turku Centre for Computer Science, February 1999. {132,134}

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, and H. Gilchrist. *Object-oriented Development: The Fusion Method.* Prentice Hall, 1994. {5}

[CHC94] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *Theoretical Aspects of Object-Oriented Programming; Types, Semantics and Language Design*, pages 497–517. MIT Press, 1994. {2,6,8}

[CN99a] A. Calvanti and D. A. Naumann. A weakest precondition semantics for an object-oriented language of refinement. In *FM'99 Volume II LNCS 1709*, volume 2, pages 1439–1459. Springer-Verlag, Berlin Heidelberg, 1999. {25}

[CN99b] A. Calvanti and D. A. Naumann. A weakest precondition semantics for an object-oriented language of refinement - extended version. Technical report, Stevens Institute of Technology, 1999. {i}

[CN00] A. Cavalcanti and D. A. Naumann. A weakest precondition semantics for an object-oriented language of refinement. *IEEE Transactions on Software Engineering*, 8(26):713–728, 2000. {1,34,39,58}

[Coo89] W. R. Cook. A proposal for making eiffel type-safe. In *European Conference of Object-Oriented Programming*, pages 57–72, 1989. {54}

[dF95] C. C. de Figueiredo. A proof system for a sequential object-based language. Technical Report UMCS-95-1-1, University of Manchester, Computer Science Department, Jan 1995. {37}

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Academic Press, 1976. {1,15,163}

[dREB+98] W. P. de Roever, K. Engelhardt, K.-H. Buth, P. Gardiner, Y. Lakhnech, and F. Stomp. *Data refinement : model-oriented proof methods and their comparison*. Number 47 in Cambridge tracts in theoretical computer science. Cambridge, UK ; New York, NY : Cambridge University Press, 1998. {19,22}

[DW91] N. B. Dale and C. Weems. *Introduction to Pascal and structured design*. Lexington, 1991. {76}

[GHJV96] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, 1996. {3,5,140}

[GHP95] P. Di Gianantonio, F. Honsell, and G. Plotkin. Uncountable limits and lambda-calculus. *Nordic Journal of Computation*, 2(2):p126–145, 1995. {34,58}

[GK96] S. J. Goldsack and S. J. H. Kent, editors. *Formal Methods and Object Technology*. Springer, 1996. {2,5,9,220}

[GM91] P.H.B. Gardiner and C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87(1):143–162, September 1991. {20,21,166,185}

[GM93] P.H.B. Gardiner and C. Morgan. A single complete rule for data refinement. *Formal Aspects of Computing*, 5(4):367–382, 1993. Reprinted in [MV94]. {17}

[Gro98] L. Groves. Adapting program derivations using program conjunction. In *International Refinement Workshop and Formal Methods Pacific*, pages 145–164. Springer-Verlag, 1998. {87,132,134,136,149}

[Gro00] L. Groves. *Evolutionary Software Development in the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 2000. {87,134}

[HHS87] C.A.R. Hoare, J.F. He, and J.W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2):71–76, May 1987. {22}

[Hoa72] C.A.R. Hoare. Proof of correctness of data representation. *ActInf*, 1:271–281, 1972. {22}

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process*. Addison-Wesley, 1999. {5,141}

[Jon92] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal methods in system design*, 8(2):p105–122, March 1992. {111}

[Kin99] S. King. *Higher-Level Algorithmic Structures in the Refinement Calculus*. PhD thesis, Oxford University Computing Laboratory, Programming Research Group, Oxford University, January 1999. {92,151}

[LBG96] K. Lano, J. C. Bicarregui, and S. Goldsack. Formalising design patterns. In *BCS-FACS Northern Formal Methods Workshop*. Springer-Verlag, September 1996. {3}

[Lei95] K. Rustan M. Leino. *Towards Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. PhD thesis, California Institute of Technology. {36}

[Lei97] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *4th International Workshop on Foundations of Object-Oriented Languages*, January 1997. {36}

[LH94] K. Lano and H. Haughton. *Object-Oriented Specification Case Studies*. Prentice Hall, 1994. {1,2}

[LLRS95] C. Lewerentz, Th. Lindner, A. Rüping, and E. Sekerinski. On object-oriented design and verification. In Manfred Broy and Stefan Jähnichen, editors, *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, volume 1009 of *Lecture Notes in Computer Science*, pages 92–111. Springer-Verlag, New York, 1995. {36}

[LW94] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994. {61,66,67}

[Mah99] B. P. Mahony. The least conjunctive refinement and promotion in the refinement calculus. *Formal Aspects of Computing*, 11:75–105, 1999. {49,132,134}

[Mar67] A. Margaris. *First Order Mathematical Logic*. Blaisdell, 1967. {158}

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997. {1,4,5}

[MG90] C. Morgan and P. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990. Reprinted in [MV94]. {17,20,22,185}

[Mik99a] A. Mikhajlova. Consistent extension of components in the presence of explicit invariants. In *29th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 29)*, pages 76–85. IEEE Computer Society Press, June 1999. Reprinted in [Mik99b, Paper5]. {68}

[Mik99b] A. Mikhajlova. *Ensuring Correctness of Object and Component Systems*. PhD thesis, Turku Centre for Computer Science, 1999. {153,219}

[Mil71] R. Milner. An algebraic definition of simulation between programs. Technical report, Stanford University, Department of Computer Science, February 1971. {17,22}

[Mor87] J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9:287–306, 1987. {15}

[Mor88] C. Morgan. Procedures, parameters and abstraction. *Science of Computer Programming*, (11), 1988. Referenced in [Utt95] along with Apt81 and Rey78. {109}

[Mor89] J. M. Morris. Laws of data refinement. *Acta Informatica*, 26:287–308, 1989. {17}

[Mor90] J. M. Morris. Piecewise data refinement. In E. W. Dijkstra, editor, *Formal Development of Programs and Proofs*, chapter 10, pages 117–137. Addison-Wesley, 1990. {17}

[Mor94] C. Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994. {1,15,18,62,67,94,97,103,135,155,158,159,160,161,163,164,165,187}

[MS97] A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, Lecture Notes in Computer Science 1313, pages 82–101. Springer, 1997. Also published in [Mik99b, Paper1]. {*i*,1,26,31,38,39,50,61,136}

[MV94] C. Morgan and T. Vickers, editors. *On the Refinement Calculus*. Springer-Verlag, 1994. {1,20,217,219}

[Nau94a] D. A. Naumann. On the essence of Oberon. In *Programming Languages and System Architectures*, volume 783 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. {34,151}

[Nau94b] D. A. Naumann. Predicate transformer semantics of an Oberon-like language. *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, 56:467–487, 1994. {25,34,91}

[PJ99] M. Page-Jones. *Fundamentals of object-oriented design in UML*. The Addison-Wesley object technology series. New York, Dorset House Pub., Reading, Mass., Addison-Wesley, November 1999. {5,141}

[Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1995. {5}

[PS94] J. Palsberg and M. Schwartzbach. *Object Oriented Type Systems*. Wiley, 1994. {2,50,51}

[Rie96] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996. {5}

[Sch88] D. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Wm. C. Brown Publishers, Dubuque, Iowa, 1988. {34,58}

[Sek96] E. Sekerinski. *A Type-Theoretic Basis for an Object-Oriented Refinement Calculus*, pages 317—331. Volume 15 of S. J. Goldsack [GK96], 1996. {27,36,43,61,159,161,162,167}

[SLU88] L. A. Stein, H. Lieberman, and D. Ungar. A shared view of sharing: The treaty of Orlando. In *Object-oriented concepts, databases and applications*, pages 31–48. Addison-Wesley, 1988. {2,9}

[UR92] M. Utting and K. Robinson. Modular reasoning in an object-oriented refinement calculus. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction, Second International Conference*, volume 669 of *Lecture Notes in Computer Science*, pages 344–367. Springer-Verlag, 1992. {*i*,1,26,39}

[Utt92] M. Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning*. PhD thesis, University of New South Wales, Kensington, Australia, 1992. {25,26,29,38,39,49,50,58,64,109}

[Utt95] M. Utting. Reasoning about aliasing. Technical report, School of Computer Science and Engineering, The University of New South Wales, April 1995. {40,74,109,219}

[vW92a] J. von Wright. Data refinement and the simulation method. Technical Report A137, Åbo Akademi, 1992. {17}

[vW92b] J. von Wright. The lattice of data refinement. Technical Report A92-130, Åbo Akademi University, Lemminkainengatan 14 A, SF-20520 Turku, Finland, Febuary 1992. {17}