

Formal Methods applied to Object-Oriented Programming

A thesis submitted to the University of Manchester
for the degree of Ph.D. in the Faculty of Science

1992

Alan Cameron Wills

Department of Computer Science

Contents

1-1	Objectives: formal methods into OOP.....	9
1-1.1	Object orientation is a Good Thing	9
1-1.1.1	<i>Re-use</i>	9
1-1.1.2	<i>Responsiveness to change</i>	9
1-1.2	Why it hasn't happened yet (much)	10
1-1.2.1	<i>Re-use between systems</i>	10
1-1.2.2	<i>Change and re-use within a product or product line</i>	10
1-1.3	Formal Methods are the Solution	11
1-2	The focus of this work: Fresco	11
1-2.1	The state and history of Fresco	12
1-3	The wider context: OO software engineering.....	12
1-3.1	Effects of OOP on the lifecycle	12
1-3.2	OO Analysis and Design	13
1-3.3	OO formal methods	13
1-4	Scope of this work	14
1-5	Structure of this thesis.....	15
2-1	The formalised goodie	16
2-1.1	Units of development effort	16
2-1.2	System composition	17
2-2	Types and classes in Fresco	18
2-3	Specification in Fresco.....	20
2-3.1	A type	20
2-3.1.1	<i>Role of types in code</i>	21
2-3.1.2	<i>Types and subtypes</i>	21
2-3.1.3	<i>Type extension</i>	22
2-3.1.4	Model-oriented specification	22
2-3.1.5	<i>Preconditions and invariants</i>	23
2-3.1.6	<i>Inheritance and subtyping</i>	23
2-3.1.7	Strengthening invariants	24
2-3.1.8	<i>Operation extension</i>	24
2-3.2	Generic types	25
2-3.3	Model refinement	25
2-3.4	Proofs and theories	26
2-3.4.1	<i>Model refinement proofs</i>	27
2-3.5	Operation decomposition	28
2-3.5.1	<i>Types and classes</i>	28
2-3.5.2	<i>Code development</i>	28
2-4	System composition.....	30
2-4.1	Capsule contents and composition	31
2-5	Summary	31
3-1	Formal methods	33
3-1.1	Specification styles	33
3-1.2	Traditional specification methods	34
3-1.3	Decomposition strategies	34
3-1.4	Inference methods	35
3-1.5	Proof tools	36
3-1.6	Logic	37
3-2	Object orientation	37
3-2.1	Definition of object-orientation	37
3-2.2	Subclasses	37
3-2.3	Types	38
3-3	Existing syntheses of object orientation and formal methods	40

3-3.1	Object-Z	40
3-3.2	Z++	40
3-3.3	OOZE	40
3-3.4	Abel	41
3-3.5	Larch/Smalltalk	41
3-3.6	VDM++	41
3-3.7	CDL and EVDM	41
3-3.8	Utting & Robson — OO Refinement Calculus	42
3-3.9	Eiffel	42
3-3.10	Annotated C++	42
3-3.11	POOL	42
3-4	Issues in application of formal methods to OOP	43
3-4.1	Concurrency	43
3-4.2	Objects	43
3-4.2.1	Object identity	43
3-4.2.2	Aliasing	43
3-4.3	Encapsulation	44
3-4.3.1	Units of encapsulation	44
3-4.3.2	Applicability of encapsulation to specifications	44
3-4.3.3	Encapsulation and invariants	45
3-4.4	Classes and types	45
3-4.4.1	Classes and types, subclasses and subtypes	45
3-4.4.2	Composing types	46
3-4.4.3	Monotonicity	47
3-4.5	Inheritance and subtyping	47
3-4.5.1	Subranges are not subtypes (for objects)	48
3-4.5.2	Reducing nondeterminism	49
3-4.5.3	Values can be range-restricted	51
3-4.6	Generic definitions	51
3-4.6.1	Subtyping amongst generics	51
3-4.6.2	Constraints on type parameters	52
3-4.7	Assertions and proofs	52
3-4.7.1	Wide-spectrum imperative OO languages	52
3-4.7.2	Expressiveness of the specification component	53
3-4.7.3	Verification	53
3-4.8	Larger units of design	53
3-4.9	Programming language issues	54
3-4.10	Semantics	55
3-5	How Fresco tackles these issues	55
3-6	Summary	55
4-1	Theories	57
4-1.1	Symbols	58
4-1.2	Expressions	59
4-1.2.1	Operator and message expressions	59
4-1.2.2	Binding-expressions	60
4-1.2.3	Blocks	60
4-1.3	Theorems	61
4-1.3.1	Substitution	62
4-1.4	Special constructs for the kernel	62
4-1.4.1	Two-way theorems	62
4-1.4.2	Theorem schemata	63
4-1.5	Contexts	63
4-2	Proofs	63
4-2.1	Informal and rigorous proofs	63
4-2.2	Matching	64
4-2.3	Subproofs	65
4-2.3.1	Proof construction	66

4-2.4	Oracles	66
4-3	Context operations	67
4-3.1	Union of contexts	67
4-3.2	Extraction from context	67
4-4	Comparison with Mural's proof system	68
4-4.1	Theories and proofs	68
4-4.2	Logic	68
4-5 Summary	68
5-1	Specifications and code	70
5-1.1	Code	70
5-1.2	Specification Statements	71
5-1.2.1	Code invariants	72
5-1.2.2	Code specs	72
5-1.2.3	Opspecs	72
5-2	Decomposition proofs	73
5-2.1	Basic rules	73
5-2.1.1	Strengthening	73
5-2.1.2	Spec-statement conjunction	73
5-2.1.3	Sequence	74
5-2.1.4	Condition	75
5-2.1.5	Loop	75
5-2.1.6	Inline form	76
5-2.2	Assignment	77
5-2.3	Operation invocation and results	78
5-2.3.1	Yield of an expression	79
5-3	Issues in the use of programming language in assertions	79
5-3.1	Underdetermined expressions	79
5-3.2	Promotion	80
5-3.2.1	Satisfying Promotions	82
5-4	Summary	82
6-1	Types	83
6-1.1	Type theories	83
6-1.2	Membership of types	83
6-1.2.1	Object Histories	84
6-1.2.2	Type membership	84
6-2	Subtypes	85
6-2.1	Defining a subtype by inheritance	85
6-2.2	Type product	86
6-2.3	Co-product and type category	86
6-3	Type definitions	87
6-3.1	Type context extraction	88
6-3.2	Model-oriented type definitions	89
6-3.3	Type invariants	89
6-3.4	Functions	90
6-3.5	Property-oriented specifications	90
6-3.6	Type Box composition	91
6-3.7	Signatures	91
6-3.8	Type inference	93
6-4	Generic types	94
6-4.1	Constant parameters	94
6-4.2	Type parameters	94
6-4.3	Generic types of immutable values	95
6-4.4	Generic types of mutable objects	95
6-4.5	Generic types with restricted parameters	96
6-4.6	Subtyping among generics	98
6-5	Creation and verification of subtypes	98

6-5.1	Subtyping proofs	98
6-5.2	Varieties and purposes of subtyping	99
6-5.3	Subtyping and inheritance	100
6-5.4	Implementability	100
6-5.4.1	<i>Implementability from scratch</i>	100
6-5.4.2	<i>Implementability and restrictive subtyping</i>	101
6-5.4.3	<i>Implementability and additional opspects</i>	102
6-5.4.4	<i>Implementability and extension</i>	102
6-5.5	Reification	102
6-6	Reification example	103
6-6.1	Supertype: compiler's symbol table	103
6-6.2	Refinement: Dictionary of Stacks	104
6-6.3	Verifying refinement	104
6-7	Creation functions	106
6-8	Types and classes	106
6-8.1	Syntactical considerations	106
6-8.2	Theorems and concrete features	107
6-9	Summary	108
7-1	Systems are compositions of capsules	110
7-1.1	Systems	110
7-1.2	Capsules	111
7-1.3	Definitions in a capsule	112
7-2	Capsule composition	114
7-2.1	The executable system	114
7-2.2	The specifications	114
7-3	Capsule certification and incorporation	116
7-3.1	Changes to theorems	116
7-3.2	Capsules and type conformance	116
7-3.3	Proof expectations	116
7-3.4	Certification and incorporation checks	118
7-3.5	Conflict and resolution	118
7-3.6	Renaming	119
7-3.6.1	<i>Hiding</i>	119
7-4	Global variables	120
7-4.1	Initialisation and persistence	120
7-4.2	Conformance	120
7-4.3	Incorporation scenario with globals	121
7-5	Creation functions and concreteness	122
7-6	The User Interface	123
7-7	Summary	124
8-1	Constraint maintenance: co-operating objects	125
8-1.1	Constrained subsystems	125
8-1.2	Callback and invariants	128
8-1.2.1	<i>Propagation of transaction status</i>	129
8-1.3	Constraints & contracts summary	129
8-2	Aliasing example	130
8-2.1	SortedList and SortedIdList	130
8-2.2	Method implementations	131
8-2.3	Using the lists	131
8-2.4	Aliasing problem summary	133
8-3	Effects calculus	133
8-3.1	Definition of concepts	134
8-3.1.1	<i>Fields</i>	134
8-3.1.2	<i>Frames</i>	134
8-3.1.3	<i>Demesnes</i>	135

8-3.1.4	<i>Frames in signatures</i>	137
8-3.1.5	<i>Separation</i>	137
8-3.2	<i>Aliasing example revisited</i>	138
8-3.2.1	<i>Type specification with framing</i>	138
8-3.2.2	<i>Sketch proof</i>	138
8-3.2.3	<i>An invariance proof</i>	139
8-3.2.4	<i>SortedIdList used with copied arguments</i>	140
8-3.2.5	<i>Commentary</i>	141
8-3.3	<i>Inference of frame of an expression</i>	141
8-3.3.1	<i>Constants</i>	142
8-3.3.2	<i>Variables</i>	142
8-3.3.3	<i>Operations</i>	142
8-3.3.4	<i>Compound statements</i>	143
8-3.3.5	<i>Assignment</i>	144
8-3.3.6	<i>Substitution</i>	144
8-3.3.7	<i>Copying</i>	145
8-3.3.8	<i>Mechanical inference of frames</i>	145
8-3.4	<i>Monotonicity and frames</i>	145
8-3.5	<i>Type Isolation</i>	147
8-3.6	<i>Reification of subsystems</i>	147
8-3.7	<i>Effects calculus summary</i>	148
8-4	<i>Naming previous states</i>	149
8-4.1	<i>Barred expressions</i>	149
8-4.2	<i>Metavariables and code</i>	150
8-4.2.1	<i>Potential inconsistencies in interpretation of metavariables</i>	150
8-4.2.2	<i>Metavariables are constant pointers only</i>	151
8-4.2.3	<i>Metavariables refer to ghost copies</i>	151
8-4.3	<i>Bars and metavariables summary</i>	152
8-5	<i>Projection to supertype, and equality</i>	152
8-5.1	<i>Equality and subtypes</i>	152
8-5.2	<i>Projection operator</i>	152
8-5.2.1	<i>Definition of equality for a type</i>	153
8-5.3	<i>Substitution</i>	153
8-5.4	<i>Application of rules</i>	154
8-5.5	<i>Comparison with conventional LPF</i>	154
8-5.6	<i>Summary of typed equality</i>	154
8-6	<i>Summary</i>	155
9-1	<i>Overview</i>	156
9-2	<i>Assessment</i>	156
9-3	<i>Future work</i>	157
10-1	<i>Fresco development language FST</i>	165
10-1.1	<i>Expressions</i>	165
10-1.1.1	<i>Conventional expressions</i>	165
10-1.1.2	<i>Smalltalk-style expressions</i>	166
10-1.1.3	<i>Assignment</i>	166
10-1.1.4	<i>Creation functions</i>	166
10-1.1.5	<i>Special notations</i>	166
10-1.2	<i>Specifications</i>	166
10-1.2.1	<i>Pre/post</i>	167
10-1.2.2	<i>Code-Invariant</i>	167
10-1.2.3	<i>Composition</i>	167
10-1.2.4	<i>Inline justification</i>	167
10-1.3	<i>Code</i>	167
10-1.3.1	<i>Control expressions</i>	168
10-1.4	<i>Metavariables</i>	168
10-1.4.1	<i>Qualified and modified metavariables</i>	169
10-1.5	<i>Theorems</i>	169

10-1.5.1	<i>Proofs</i>	169
10-1.6	Types and classes	170
10-2	<i>Fresco kernel types</i>	170
10-2.1	Sets	170
10-2.2	Lists	171
10-2.3	Maps	172
10-3	Kernel proof rules	172
10-3.1	Comparison with conventional LPF	172
10-3.2	Projection to a type	173
10-3.3	Opspecs	173
10-3.4	Types	174
10-3.5	Effects	174
10-3.6	Frames	174
10-3.7	Barred variables	176
10-3.8	Projection	176

Abstract

This work investigates the application of formal methods to object-oriented programming. The desirable features of such a synthesis are defined, and the problems of achieving it – such as aliasing – are investigated. Some solutions are proposed.

The work focuses on the design of 'Fresco', which is a development environment for the construction of object-oriented software with formally-specified and proven components. Software is developed interactively (after the style of Smalltalk) together with proofs of conformance to specification.

Specifications may be attached to abstract and concrete classes, and a strict notion of subtyping is used to achieve polymorphism. Generic types are also supported. Types are interpreted as sets of possible histories of objects.

Software components are generated and transmitted in 'capsules'. A capsule may contain specifications and/or implementations of new software, or modifications of existing software. A system is composed of a sequence of capsules, which have an acyclic dependency graph. While capsules may be brought together in different configurations in different systems, Fresco can ensure that each capsule performs as specified, prohibiting configurations which could fail because of interference between capsules.

Exclusion

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Thank you

Cliff Jones.

John Fitzgerald, Mark van Harmelen, Peter Lindsay, Mario Wolczko.

The Department of Computer Science, Manchester University.

All my friends.

The Author

Eur. Ing. Alan Wills gained a B.Sc. in Physics at the University of Liverpool in 1974, and an M.Sc. in Computer Science at the University of Manchester in 1979. He has worked on communications systems and software development support tools in a variety of industrial and academic projects, and now provides consultancy and training in object-oriented software engineering.

1 Object-orientation and formal methods

1-1 Objectives: formal methods into OOP

1-1.1 Object orientation is a Good Thing

1-1.1.1 Re-use

The advertised benefits of object-oriented programming include re-use in various forms [Goldberg 83, Cox 86, Meyer 88]. These include:

- Parametric and inclusion polymorphism serve to minimize the amount of code written within one system — in other languages, similar abstractions need different pieces of code.
- Inheritance allows the common parts between different components to be factored into one piece of code.
- Strong encapsulation and narrow interfaces make OO components portable between systems and organizations.

In an object-oriented culture, there is less code-writing per product. The code which is written for any product should cover more cases, be more adaptable to variants of the same product, and other products dealing in the same domain. Whilst traditional component libraries have covered very restricted fields, it now makes sense for an organization to make the development and maintenance of a component library a key part of their planning [HW 90]. It follows that a market in such components should be expected to develop: free exchanges already exist — e.g. [WW90].

1-1.1.2 Responsiveness to change

Partly for the same reasons, OOP can also produce very flexible code. Changes in requirements are an inevitable feature of system design, and one which previous software engineering methods have not tackled well. Change can be handled well in an OO program because

- A polymorphic component P is one which is designed to work in conjunction with many other component C_i , provided each has a set of characteristics defined in a specification S . C_i need not exist when P is designed; a new C_i may be added to the system without altering P , provided the new component conforms to S .
- Because inheritance is used to factor the common features of many components, there is often one place in a program which needs to be modified, where in other methods there would be several. This tends to ensure that the alterations to a system remain uniform, so that the coherence of the system

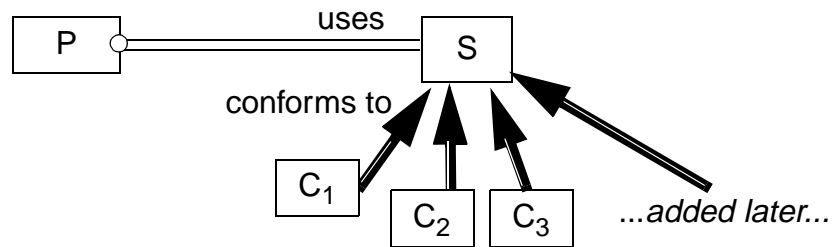


Fig. 1. Polymorphism

is better preserved over its lifetime.

- Strong encapsulation means that each component represents a clearly-defined concept, generally reflecting the users' perceptions of the real-world domain which forms the context of the system. Changes in requirements tend to happen within clearly-defined areas in the domain, whilst its overall structure remains relatively constant. It is therefore easy to identify those parts of the code which need changing, and these tend to be localized.

A well-managed object-oriented software house should therefore be able to produce many variants of its products, and be able to cope rapidly with changes in the demands of the market and individual customers.

1-1.2 Why it hasn't happened yet (much)

1-1.2.1 Re-use between systems

A look at a public OO library (such as `goodies-lib@cs.man.ac.uk`) reveals that its dependency graph is rather flat: there are very few software components which build upon others. This is partly because it is no-one's responsibility to see that it is well-structured, and contributions tend to cover several poorly-separated ideas in one go.

But there is also the usual 'not-invented-here' syndrome: to some extent a distrust of the work of others, and to some extent a feeling that it is easier to invent the components you need for this job, than to have to search through and fully understand someone else's code and do your design so as to use them. Code created for other projects is, in any case, likely to depend on undocumented assumptions which may be invalid outside their original environment. Picking up code designed for another project by people you don't know is very different from the informal to-and-fro between the writers of separate modules in a single project.

Certainly no builder of a safety- or money-critical system would be wise, currently, to use a collection of components from who-knows-where, whose designers may be difficult to contact and have no further interest in them.

1-1.2.2 Change and re-use within a product or product line

The effectiveness of polymorphism depends on the existence, precision, and utilization of the specification **S** (in the above model). In inclusion polymorphism, **S** is typically represented by an abstract class; but a class expressed in programming language is not capable of representing a specification, and so of course, the class can only stand in place of the specification, which has to be documented separately. In current practice, this very rarely happens: and in consequence, very careful checking and re-testing is usually required if a new C_i is to be introduced.

Indeed, in some forms of polymorphism, S has no representative at all: for example, the characteristics required of the items in any Smalltalk `SortedCollection` — those of a totally ordered set — are specified only informally in the documentation of that class.

It is very difficult to test polymorphic code: the C_i components with which it is supposed to work may not all have been designed yet.

Perhaps the biggest obstacle is that many practitioners are not aware of these ideals.

1-1.3 Formal Methods are the Solution

A well-written formal specification abstracts and separates the different features of interest to the client. This makes it easier to read than the code. The separation of features makes it easier to reason about the correctness of client-code.

The writing of a formal specification also tends to clarify the intentions of the designer, leading to components which are conceptually more succinct and therefore more likely to be useful in contexts other than those for which they were designed.

A formal proof of a component is an assurance that it is reliable. Whilst it can be tedious to generate in the first place, a documented proof which comes with a redistributed component can be checked entirely mechanically.

S is usually not a formal specification in current practice, but formal proofs can make it unnecessary to re-test P for each new C_i : when P is written, it is proven to work with anything which conforms to S ; and when each C_i appears, it is proven to conform to S .

1-2 The focus of this work: Fresco

Given these potential benefits, this work investigates the application of formal methods to object-oriented programming. As a focus, a tool is being designed which will embody the ideas: Fresco is to be a development environment for the construction of object-oriented software with formally-specified and proven components.

Fresco programs (in its currently envisaged form) are written in a version of Smalltalk-80, differing from that language only in its concrete syntax. Specifications are written in an extension of the same language. Proofs will be generated with the help of an adaptation of the Mural proof assistant, which is the result of an earlier Manchester University project [Mural].

As in Smalltalk, the intention is that software will be developed interactively, and with much use of modules interchanged between Fresco programmers. The unit of design and interchange is the ‘capsule’, which may contain specifications and/or code for new software, or for modifying existing software. Every Fresco system is a composition of capsules, beginning with a kernel of basic building blocks. Whilst every capsule depends on a specific set of others, different configurations of capsules may compose different systems. Fresco can ensure that each capsule performs as its author intended, and prohibits configurations whose capsules would conflict.

The objective is to maximize the benefits of object-oriented programming, of re-use and flexibility.

1-2.1 The state and history of Fresco

It must be emphasized that, despite the tendency in the remainder of this thesis to use the present tense to describe it — purely for reasons of style — Fresco is not yet a fully extant system. Parts of the mechanics of capsule management have been prototyped; and Mural, the precursor to Fresco's proof system, does exist. The rest is faith and hypothesis.

Mural is an interactive theorem prover's assistant, built as the Manchester University contribution to the IPSE2.5 project [Ipse]. Built in Smalltalk, it represents a considerable advance on the interactive style of preceding proof tools, and implements a very flexible inference system.

An objective of Mural was to be a generic tool, which could be used for pure mathematics, or — with a suitable front end — for program verification or any other application. This flexibility was demonstrated to some extent, though with a rather uncomfortable transition between the front-end and mathematical parts. The intention for Fresco is to take over many of the ideas in a more suitable implementation. (Re-use is only worth it for the best bits!)

Smalltalk was chosen as an implementation language for Fresco and for Mural because of its ready flexibility and prototyping strengths.

Smalltalk is also the basis of the target language of the initial version of Fresco, partly because that's clearly easier, but also because an aim in Fresco is to demonstrate that formal methods and evolutionary programming are compatible.

1-3 The wider context: OO software engineering

1-3.1 Effects of OOP on the lifecycle

OO programming allows new systems to be built very quickly, especially if they are similar to previous ones, or in the same domain. One consequence is that trial systems can be prototyped, with feedback from users at an early stage of analysis. Much more feedback from users can be obtained and applied between delivery of successive versions of a system. Any pretence of a waterfall model of software development can no longer be supported: product planning and project management are clearly deeply affected (which is good for consultants).

Perhaps one of the benefits of the OO revolution is that the lure of its potential benefits is making organizations think more seriously about methodology: since it is clear that you do not get the benefits unless you apply the ideas properly. Hence the growth in the fields of Object Oriented Analysis and Design [Coad, Rumbaugh, Jacobson, Booch].

Hopefully, the same effect will apply to the adoption of formal methods. The notion of abstract class gives a clear focus and motivation for abstract behavioural descriptions, and also an obvious place to hang the specifications (rather than just in the filing cabinet).

It is worth noting that the re-use of specified components makes it far more cost-effective to specify and prove a component than with traditional methods.

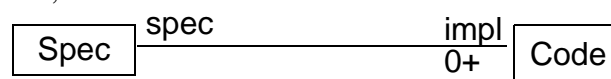
1-3.2 OO Analysis and Design

The topic is still in its infancy, and there are many ideas and confusions, and many notations and tools to support them. “Object orientation”, perhaps less of an automatic grant-attractor for researchers these days, is still a money-maker for textbook writers, commercial trainers, and consultants, and the tag is often attached to imperfect adaptations of old material. Nevertheless, some good common ground can be discerned.

The starting point is generally the assertion that the domain within which the proposed system will work can itself be modelled as an OO system. Such a model is a useful tool in analysis, and can be the base upon which another model, that of the required system, can be built. The requirements are then reified into a set of executable classes. (And it has to be said that many authors tend to confuse analysis with design, introducing internal structure during analysis which they assert should carry through to the final design.)

The notations used are similar to those of entity-relation models, using diagrams together with supporting text. Boxes represent classes, arrows subtyping or subclassing (which are generally confused) and other lines represent references from instances of one class to those of another.

Rumbaugh and others have emphasized the utility of these relations as abstractions at early stages of design. For example, in a high-level design of a program development support tool,



[Wills 91b] extends such a notation to include formal specifications in the form of invariants and pre/postconditions. It is an ambition for Fresco that the designer should be able to browse designs in diagrammatic form (though this is not tackled here). It is the author’s belief that boxes with lines between them have considerable power to sweeten the pill of formal specification and verification.

1-3.3 OO formal methods

The emphasis of this work is to apply formal specification and proof to object-oriented program components. Others have come from the opposite direction, and are interested in applying OO principles to specifications, to bring the benefits of modularisation. In particular, many OO variants of Z have been described. Not all of these are especially suitable for describing OO program components: the separate modules are not necessarily separately implementable, and the modular partitioning of the implementation may be quite different (Fig. 2.)

These efforts may be seen as part of the general trend to produce OO variants of Pascal, Cobol, Ada, and so on. Where the language already has a modularising construct, the result is often uncomfortable. While there is some argument for making a new, backward-compatible version of an old programming language, this seems inapplicable to specification languages: since so few specifications have been writ-

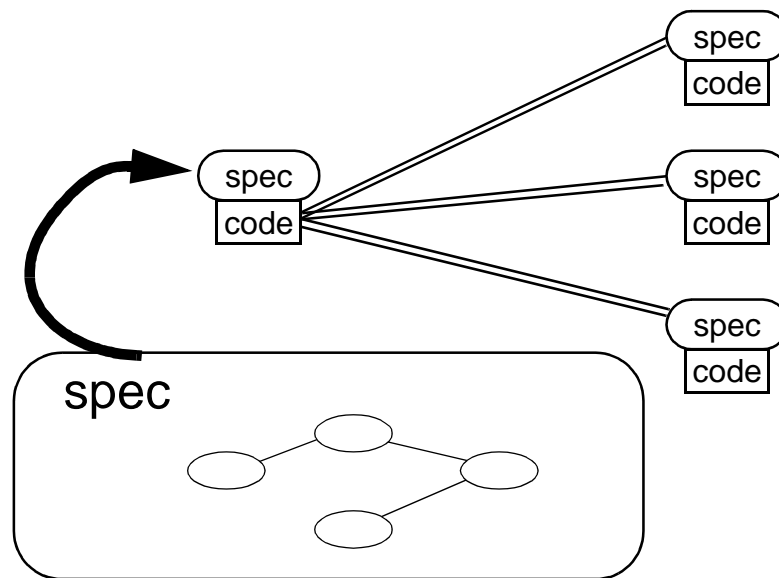


Fig. 2. Different internal structures for spec and implementation

ten, and since even fewer people understand them, there is really very little investment to be preserved.

1-4 Scope of this work

The purpose here is to investigate and demonstrate the application of formal methods to object-oriented programming, retaining and enhancing the benefits of OO programming and design. The problems will be elucidated, and some solutions proposed and tried out.

The preceding text should have given some flavour of the author's keenness to integrate formality with OOP in such a way as to preserve all the benefits. An evolutionary style of programming should still be possible, and the strong encapsulation of OOP must be promoted in the specification method. The method should be reasonably practicable on an everyday basis. Minimal constraints should be imposed on the order of construction of proofs or specifications, just as for the software itself.

It cannot be hoped for that it will be possible to complete proofs in full detail: we will follow the spirit of 'rigorous' proof [Jones80], in which some parts of a proof can be left as informal arguments until challenged in review.

Whilst Smalltalk is fixed upon as a target programming language, the principles are applicable to other languages. It is not the intention to generate a formal system which will cope with anything you can program in Smalltalk: rather, the programmer should stick to those constructs which can be dealt with by the formal system. This makes it easier to generate the basic semantic model for Smalltalk, and follows in the tradition of the formalists' approach to the *goto*.

The main body of this work is on the design and semantics of the Fresco language and proof system. The system can be described in layers, starting with the basics of the proof system (Chapter 4), and continuing with the specification, refinement and verification of three levels of description: statements (Ch. 5), types (Ch. 6), and cap-

sules (Fresco's module of design effort) (Ch.7). These provide a firm basis on which the most difficult issues in OO/FM can be discussed, and Fresco's contribution to the field can be measured by its utility as a vehicle for formulating questions and experimenting with solutions in these areas.

Certain areas are excluded from consideration: concurrent programming; programming languages with models far removed from those of Smalltalk or Eiffel (such as CLOS and Self).

The scope of the work is quite broad. The intention is to map out the area with the benefit of a formal approach, rather than to examine any one detail in the fullest possible rigour — for many aspects of what is covered here, that will be a matter for a future thesis.

1-5 Structure of this thesis

This objective of this chapter has been to motivate and set the scope of the work, and put it in a wider context. The benefits to be expected of object-orientation, formal methods, and their synthesis have been reviewed: principally, fast and reliable construction through re-use, and responsiveness to changes of requirements.

The next chapter gives an overview of Fresco and the way in which it should be used, so as to make plain the objectives of the more detailed descriptions in the subsequent chapters. Chapter 3 then surveys the scene, assessing the state and direction of the work of others, defining the issues of interest, and establishing a vocabulary of terms and concepts best suited for the appreciation of what follows.

Chapters 4–7 describe the various features of the Fresco language individually; but a summary is provided in the Appendix. A number of issues particularly concerned with aliasing and encapsulation are separated out into Chapter 8, which also provides the opportunity for some larger examples.

Finally, Chapter 9 assesses how far Fresco meets its aims and contributes to the field, and considers the directions future work should take.

A bibliography and a summary of language, kernel types, and proof rules is appended.

2 OO Software Engineering with Fresco

Fresco is a scheme of software development which enables programmers to interchange well-defined and guaranteed software components called ‘capsules’. Programming proceeds according to the precepts of evolutionary software development, underpinned by the precision and reliability afforded by formal methods.

This view of software development and exchange motivates the Fresco notions, central to this thesis, of types as theories of object behaviour, and subtypes as theory-extensions.

Later chapters present the Fresco type system in detail. The purpose of this chapter is to give an informal overview of the material, and of the development method Fresco is intended to promote.

2-1 The formalised goodie

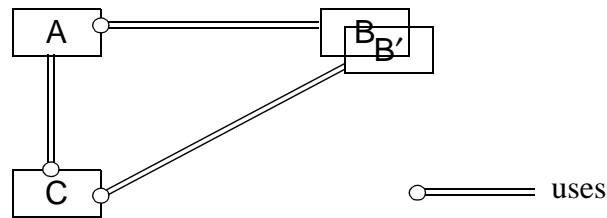
2-1.1 Units of development effort

Object-oriented programming makes possible a culture in which systems are rapidly built from widely-distributed and adapted components. Developers can build and sell or exchange components as well as complete systems; and can treat their software libraries as capital resources which they augment every time they write a new component.

The units of distribution in the successful Smalltalk re-use culture are not classes, nor even groups of classes. A look at any ‘goodies’ library shows them to be mixtures of new classes, new methods for existing classes, and new implementations of existing classes and methods. (In Smalltalk, classes and methods are updated and compiled dynamically into the running system.) In `goodies-lib@cs.man.ac.uk`, 73% of the files modify existing classes, and 44% define no new classes. Each programmer’s efforts build upon those of one or more predecessors by improvement and extension. Fresco formalises this: software is packaged in *capsules*, which define extensions to classes, redefinitions of old methods, as well as new classes; and each capsule includes specifications of the classes which result.

If this notion of ‘deltas’ as units of designer-effort seems a little strange, consider this scenario. Class A uses class B extensively, and sometimes passes B-instances back to its own clients. I design class C, which uses A; but C needs B to perform some extra function, used whenever B-instances are passed back from A. Ideally, I should design a B’ which inherits from B. But then I have to design an A’ which is all the same as A, except that it calls upon B’ instead of B. If A has been designed with sufficient foresight, then this will be easy; but more likely, it will be a pain!

What I really want to do is just to add the extra function to B — more economical and less error-prone.



More generally, many of the real-life examples of redefinition are connected in some way with improving the inheritability of a class, or broadening its functionality. Others are concerned with improving the performance (so that all clients get the benefit, not just those who know about a new subclass); and most of the rest, with enhancing user-interaction without altering the procedural interface.

[Szyperski 92] gives the example of a statistics package to be added to **RealNumbers**: if there are subclasses of **RealNumbers**, they should inherit the new functions too. A separate **RealNumbersWithStats** class would not achieve this — unless the existing subclasses were altered to inherit from it.

In the other direction, the interdependence of frameworks of classes has been much discussed [e.g. JF88]. If a diagram of dependencies between classes (or any other units of design) is drawn, it makes no sense to attempt to transport separately any units which belong to a loop. (Fresco capsules therefore form an acyclic dependency graph — if two would form a loop, they ought to be in the same capsule.)

Functional units and their hierarchies are good for integrating into one structure all the diverse functions which can be created by a single designer [team] while the hierarchy remains under that designer's control; additional requirements may trigger a restructuring. But when we consider design effectively undertaken by many designers between which there is only a one-way flow of information, then the transmissible units of design-effort must be not functional units, but changes to their definitions. But it is important that when a system imports such deltas from diverse sources, they shouldn't invalidate each other: each should be able to change the implementation of what went before, and should be able to enrich any part of the system's behaviour, but not to alter (or delete!) the functional specification of existing behaviour, which other parts might depend on.

2-1.2 System composition

Fresco supports the specification and rigorous development of software capsules. A capsule contains code, specifications, and proofs, and systems are built by composing capsules. All development work is done within the context of some capsule, and systems are built by importing capsules and developing new ones. Every definition in a system is part of some capsule. The mechanism has the potential to guarantee that each capsule functions as its author intended, without interference from others: although the functions a capsule provides can subsequently be extended or improved, the properties its clients rely upon will never be invalidated.

Part of the scheme's operation depends on restricting the ability of a capsule to override existing definitions, to those belonging to capsules on which it has a documented dependence: this by itself can help to reduce the likelihood of clashes. Whilst the full benefit depends on the (admittedly theoretical) employment of fully

formal proofs, greater reliability is nevertheless obtained by using specifications with more or less ‘rigorous’ proofs. Even where proofs are completely informal, the system highlights correspondences between specification and code which should be rechecked whenever anything is altered.

The elements of a capsule may be created in any order: code first or specifications first. Fresco generates appropriate proof obligations wherever the consistency of the code and specifications cannot be verified automatically. Before the capsule may be exported for distribution to other designers, Fresco performs a ‘certification check’, that all the proofs have been completed, and are consistent with the definitions (see Figure 1). A complementary ‘incorporation check’ ensures that imported capsules (i) only alter the code of capsules they claim to know about and (ii) have internally consistent proofs (even if partly informal ones) and hence, hopefully, code that conforms to their specifications.

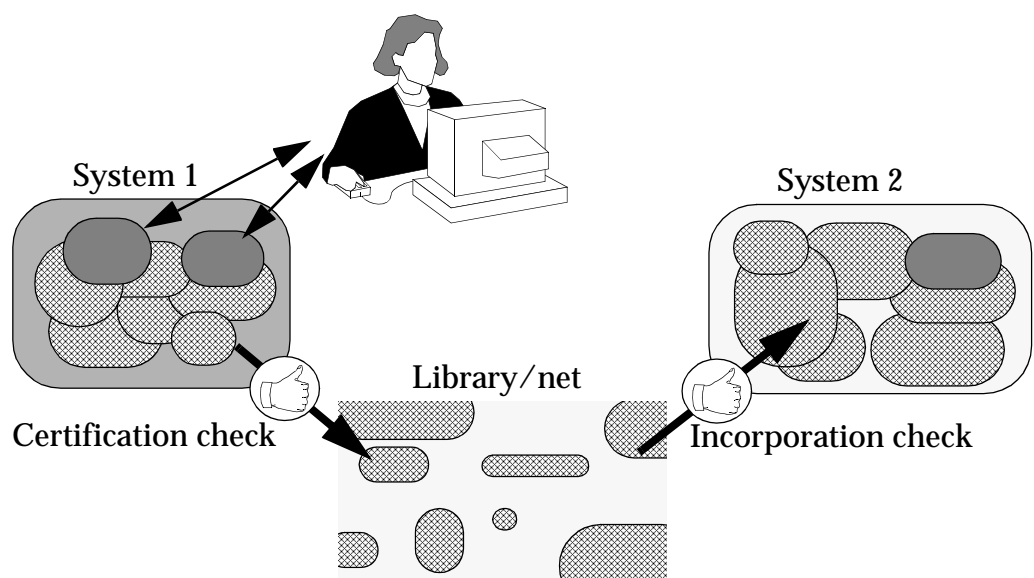
It is essential that an extended version of any class should behave the same to previous clients as its predecessor did: in Fresco terms, it should conform to the same type. The next section introduces Fresco’s type system and outlines how it fits into the proof system. We will ultimately come back to capsules and explain their composition into systems, in the light of the type system.

2-2 Types and classes in Fresco

In traditional formal development methods, the documents representing top-level specification, code, and intermediate levels of refinement are usually separate, with some trace information and proofs interrelating them. Fresco integrates all these into one database, in a unified syntax, accessible through a single browsing/editing system.

The basic unit of specification is the ‘type’, and of implementation, the ‘class’. The class is unchanged from Smalltalk (except for the concrete syntax). An object is created as an *instance* of only one class, but may be a *member* of many types. Types

Fig. 3. Fresco systems are compositions of capsules

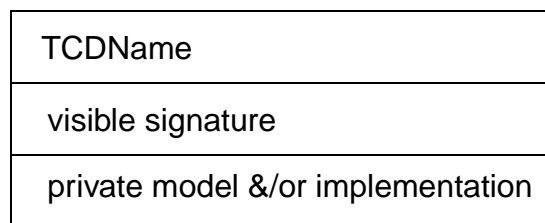


are used to document constraints on variables or parameters; and to document required and provided behaviour of software components.

An object's type describes its behaviour, visible as its response to messages, and the constraints which apply to messages it may be sent. (Notice that this is a far stronger notion of type than in most programming languages, where type membership is about which messages are understood, but not what they do.) Each type is specified in model-oriented style, with pre/postconditions written in terms of model variables; which may, but need not, correspond to any actual variables in any class which implements the type.

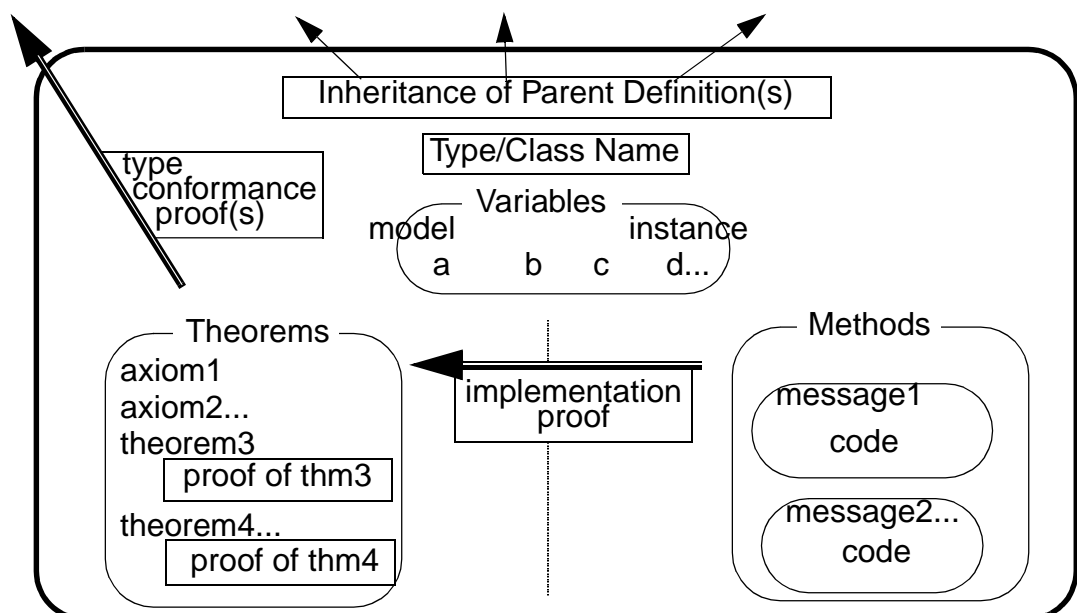
Despite the distinction, type and class definitions are interwoven, for convenience, into a single all-purpose piece of syntax, the type/class definition (TCD). Fresco has TCDs instead of classes, and they are realised by adding opspecs and invariants to Smalltalk's classes. Some languages, such as POOL and Abel, separate the syntaxes of types and classes; Fresco permits partial or complete class definitions to be integrated with types where the designer considers this appropriate: in that case, the assertions apply to the real instance variables.

The (planned) Fresco browser will provide a Smalltalk-like interface to the hierarchy of TCDs, and a diagrammatic representation similar to that of [Rumbaugh]. On paper, boxes of this form are used:



The model section may be missing; the signature section may also be missing, if the tcd name by itself is to be used in a diagram of class relationships. So that a specification can be spread over several pages of a document, the same tcd may be defined in several different boxes: any implementation must conform to the speci-

Fig. 4. Components of a Fresco type/class definition



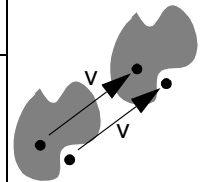
cations in all the boxes — and it is up to the designer not to specify conflicting requirements. The specification section defines behaviour which clients may rely upon (whether or not they have seen every box referring to this class); the implementation section is of no concern to clients.

2-3 Specification in Fresco

2-3.1 A type

Shape is the type of objects representing mutable two-dimensional shapes. Every such instance has at least two operations which can be performed on it: you can ask whether it **contains** a given two-dimensional **Point**, and you can **move** the whole shape by some **Vector**, which translates the set of **Points** it contains.

Shape	
fn contains	$\in (\text{Point}) \text{ Bool}$
op move	$\in (\text{Vector})$
mv-def: $v, p \cdot p \in \text{Point} \vdash$ $\{ v \in \text{Vector} :- (\text{self.contains}(p) \Leftrightarrow \text{self.contains}(p+v)) \} \text{ move}(v)$	



The description lists the signatures of operations which clients can access; contains is declared with **fn**, meaning it does not alter the state of anything. (A more detailed method of defining the scope of the effects of a method is described in §8.) The axiom labelled mv-def details the effect of **move** on the state of the object. The axiom is in the form of an ‘opspec’:

label: variables · { precondition :- postcondition } operation(parameters)

For any match between the variables and specific objects, provided that the client ensures that the precondition is met, then the operation will terminate properly and the postcondition will be satisfied. Within the postcondition, barred items refer to the state prior to the operation. In this case, unusually, one operation is specified in terms of another. More than one opspec may apply to one operation.

The signatures of the operations are interpreted as ‘formation’ axioms — for example

$$p \cdot p \in \text{Point} \vdash \text{contains}(p) \in \text{Bool}$$

Other theorems may be derived from the axioms of a type, and the entire set of derivable theorems is called the type’s theory.

An opspec may be extracted systematically from the context of its type T :

$$v, s \cdot \{ s \in \text{Shape} \wedge v \in \text{Vector} :- \forall p \cdot p \in \text{Point} \Rightarrow (\bar{s}.\text{contains}(p) \Leftrightarrow s.\text{contains}(p+v)) \} s.\text{move}(v)$$

In this form, the opspec can be used as a theorem in the proofs of clients which use **Shape**.

The type-membership assertion $x \in T$ means that x conforms to all the axioms (and therefore all the derivable theorems) of T ; many other types may also contain x . The type-description carries no implication that other operations may not be applicable

to members of **Shape**: merely that we do not know anything of how they will behave. The vacuous specification

$$\{ \text{false} :- \text{true} \} \text{ op}$$

is implicitly true of every operation (including those we do not yet know about).

2-3.1.1 Role of types in code

Types are used in code just as in ordinary programming languages; except that it is possible to distinguish types which would not be distinct in an ordinary signature-checking language.

Fresco code may be developed with specification-statements in the style developed by Morgan, Robinson, and others [Morgan]; the specifications may contain typing assertions which describe the properties of a variable's contents at that point in the code. A type is used as an abbreviation for a set of properties; as such, it can be seen as a convenience: the more basic tool is the ability to assert properties of an object. All typing assertions, whether dispersed in the code, or at variable and parameter declarations, are equivalent to a set of assertions about the object's response to applied operations.

2-3.1.2 Types and subtypes

A Fresco type is the set of all objects each of which conforms throughout its life to a set of theorems about its observable behaviour. The type defines a set of possible histories of operations on the object; an ops spec defines the states in which a given transition may occur, and the relation between the states at each end of a transition.

If H_{ST} is the set of possible histories conforming to the type ST , and H_T to T , then if $H_{ST} \subseteq H_T$, we say that ST is a subtype of T , written $ST \subseteq T$.

A client may be designed which will work for all subtypes of **Shape**, without knowing anything more than **Shape**'s theorems. (For example, a handler which keeps a list of **Shapes** and permits users to move them around the screen.)

2-3.1.3 Type extension

FourSides ::+ Shape	
op setp1 ∈	(Point)
op setp2 ∈	(Point)
op setp3 ∈	(Point)
op setp4 ∈	(Point)
axfsc:	$p \cdot \{ \text{:- } \uparrow = p.\text{withinLoop}(\langle p1, p2, p3, p4 \rangle) \} \text{ contains}(p)$
axfs1:	$np \cdot \{ \text{nonIntersectingLoop}(\langle np, p2, p3, p4 \rangle) \text{ :- } p1=np \} \text{ setp1}(np)$
axfs2:	$np \cdot \{ \text{nonIntersectingLoop}(\langle p1, np, p3, p4 \rangle) \text{ :- } p2=np \} \text{ setp2}(np)$
axfs3:	$np \cdot \{ \text{nonIntersectingLoop}(\langle p1, p2, np, p4 \rangle) \text{ :- } p3=np \} \text{ setp3}(np)$
axfs4:	$np \cdot \{ \text{nonIntersectingLoop}(\langle p1, p2, p3, np \rangle) \text{ :- } p4=np \} \text{ setp4}(np)$
var p1 ∈	Point
var p2 ∈	Point
var p3 ∈	Point
var p4 ∈	Point

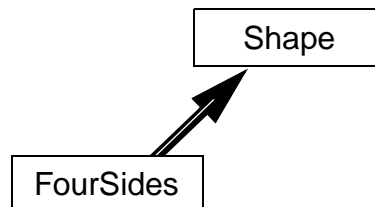
// disallowed:



FourSides is the type of objects representing shapes bounded by four straight edges. It is defined as an extension (‘::+’) of **Shape**: every axiom is inherited. Any theorem derived from the axioms is therefore also true of the derived type. Any client only interested in the movability of all members of **Shape** may therefore make the same assumptions about members of **FourSides**. Fresco type membership is defined by conformance to behavioural theorems, and so for any types **A**, **B**

$$A ::+ B \vdash A \subseteq B$$

Equivalent to the ‘::+’ notation is the thick arrow:



2-3.1.4 Model-oriented specification

FourSides has a model, the four variables p_i , in terms of their effects on which the operations are defined. There is an invariant on the model, which ought to be true before and after execution of every operation. Its maintenance is only the responsibility of an implementation, and not of the clients — they just have to conform to the stated preconditions of the operations.

The model is hidden from implementations of clients, but clients may use it in their own specifications and reasoning. See §3-4.3.2 – *Applicability of encapsulation to specifications* (p.44).

Since it is not our business here to learn geometry, let’s assume a predicate **nonIntersectingLoop** which ensures that the boundaries do not cross; and a predicate **inPoint**, **withinLoop**, which tells whether a point is within the bounds defined by a given tuple of points.

The special variable \uparrow is used within a postcondition to signify the object returned by the operation.

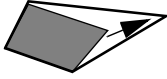
2-3.1.5 Preconditions and invariants

At this stage, we know that `setpi` sets its corresponding vertex, but nothing else about what it does or doesn't do. Notice that their preconditions duplicate the invariant. If they did not, an implementation would be obliged to achieve the postcondition for any prior state; since it is impossible to guarantee that and be sure of satisfying the invariant as well, the type would be unimplementable. (See §3-4.5 — p.47.)

2-3.1.6 Inheritance and subtyping

Whilst the type of mutable rectangles is not a subtype of mutable quadrilaterals, `FourSides` does describe the static and dynamic properties which they have in common.

Quadrilateral ::+ FourSides	
axq1:	$np \cdot \{ :- p2=\overline{p2} \wedge p3=\overline{p3} \wedge p4=\overline{p4} \} \text{ setp1}(np) // \text{ other points are fixed}$
axq2:	$np \cdot \{ :- p1=\overline{p1} \wedge p3=\overline{p3} \wedge p4=\overline{p4} \} \text{ setp2}(np)$
axq3:	$np \cdot \{ :- p1=\overline{p1} \wedge p2=\overline{p2} \wedge p4=\overline{p4} \} \text{ setp3}(np)$
axq4:	$np \cdot \{ :- p1=\overline{p1} \wedge p2=\overline{p2} \wedge p3=\overline{p3} \} \text{ setp4}(np)$



No new operations or model variables are introduced here, but new opspecs apply to the existing operations. Implementors must prove that their implementations meet all the applicable axioms (e.g. code for `setp1` must meet both `axfs1` and `axq1`), and clients may assume any or all of the axioms they know about (so if I know I've got a `FourSides` but don't know what kind, then at least I can rely on `axfsi`).

This is reflected in an interpretation rule for compositions of opspecs:


$$\frac{\begin{array}{l} \{pre1:-post1\} \text{ code} \\ \{pre2:-post2\} \text{ code} \end{array}}{\{pre1 \vee pre2 :- (pre1 \Rightarrow post1) \wedge (pre2 \Rightarrow post2) \} \text{ code}}$$

So subtypes always have weaker preconditions and stronger postconditions. This is an effect of the Fresco interpretation of inheritance, not something that the designer has to ensure. On the other hand, there is no guarantee that it will be possible to write code which satisfies the combined specifications: that is, the type may be empty.

A type may extend more than one supertype (again with no guarantee of implementability of the result: that is up to the specifier). Model variables and operations are identified by name, unless explicitly renamed.

2-3.1.7 Strengthening invariants

HVRectangle is a different extension describing rectangles with sides parallel to the axes. Repositioning any corner leaves the opposite one unmoved, but the other two adjust accordingly.

HVRectangle ::+ FourSides	
$\begin{aligned} & np \cdot \{ :- p1=np \} \text{setp1}(np) && // \text{ now no constraint on pre} \\ & np \cdot \{ :- p2=np \} \text{setp2}(np) \\ & np \cdot \{ :- p3=np \} \text{setp3}(np) \\ & np \cdot \{ :- p4=np \} \text{setp4}(np) \\ & np \cdot \{ :- p3=\overline{p3} \} \text{setp1}(np) && // \text{ opposite point is fixed} \\ & np \cdot \{ :- p4=\overline{p4} \} \text{setp2}(np) \\ & np \cdot \{ :- p1=\overline{p1} \} \text{setp3}(np) \\ & np \cdot \{ :- p2=\overline{p2} \} \text{setp4}(np) \end{aligned}$	

Invariants are considered to conjoin with all pre and postconditions. Implementors of HVRectangle should observe both *invhvr* and *invfs*: whether the result is implementable depends on the effect on the postconditions. In the case of a nondeterministic specification, strengthening an invariant is OK if it only cuts down some of the implementor's options without cutting them out altogether. A new invariant may also impose constraints on new model variables.

It is not possible to weaken the invariant in an inheriting type: but few examples where this would be useful and good style have been found.

2-3.1.8 Operation extension

Notice that the domain of an operation can be extended by adding a new axiom. For example an existing operation

$$\{ r \geq 0 \wedge r \in \text{Real} :- \uparrow \in \text{Real} \wedge \uparrow * \uparrow = r \} \text{sqrt}(r)$$

can be extended to

$$\{ r < 0 \wedge r \in \text{Real} :- \uparrow \in \text{Complex} \wedge \uparrow * \uparrow = r \} \text{sqrt}(r)$$

Old clients won't know the difference. A useful more general theorem can be inferred:

$$\{ r \in \text{Real} :- \uparrow * \uparrow = r \} \text{sqrt}(r)$$

2-3.2 Generic types

Generic types may be defined with type paramters. In this example, the arguments are restricted to types conforming to a subtype of **TotalOrdering**:

SortedList of: T ::+ (TotalOrdering of: T)	
op add:	(T)
fn get:	(Nat) T
...	
s :	List(T)
$i, j \cdot i < j \wedge j < \text{self.len} \Rightarrow s[i] \leq s[j]$	
...	

An instance of this, for example the type of sorted lists of integers, is written

SortedList of: Integer

and in the case of specific types, the designer may include theorems like

$T \cdot T \text{ sortedList} = (\text{SortedList of: } T)$

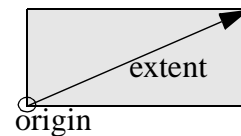
so that clients may write

Integer sortedList

2-3.3 Model refinement

HVRectangle1 is not defined as an extension of **HVRectangle**, but is believed to represent a subtype of it. Whilst the operations available to clients are the same, the model variables are different: while **HVRectangle** is defined with the four points **p1..p4**, **HVRectangle1** is defined with one corner and a vector representing a diagonal. It will be necessary to prove the subtype relationship.

HVRectangle1	
op contains \in	(Point) Bool
op move \in	(Vector)
op setp1 \in	(Point)
op setp2 \in	(Point)
op setp3 \in	(Point)
op setp4 \in	(Point)
$p \cdot \{ p \in \text{Point} :- \uparrow = \text{origin} < p \wedge p < \text{origin} + \text{extent} \} \text{ contains}(p)$ $p \cdot \{ p \in \text{Vector} :- \text{origin} = \text{origin} + p \wedge \text{extent} = \text{extent} \} \text{ move}(p)$ $p \cdot \{ p \in \text{Point} :- \text{origin} = p \wedge \text{origin} + \text{extent} = \text{origin} + \text{extent} \} \text{ setp1}(p)$ $p \cdot \{ p \in \text{Point} :- \text{origin.y} = p.y \wedge \text{origin.x} = \text{origin.x}$ $\quad \wedge (\text{origin} + \text{extent}).y = (\text{origin} + \text{extent}).y \wedge (\text{origin} + \text{extent}).x = p.x \} \text{ setp2}(p)$ $p \cdot \{ p \in \text{Point} :- \text{origin} = \text{origin} \wedge \text{origin} + \text{extent} = p \} \text{ setp3}(p)$ $p \cdot \{ p \in \text{Point} :- \text{origin.x} = p.x \wedge \text{origin.y} = \text{origin.y}$ $\quad \wedge (\text{origin} + \text{extent}).x = (\text{origin} + \text{extent}).x \wedge (\text{origin} + \text{extent}).y = p.y \} \text{ setp4}(p)$	
var origin \in	Point



(Assume a relation $<$ on Points: $p1 < p2 \Leftrightarrow p1.x < p2.x \wedge p1.y < p2.y$.)

In order to prove subtyping in general, we need to prove that all the axioms AX_T of the supertype T are observed by any member of the subtype ST :

$$\begin{aligned} ST \subseteq T &\Leftrightarrow \forall x \cdot x \in ST \Rightarrow x \in T \\ &\Leftrightarrow (AX_{ST} \vdash AX_T) \end{aligned}$$

The model variables present a slight complication. In the formal semantics of the language (into the details of which we shall not go yet) model variables are hidden with existential quantification: the history of the visible features is such that there is a history of tuples (such as $\langle \text{origin}, \text{extent} \rangle$ and $\langle p1, p2, p3, p4 \rangle$) such that the axioms are satisfied. It is therefore sufficient to demonstrate that any relation between the two sets of variables exists, from which subtyping can be proven. Effectively, the trick is to imagine that the subtype inherits the axioms and variables of the supertype, and that an invariant relates the two sets of variables: given that ‘retrieval relation’, show that the inherited axioms are all derivable from the subtype’s axioms, and therefore redundant.

A suitable retrieval relation for this example is

$$p1 = \text{origin} \wedge p3 = \text{extent} + \text{origin}$$

(The invariant in `HVRectangle` constrains $p2$ and $p4$.)

2-3.4 Proofs and theories

The full form of a theorem is

[label :] [variables ·] [hypothesis1, hypothesis2, ... \vdash] conclusion

where the hypotheses and conclusion may be opspecs, ordinary predicates, or nested theorems. A theorem may be used as a proof rule: if, with a consistent substitution of expressions for variables, known theorems can be found to match all the hypotheses, then the conclusion may be inferred. A theorem may be proven either by matching it in its entirety from such a conclusion, or by proving its conclusion within a context in which its hypotheses are assumed.

Each theorem derived in this way should be annotated with a justification pointing to its antecedents. There are many ways to display a proof in this style — the method adopted here is similar to the ‘natural deduction’ style used in Mural.

The generation of a context in which hypotheses are assumed is a localised version of the overall structure imposed on knowledge represented by theorems. A theory is a set of variables and a set of axioms, together with the theorems which can be derived therefrom. We have already met the theories generated by type descriptions. Just as the variables of an individual theorem may be substituted by expressions to specialise the theorem to a specific case, the variables of a type theory — the model variables — may be substituted by expressions to demonstrate applicability to a specific object.

Each theory may be the context within which another is nested: all its variables and theorems are inherited. The outermost types defined by a designer are defined within the context of a standard Fresco theory which inherits information about predicate calculus, sets, arithmetic, the Fresco language, and ‘built-in’ types and classes.

Fresco’s proof support tools impose no chronological order upon the creation of theorems or their justifications. Help in searching a context for theorems to support any

other is provided, and consistency and completeness checks can be called upon. Fresco generates ‘proof expectations’ — theorems which ought to be verified to support an implementation claim. The ‘filing out’ mechanism makes difficult the publication of capsules containing incomplete proofs, and the same checks are run on imported capsules.

2-3.4.1 Model refinement proofs

Since `withinLoop` is not defined here, we cannot verify `contains`, but we can check its relationship to `move`. Since that axiom does not refer to a model, the retrieve relation is not used:

```

h      p · p ∈ Point
1.h1   p · { p ∈ Point :-  $\hat{\uparrow} = \text{origin} < p \wedge p < \text{origin} + \text{extent}$  } contains(p)
1.h2   p · { p ∈ Vector :-  $\text{origin} = \text{origin} + p \wedge \text{extent} = \text{extent}$  } move(p)
1.1h1  v ·  $\text{origin} = \text{origin} + v \wedge \text{extent} = \text{extent}$ 
1.1h2   $v \in \text{Vector}$ 
1.1.1   $\text{origin} + v < p + v \wedge p + v < \text{origin} + v + \text{extent}$ 
           $\Leftrightarrow \text{origin} < p + v \wedge p + v < \text{origin} + \text{extent}$  by subs-eq from 1.1h1
1.1.2   $\text{origin} < p \wedge p < \text{origin} + \text{extent} \Leftrightarrow \text{origin} < p + v \wedge p + v < \text{origin} + \text{extent}$ 
          by Point::Sym+-< from 1.1h2, h, 1.1.1
1.2     $\text{self.contains}(p) \Leftrightarrow \text{self.contains}(p + v)$  by fn-defn from 1.1h1, h, 1.1.2
2      v · { v ∈ Vector :-  $\text{self.contains}(p) \Leftrightarrow \text{self.contains}(p + v)$  } move(v)
          by refine from 1.h2, 1.2

```

The theorem to be proven is formed by the hypotheses h1, h2 and the concluding line labeled ‘3’. There are several subproofs (1, 2, and 2.1) which may have local variables and hypotheses: these are necessary for the application of rules containing subtheorems. Non-hypotheses are justified with `by` rulename `from` antecedents. Rules used here include:

`refine` : $\{ P1 :- R1 \} S, (P \vdash P1), (\bar{P}, R1 \vdash R) \vdash \{ P :- R \} S$

`subs-eq` : $E1 = E2, P[E1] \vdash P[E2]$

`^<-elim` : $A \wedge B \wedge \dots \vdash A, B, \dots$

`Point::Sym+-<` : $p1 \in \text{Point}, p2 \in \text{Point} \vdash E[p1] < F[p1] \Leftrightarrow E[p1+p2] < F[p1+p2]$

‘`E[e]`’ stands for any expression with a subexpression `e`.

Certain rules such as `^<-elim` and its complement, and the commutativity of some common operators, are used so frequently that they are built into the support tool and made implicit.

Another proof, this time using the retrieve relation, and with three useful conclusions (lines 4, 5, and 6) dealing with different supertype axioms:

h1	$p1=origin \wedge p3=origin+extent$	
h2	$np \cdot \{ :- origin=np \wedge origin+extent = \overline{origin+extent} \} setp1(np)$	
1h	$np \cdot nonintersectingLoop(np,p2,p3,p4)$	
1	$\vdash true$	by true-intro
2h	$origin=np \wedge origin+extent = \overline{origin+extent}$	
2.1	$p3=\overline{p3}$	by subs-eq, \wedge-elim from 2h, h1
2.2	$\vdash p1=np$	by subs-eq, \wedge-elim from 2h, h1
3	$true$	by true-intro
4	$np \cdot \{ nonintersectingLoop(np,p2,p3,p4) :- p1=np \} setp1(np)$	by refine from 1, 2.2
5	$np \cdot \{ :- p1=np \} setp1(np)$	by refine from 3, 2.2
6	$np \cdot \{ :- p3=\overline{p3} \} setp1(np)$	by refine from 3, 2.1

2-3.5 Operation decomposition

2-3.5.1 Types and classes

Whilst an object may be a member of many types, it is an instance of precisely one class, which describes its implementation as a list of component variables and a set of methods. It is useful to annotate a class with invariants and opspects, and so we merely extend the type notation to include method-definition. Class descriptions may be derived from each other, for convenience, but that has little to do with any useful behavioural relationship, and is not dealt with here.

Whilst the definitions of a type and a class may be combined in one type/class description, it is not automatic that a class's instances conform to its 'home' type. Fresco ensures that all methods defined in or inherited by a class are proven to conform to the relevant axioms of the home type. Not all the axioms of a type need be provided for by methods in the associated class — partially-implemented 'abstract classes' are allowed. But the proof of a method which creates a new member of a type depends at some point on a theorem of the form

$$C.implements(T), x \text{ class} = C, T\text{-invariants}[selfx] \vdash x \in T$$

and the first hypothesis can only be satisfied by a special 'built-in' justification which checks whether proofs exist that all axioms of T are satisfied by the methods of C .

2-3.5.2 Code development

Although the abstract syntax and semantics of the coding component of the Fresco language is that of Smalltalk, the concrete syntax is somewhat adapted to fit with the reasoning system. (Nor do we expect to be able to give formal rules for every detail of the language.)

The axiom applicable to **setp1** can be satisfied by

```
np · setp1(np)  $\triangleq$  ( var p4 ·
    p4 := origin+extent;
    origin := np;
    extent := p4 – origin )
```

and the proof is largely documented by annotating the code with pre/post specifications. Preferably, the code should be developed from the axiom in stages as advocated in [Morgan].

```

1      origin+extent = origin+extent by A=A
2      { origin+extent = origin+extent :- p4 = origin+extent }
          p4 := origin+extent by assign
3      { :- p4 = origin+extent } p4:= origin+extent by stren from 1, 2
4      { p4 = origin0+extent0
          :- origin=np ∧ p4 = origin0+extent0 } origin:= np by assign
5      { p4 = origin0+extent0 :- origin=np ∧ p4 = origin0+extent0 } origin:= np
          by stren from 4
6      { origin=np ∧ origin+p4-origin = origin0+extent0
          :- origin=np ∧ origin+extent = origin0+extent0 } extent := p4-origin by assign
7      { origin=np ∧ p4 = origin0+extent0
          :- origin=np ∧ origin+extent = origin0+extent0 } extent := p4-origin
          by stren from 6
8      { :- origin=np ∧ origin+extent = origin+extent }
          (p4:= origin+extent; origin:= np; extent:= p4-origin) by seq from 3,5,7

```

These rules are used:

```

seq:      { P :- M1[ $\bar{x}$ ] } S1, { Mi-1[ $\bar{x}$ x0] :- Mi[ $\bar{x}$ x0] } Si ⊢ { P :- Mn[ $\bar{x}$ ] } (S1; S2;...Sn)
assign:   { P[e] :- P[v] } v:= e
var-decl: (var x ⊢ { P:- R } S) ⊢ { P:- R } (x · S)

```

The rules are somewhat more complicated when the possibility of expressions with side-effects is taken into consideration: which is the clearest encouragement to avoid them! Chapter 5 deals with those cases where the effects are on items mentioned in the relevant expressions; cases in which the effects may be on other items are entirely ignored until chapter 8.

An alternative ‘in-line’ style (as in [Morgan]) may be used for documenting decomposition proofs, in which any spec or code may be prefixed by another spec which it refines:

```

{ :- origin=np ∧ origin+extent = origin+extent }
(
  var p4 ·
  { :- p4 = origin+extent }
    { origin+extent = origin+extent
      :- p4 = origin+extent }
    p4:= origin+extent;
  { p4 = origin0+extent0
    :- origin=np ∧ p4 = origin0+extent0 }
    { np = np ∧ p4 = origin0+extent0 :- origin=np ∧ p4 = origin0+extent0 }
    origin:= np;
  { origin=np ∧ p4 = origin0+extent0
    :- origin=np ∧ origin+extent = origin0+extent0 }
    { origin=np ∧ origin+p4-origin = origin0+extent0
      :- origin=np ∧ origin+extent = origin0+extent0 }
    extent:= p4-origin
)

```

Some of the justifications and auxiliary proofs are a little difficult to integrate into the code in such a style, though a good browsing tool should be able to expose and hide them as required (a technique known as holophraxis).

2-4 System composition

All Fresco software development work — specification, coding, proof, documentation — is done within the context of some capsule. A designer may develop several at once within the same system, but has to switch consciously between them: each corresponds to a separate ‘desktop’. Once developed, the designer can ask Fresco to certify the capsule: that is, to check that the proof obligations are all up-to-date and have complete proofs. A certified capsule can then be incorporated into another system.

Each capsule has a name which is unique worldwide: the full identification includes date and hostid of origin, and author’s name etc. are included in the ‘header’ documentation. Each builds on the work embodied in other capsules, its *prerequisites*. A capsule cannot be incorporated into a system unless its prerequisites are already there. The prerequisite graph is acyclic and directed; capsules are not functional modules, but modules of programmer effort: if two modules are interdependent, then they should be defined as separate TCDs within the same capsule; capsules’ dependencies are unidirectional.

During development, Fresco ensures that the designer does not use (or inherit from) anything defined by another capsule which is not a prerequisite. As far as TCDs and global variables are concerned, this is just a question of tracing the definitions of names: every definition in Fresco is associated with a particular capsule. But for messages, this can’t be done with complete certainty until an attempt to construct a proof, which must refer to the definitions of operations in particular types.

On incorporation into another system, Fresco checks that the definitions given by the incoming capsule do not clash with those of other capsules which are not its prerequisites. A renaming scheme can be invented which circumvents some of the problems, where a new definition accidentally has the same name as something else. But in the case where two cousin capsules (with a common prerequisite, but neither prerequisite of the other) try to redefine the same item in different ways, then they can only be declared incompatible and cannot both become part of the same system.

A capsule may only define new TCDs and conformant augmentations of existing ones. The TCDs in a capsule are therefore composed using ‘&’ with the ones

already existing in the system (which should come from prerequisites); so that the new code implements the old specification as well as the extension. (Figure 4.)

Once certified and published, a capsule cannot in general be modified (without renaming it); but a new version may be issued if it conforms to the old one. An extension to the naming scheme encodes the version history (branches are allowed, of course: improvements may be made by diverse authors), and prerequisites must be quoted with name and version. Then any later version will be a satisfactory substitute.

2-4.1 Capsule contents and composition

A capsule is a tuple $\langle \text{name, version, prerequisites, definitions} \rangle$. Definitions includes all TCDs, together with global-variable definitions.

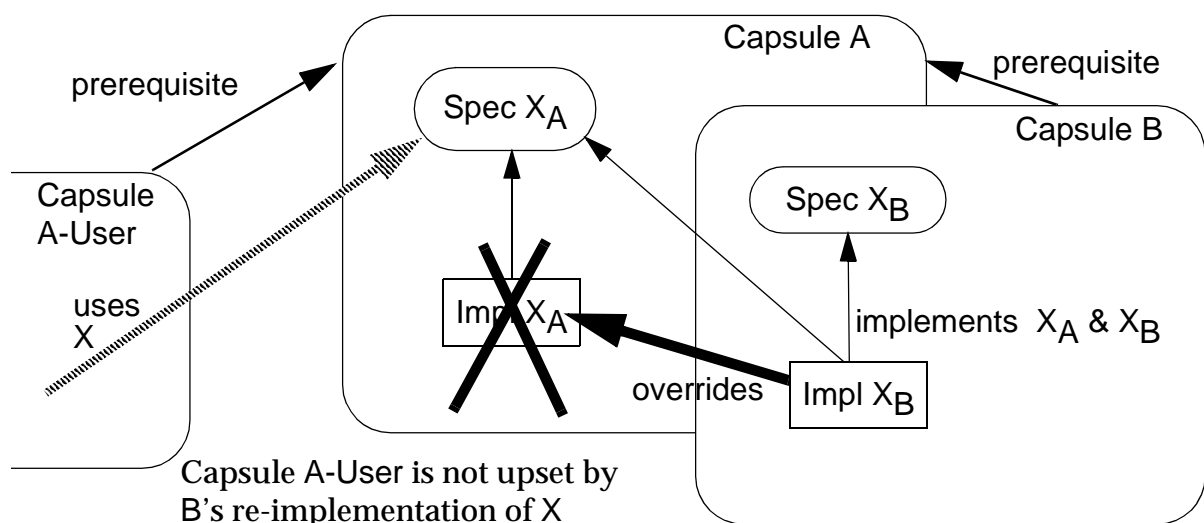
A Fresco system definition is a pair $\langle \text{capsules, definitions} \rangle$. *capsules* is a list of the capsules the system has incorporated. All definitions can be attributed to a particular capsule. Every system has a Kernel capsule, which contains all the standard-issue classes and globals. Run-time components of the system — the heap, stack, interpreter state, and so on — depend on the code in just the same way as in ordinary Smalltalk.

So the definitions in a system are determined by its capsules, and by the order in which they were incorporated, which in turn is determined by the prerequisite graph. Each capsule's incorporation adds the new capsule to the list, conservatively extends the types, overwrites method definitions and adds fields to classes.

2-5 Summary

Class definitions in Fresco also carry type specifications, in the form of model-oriented specifications, which may apply to model or actual instance variables. Inheritance may be conformant or non-conformant: including type information or not. Conformant inheritance guarantees substitutable subtyping.

Figure 4. Capsule composition conjoins specs and overrides implementations



Subtypes may be constructed either by inheritance or by reification, involving a proof with a retrieval function.

Methods may be constructed using a variant of Morgan's method of specification-statements.

Proofs are constructed in a style similar to 'natural deduction'.

The unit of design modularity is the capsule. The design of each capsule is founded on some set of precursors. Each capsule effectively defines a theory of the definitions it carries, which is used by its importers. Capsules contain new class definitions, and augmentations of old ones. Fresco ensures that (if proof obligations are correctly fulfilled) no conflict will arise between capsules in any configuration.

The Fresco notation includes diagrammatic elements (used for analysis and design), inexecutable assertions (used for specification) and Smalltalk code. Classes may be related by conformant and non-conformant inheritance, reification, and composition; the last may carry a contract.

The proposed Fresco environment will support the interactive development of capsules, including their type/class definitions, contracts, and proofs.

3 The state of the art: background and issues

This chapter combines a survey of relevant work by others with discussions of a number of issues which arise from or contrast between their efforts. There is a brief summary of where Fresco stands on these matters. The scene is thereby set for the detailed description of Fresco which follows in chapters 4—6.

3-1 Formal methods

3-1.1 Specification styles

Algebraic specification [GGH] defines a type using equations over its visible operations. Advantages and disadvantages of this technique:

- The equations are deterministic. Where loose specifications are required, a separate kind of specification must be used. This limited expressiveness tends to push the specifier into making design decisions too early on.
- Well-suited to execution for prototyping purposes with a Prolog-style interpreter. In Fresco, this is not seen as an advantage, since OOP already provides for rapid prototyping of a superior sort (§1-3.1 — p.12).
- Not so easy for large specifications. It is inevitable that existentially-quantified intermediate variables will be required for the equations of anything more complex than the usual examples (stacks, natural numbers, and so on). These correspond to components in some hypothetical internal state. The hypothesising of such a state is the basis of the model-oriented methods — except that they force a consistent model of the internal state over all the axioms, rather than inventing a new one ad hoc for each axiom.
- Amenable to proofs using term-rewriting systems.

Model-oriented specification begins with the definition of a hypothetical internal structure of the objects of interest, and defines all the visible operations in terms of their effects on that. Advantages and disadvantages:

- The model may be mistaken for a description of the required or expected implementation.
- A badly-designed model may contain states indistinguishable by the use of the external operations. The usual refinement techniques can be confused by this state of affairs, disallowing valid implementations.
- Particular specifications are easy to understand, and the technique in general is easy to teach. This is as true for large specifications as for small ones.
- Since one type is always modelled in terms of others, it is impossible to define primitive types this way.

In addition to the data structure (usually a record-like definition), there is an *invariant* which constrains the model structure to an appropriate set of states, and operation-specifications (*opspecs* in Fresco parlance) which consist of a *postcondition* and in some languages a *precondition*.

The advantages of model-oriented methods seem to outweigh those of the algebraic: Fresco has no need for prototyping in the logic programming style; we need loose specifications; and term-rewriting appears to have limited utility in proof systems.

3-1.2 Traditional specification methods

The principal features of the model-oriented specification language **Z** [Spivey] are

- Modularisation by “schemata”.
- A good associated tradition of interspersing formal schemata and informal text.

The schema structure of a Z specification is chosen for readability; but each schema cannot in general be implemented independently of the others — the structure needs to be flattened before a reification can be done.

There is a poor separation in Z between client and provider: for example, clients must infer the precondition of an operation from the postcondition and invariants — there is little emphasis on a clear contract. This is remedied in many OO variants of Z.

VDM [Jones86]

- has always been a development method, whose notation “Meta-IV” (almost universally called “VDM”) was designed with that in mind.
- There has been little work on modularising VDM until recently [Fitzgerald 90, Bear], so that specifications tend to be big and daunting.

To develop OO forms of VDM, a method of composing specifications must be added — necessary for a useful form of inheritance.

Hoare suggests [Hoare90] that Z forms the better base for specification, and VDM for development. This advice is followed to some degree in Fresco: the style of presentation of specifications, and the composability of pieces of specification owe their inspiration to Z; while the emphasis on client-provider contract, data reification, and the philosophy of rigorous proof come from the author’s experience of VDM.

Larch [GHW85] is a two-tier specification method, in which algebraic methods are used to define terms used in the second tier, where they are employed in the style of the chosen implementation language.

3-1.3 Decomposition strategies

Once the requirements for a particular operation have been defined, there are a number of approaches to producing and verifying the code which is intended to meet them.

Assertions & VCG. After writing a method, it is peppered with assertions — predicates intended to be true whenever execution passes over them. The peppered text is given to a Verification Condition Generator, which generates a verification condition — a large predicate which you try to prove. It is often difficult to see the connection between any part of the VC and the features of your program; so it is

difficult to get intuitions about how to prove it, and hence difficult to see where the problem is if you get stuck. The assertions are added after the program is written, and there may be a tendency to write assertions which differ little from the code. Nevertheless, more real code has reputedly been verified this way, meeting the highest requirements of the national software reliability standards, than by any other; the system used was Gypsy [Good 82].

Stepwise refinement: A *specification-statement*, consisting of a pre & postcondition pair, can stand for any segment of code not yet developed, permitting reasoning about the surrounding code to proceed. You begin with a specification statement, and find a refinement or decomposition. The step is a small one, and so easily proven. After many such steps, the method is decomposed entirely to code. The principle was expounded in [Jones80], and is well illustrated in the refinement calculus [Morgan 90], in which the steps can be documented ‘inline’ with the code.

This is the approach adopted in Fresco. The inline documentation of reification on the small scale is in keeping with the larger scale OO use of the abstract class, which represents a specification documented in the same space as the code.

Transformational development, the stepwise application of rules to a specification to turn it gradually into a program, has been demonstrated for applicative languages such as Refine [Green]. The technique can also be used for optimisation. The collection of rules must be extensible by the user. It would be interesting to try a mixture of this technique and the refinement style in the context of a rule-based system such as Fresco.

3-1.4 Inference methods

Term rewriting. A number of efforts have been made to produce fully automatic theorem provers. They have generally centred on the technique of *term rewriting*, in which the machine repeatedly applies conditional equalities — that is, of the form

$$\text{hyp1, hyp2, ...} \vdash f = g$$

This is especially applicable to the algebraic specification methods, where data types are described in this form. Term-rewriting tactics deal very well with propositional logic, and some success has been had with more sophisticated problems.

The fully automatic theorem prover of Boyer and Moore used term-rewriting: their experience suggests that when such a tactic fails, it is often difficult to see where the problem lies. But since the machine’s complement of rules and tactics can never be sufficient to cover all possible programs, human intervention is often necessary, to suggest new axioms for the machine to try. [Lindsay 88]

It seems possible that as ‘artificial intelligence’ techniques improve, machines will yet be got to do the bulk of theorem proving; indeed, this is a necessary precondition for widespread use of formal verification. Until that day, a co-operative effort between user and machine is necessary; and for that reason, half-done proofs must be easy to read and to relate to the problem domain, so that the user can take over where the machine gets stuck.

Natural Deduction is a proof system of Gentzen and others [Prawitz71], designed to meet this criterion of reflecting human appreciation of the problem (an interesting parallel to the aims of OO analysis and design). This is the basis of the proof system in Mural [Mural], the proof-assistant precursor to Fresco. In Natural Deduction,

proofs are composed of lines each of which is derived from a subset of its predecessors by matching a proof rule. Each proof begins with a set of hypotheses; the proof supports any theorem with those hypotheses and a conclusion matching any of the derived lines. Proof rules and theorems are identified. A hypothesis may be a straight assertion or a sequent $(X, Y \vdash Z)$, and where such a hypothesis is encountered in a proof rule, it must be matched with a subproof — that is, a proof in which the preceding lines of the containing proof may be used as supports.¹

In Mural, theorems are organised into theories, each of which is defined by a set of axioms and constants. A theory-definition may inherit axioms and constants (and therefore derived theorems) from one or more others, forming an acyclic graph. This permits mathematical knowledge to be built up in a way which reflects a formalised view of the working style of mathematicians: one body of ideas is built upon the conclusions of predecessors.

Fresco adapts this system, identifying sequents with theorems and proofs with theories: they are generalised into the idea of a ‘Context’; types are then introduced as a specialisation of Context.

3-1.5 Proof tools

Mural is a **proof editor**: a tool which displays and browses theories and their theorems and justifications, helps seek justifications, helps instantiate new justifications, and ensures the consistency of the proofs and theories. Contrast this with the Boyer-Moore system or with Gypsy, in which the machine works in batch mode, searching for proofs or checking them.

The interactive style has become easier to support as hardware has advanced. LCF [GMW79] was teletype-based; Mural is window+mouse-based, making it easier to think of the tool as providing a window on a database of proofs, constrained to be consistent with the rules.

In fact, the constraints should not be imposed too harshly any CAD system should allow inconsistencies whilst the elements are being juggled and experimented with, but draw them to the user’s attention, and provide the means to find outstanding incompletenesses or inconsistencies.

LCF developed the notion of **tactic**: an algorithm or heuristic for discovering a proof, described in some suitable language, and using the same searching and rule-application primitives as are available to the human. The user can invoke tactics where it seems likely that the machine will be able to find a proof: a well-designed set of tactics should permit the user to do the creative sketch proofs, and then to get the machine to do the straightforward filling-in of the details.

1. An alternative and more-or-less ‘natural’ view of proof as arcade game may be envisaged. You begin with a goal-assertion, inscribed in a small cloud and gently floating around the screen, bouncing at the edges. You have a battery of rules, one of which you zap it with: there is a small explosion (with suitable sound effects) as it breaks up into subgoals, to which you then apply rules in the same way. Some subgoals will directly match rules, and if your firing strategy is well-chosen, you eventually clear the screen of goals; if, on the other hand, the screen gets so cluttered with a heaving mass of logic that you can no longer see the rules for the subgoals, you can consider yourself to have lost. This is all just a matter of user interface of course, and should be good for occasionally peppering up jaded theorem-provers of the future, particularly those who have come into the business from particle physics.

[Mural] documents an attempt in this direction. It is only a partial success, since some rules (for example the induction rule) have so many possible ways of being applied to any subgoal that a mindless series of matches would be hopelessly long. However, there is a set of built-in and user-definable tactics that can be invoked when desired.

3-1.6 Logic

First order predicate calculus is inconvenient as a logic in which to deal with partial functions, which are very common in computing. For example, intuition suggests that even where $x < 0$, there is a useful meaning to this formula:

$$x < 0 \vee \text{sqrt}(x) < 2$$

where in FOPC, it would be undefined.

LPF (“Logic of Partial Functions”) [CJ 90] is one of the logics for dealing with such formulae. There are others, but LPF has the virtues that:

- only one extra operator — δ , for testing definedness — is added;
- the symmetry of the operators is preserved;
- and the theorems are straightforwardly those of FOPC, after deleting all that depend on $p \vee \neg p$ (plus some new ones for dealing with δ).

The intention in LPF is to interpret formulae the way designers would expect, so the above proposition is true if $x < 4$ and false otherwise, including cases below 0.

3-2 Object orientation

3-2.1 Definition of object-orientation

[Wegner 90] defines an object-oriented language as one including these features:

- **Object:** a collection of operations associated with a mutable unit of state; the operations can alter the state and yield results. An operation is invoked by applying or “sending” a **message** comprising a **selector** name or signature and arguments. The same selector may be implemented by different **methods** in different objects.
- **Class:** a definition of internal structure and methods shared between the objects which are **instances** of that class. We may speak of the **attributes** of a class, meaning the definitions of the names, types etc of the attributes of its instances.
- **Inheritance:** the derivation of one class-definition, a *subclass* from one or more others, its *superclass(es)*.

3-2.2 Subclasses

Halbert & O’Brien [HO87] list many uses for the subclass relationship:

- Generalisation — adding extra information, for example adding **width** to adapt a class of squares to represent rectangles.

- Variation — providing variants of a basic theme.
- Composition of characteristics from multiple inheritors, for example to construct `ScrollableWindow` by inheriting from `Scrollbar` and `Window`.
- Specialisation — e.g. `FourSidedShape` to `Square`.
- Reification — e.g. `Set` to `BitMapSet` or `SparseArraySet`.

The art of arranging a subclass hierarchy to achieve maximum flexibility is discussed by Johnson and Foote [JF88]. They also discuss the definition of **frameworks** — groups of interdependent classes, like Smalltalk’s Model-View-Controller system, which can be specialised as a whole group. [Helm90] discusses the documentation of such frameworks and the contracts between their members.

3-2.3 Types

As the theory of the semantics and utilisation of object-orientation has grown, the idea of **encapsulation** has been clarified. It is the separation of concerns between **client** and **implementor**. The client is concerned only with the externally visible behaviour of the module, defined by its specification. The implementor is concerned to satisfy the specification by providing a suitable implementation.

This gives rise to the notion of **type** — a set of objects which behave according to a given specification from the external client’s point of view. A class defines the internal structure which implements a type. Its clients are interested in the type that an object belongs to, and the type-description represents the set of assumptions that the designer of the client makes about the object being used.

[CHC90] points out the class/type distinction, and the corollary that subclasses do not necessarily define subtypes. A **subtype** is a subset of a type: if $x \in TT$ and $TT \subseteq T$, then $x \in T$; anything that can be said about members of T is also true of any member of TT , and any client of T will be satisfied with any member of TT . Subtyping, rather than subclassing, is clearly the more important property when the aim is to write polymorphic code (e.g. a screen manager which can handle many kinds of window). From a software-engineering point of view, the dependency between the client and the supertype (the generic window description) and its independence from the individual subtypes (of particular kinds of window, icon, etc) is worth a lot more than the subclassing facility of factoring some code.

[Meyer88] advocates only using subclassing where the intended subclass also implements a subtype; this cuts out all but the last two of Halbert and O’Brien’s uses. He and others have pointed out that in many (though not all) of the other cases, the economy of implementation turns out to be a false one, for three reasons:

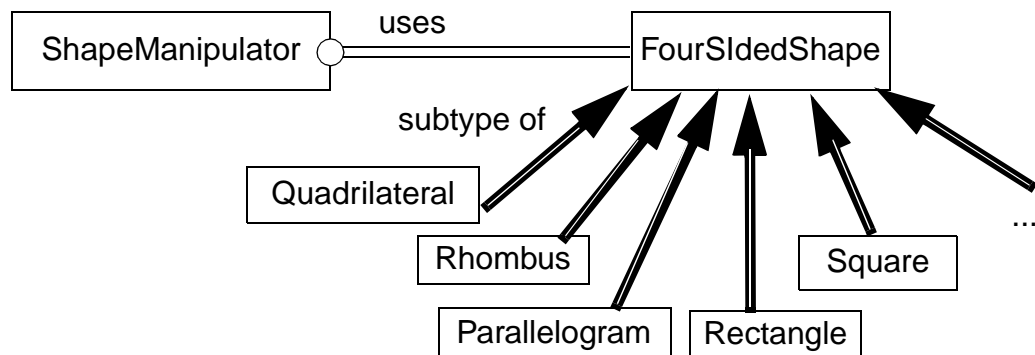
- The implementation is typically overcomplicated with overriding redefinitions of methods.
- Changes of requirements tend to affect types (and therefore their subtypes) rather than subclasses — so any change is liable to require a rewrite of the class hierarchy.
- The “yoyo problem”: because some methods are defined within a class and others left to its superclasses, it can be difficult to follow the precise execution path of a program as it goes up and down through the different levels of the hierarchy. The reader must therefore employ procedural abstraction: that is, when you come to an operation call, do not put your finger on it and

attempt to look up its implementation, but instead use its specification to treat it as an atomic operation. If subclasses are also subtypes, then it is sufficient to know the specification relevant to the type you know the receiver belongs to: it is not necessary to treat different possible subtypes separately.

Further guidelines as to the design of class hierarchies have appeared, such as the 'Law of Demeter' [Lieberherr 88] (which prohibits sending messages to objects not your immediate neighbours) and the principle that all classes should be either leaves with no subclasses, or abstract classes, with no instances, representing types [DT92].

The notion of inheritance can apply to any kind of definition, not just classes. Notice that inheritance between type definitions does not necessarily imply that the inheriting definition will be a subtype. [Cusack90] examines the properties a language must have in order to obtain this desirable relationship. Fresco's type definitions do have this property.

Polymorphism appears in two forms in OOP: inclusion and parametric.



From this it is clear that the purpose of `FourSidedShape` is to specify what `ShapeManipulator` can manipulate. `ShapeManipulator` is dependent upon `FourSidedShape`, in the sense that if `FourSidedShape` were altered, `ShapeManipulator` would have to be examined, possibly altered too, and in any case reverified. These dependency diagrams are a powerful tool in software maintenance. However, `ShapeManipulator` is *not* dependent upon any subtype of `FourSidedShape`: provided they really are subtypes, they will all behave at least in the way `ShapeManipulator` expects (as well as doing more that it isn't interested in). Furthermore, we could add new subtypes, and `ShapeManipulator` would deal with them just as well, without any need for reverification of it. What *is* necessary is to verify that the new type really is a subtype, but this is a cheaper proposition than going around all the dependents of `FourSidedShape`, many of which may be unknown to its author.

Parametric polymorphism: A module of code may also be polymorphic in the more general sense of being parameterised by type. For example, `SortedList(Integer)` is a class or type (whereas `SortedList` by itself is not). In general, it is necessary to restrict the parameter to have a certain set of characteristics: objects can be sorted only if they belong to a type which has an ordering relation.

3-3 Existing syntheses of object orientation and formal methods

Three strategies have been followed by OO formalists; they may at some stage converge:

- the introduction of OO principles to an existing specification language, with the objective of introducing OO benefits into the business of specification;
- the adaptation of a specification language or technique to facilitate the specification of OO program components;
- the introduction of specification elements into an OO programming language.

The following sections describe some of these languages; then we shall look at some of the issues upon which they differ.

3-3.1 Object-Z

OZ [CDDKRS] inherits from Z several pleasant presentational characteristics, and adds a new class construct that encapsulates state and operation schemas. A semantics has been described, in terms of possible sequences of operations (and their results) [DD90].

As an OO-ification of a specification language, it is not OZ's chief concern to specify OO program modules. Therefore, for example, there is no built-in distinction between equality and identity: if these are to be modelled, they must be explicitly described by the specifier.

Inheritance is used as a convenient device for constructing specifications: there is no strict relationship with subtyping. Thus, for example, one may describe a class *Quadrilateral* with an operation *Shear*, and then derive from it a *Rectangle* with *Shear* redefined to be unusable, and a stronger invariant.

3-3.2 Z++

Lano and Haughton have done considerable work on the refinement of specifications written in OO extensions to Z [LH92], and have designed Z++ as an exemplar. Object identity is a feature of the Z++ semantics. Z's schemas are rejected as too difficult to reason about, and class and method definitions are used as modules in their place. Z++ has a strict notion of subtyping, and inheritance is designed to guarantee strict subtyping, provided you are willing to accept their renaming convention. For example, *Rectangle* can have its own version of *shear* which does nothing; but it is also considered to have the method *Quadrilateral::shear* — so that if a client always qualifies a method-name with the name of the type expected, the right things are guaranteed to happen. This seems a bit of a cheat somehow.

3-3.3 OOZE

The prospective extender of Z should ignore its semantics in favour of its superficial appearance, which is by far its best characteristic, and the cause of its wide popularity among the general masses. In particular, it should be appreciated that schemas

are a purely syntactical device, and that the schema calculus, unless firmly rejected, will only interfere with any proper modularisation scheme.

Alencar and Goguen's "Object oriented Z Environment" [AG91] provides a Z-like syntactic appearance to a well-developed algebraic system, OBJ3. Methods are specified with conditional equations, and there is an executable subset of the language, intended for prototyping.

Loose specifications cannot be written with equations, although there is a separate "Theory" construct for that purpose. This duality seems a little uncomfortable, and seems likely to bias the specifier towards implementations.

Subclasses are not necessarily subtypes (in the strictest sense), since method specifications can be arbitrarily overridden in subclasses. However, if a method specification "promotes" the superclass's version of itself, correct subtyping is assured.

Modules (for encapsulation) and classes (templates of objects) are separate concepts in OOZE — though the syntax makes it easy to make the two coincide.

3-3.4 Abel

Abel [DLO86] is one of the most well-developed languages in this field. It is a wide-spectrum language — that is, with both inexecutable specification features and imperative executable features, and there are formal verification rules and tools.

As in OOZE, specifications are algebraic in style, leading to the necessity of a separate kind of module to specify loose properties. Unlike OOZE, a good OO programming language is part of the package, rather than just a rule-based prototyper.

3-3.5 Larch/Smalltalk

[Leavens 90, 91] describe the application of Larch to OOP, and to Smalltalk in particular. The scope of the work is restricted to immutable values at present. This seems partly to be because of the extra difficulties of working with mutable objects in Larch's algebraic layer.

3-3.6 VDM++

The aim of VDM++ is to introduce the benefits of object-orientation to specifications [DK91]. Subclasses may be formed by selective inheritance — for example, *Rectangle* may be formed from *Quadrilateral* by omitting *shear*. This ensures that *Rectangle* is self-consistent — though of course does not ensure that every *Rectangle* can be treated as a *Quadrilateral*. It is a convenient constructional mechanism, but unrelated to subtyping.

The primitive types (numbers etc.) and constructors (sets, lists, mappings) are as in VDM. Object-identities are a special primitive type.

3-3.7 CDL and EVDM

The work of Huw Oliver [Oliver 88] aims to specify re-usable program components; Ada is used as the exemplar implementation language. It is therefore not really object-oriented, but more about packages (which do not have an instance-creation mechanism). Specification languages CDL and EVDM (both based on VDM) are

given a semantics in terms of the kernel language COLD-K [Jonkers 88]. One interesting feature of the work is that CDL is the result of experiments with specifying Ada modules, while EVDM comes from the other direction: it is the result of adding modular features to VDM.

3-3.8 Utting & Robson — OO Refinement Calculus

[UR91, UR92] describe a system with similar objectives and approach to Fresco. Subtyping is of the proper substitutable variety, and specification-statements [Morgan 90] are the basis of the language. This is therefore one of the few pieces of work to investigate verified refinement to code, in the OO arena.

3-3.9 Eiffel

Eiffel [Meyer 88] is a programming language in which classes may have invariants, pre and postconditions. Meyer points out that it is only with specifications that inheritance takes on its full meaning (as subtyping). Assertions are written as boolean expressions of the programming language.

In Eiffel, opspects and invariants are written in terms of instance variables. This means that in an abstract class, you either must write quasi-algebraically, defining each public operation in terms of private operations; or you must introduce instance variables where you otherwise might defer them until the definitions of the subclasses. This is somewhat anti-encapsulation.

3-3.10 Annotated C++

A++ [CL90] adds invariants and pre/post conditions are added to classes in C++. C++ “public” inheritance is identified with the intention to implement a subtype. The axioms used for specification are uniform with the assertions that may be inserted into the code (in Morgan’s style). C++ expression syntax is extended to provide quantifiers; but where possible, the A++ compiler can use the assertions to insert debugging checks.

Cline and Lea insist that axioms defining behaviour should use only the publicly visible functions, and not private instance-variables. This is a reasonable restriction in C++, where the specification defined in a superclass should not bias the implementation — variables once declared cannot be undeclared in subclasses. But the consequent restriction to an axiomatic style is tedious for all but the simplest types.

An alternative scheme would be to introduce the idea of abstract model variables, which would be ignored by the compiler. (This would work for Eiffel as well.)

3-3.11 POOL

Although POOL [America 87] does not have a full specification element, it is worthy of notice for its syntactical separation of type and class. Types define sets of objects which conform to a given signature and set of properties; but in its current incarnation, properties are documented only informally. To check refinement, the compiler compares property names. Properties may refer to any aspect of the object’s behaviour. The rules for subtyping are interesting, and cover generic types.

Both types and classes may be defined by inheritance (each from other definitions in their own category). Because the behaviour is described only by property labels, there can be no guarantee that an inheriting type definition is a subtype. Some of the properties are informally described as invariants on states, and inheritance of invariants can lead to non-subtypes (see §3-4.5 — p.47).

3-4 Issues in application of formal methods to OOP

It may reasonably be asked why the application of formal methods to OO programming requires any special consideration. This section reviews what we require of such a combination, and where the particular difficulties are.

3-4.1 Concurrency

Concurrent programming is not dealt with here. However, many of the same problems arise, because of the complexities of imposing encapsulation strictly in a typical object-oriented system, especially those problems associated with aliasing (see below).

3-4.2 Objects

3-4.2.1 Object identity

The specification of program modules requires that every behavioural aspect that a client might need to know should be formalisable.

In many OOP languages, and always in Smalltalk, information is passed by reference, and subcomponents are references rather than complete structures. In Smalltalk, Eiffel, and others, the syntax treats this plethora of pointers implicitly, treating a reference as if it were the item referred to.

This pretence breaks down wherever two pointers refer to the same object. In Smalltalk, $x==y$ is true iff both names contain equal pointers to the same object; whereas $x=y$ depends on the classes of the object(s), and will usually compare the values the objects (with any subcomponents) represent in their current state.

A specification method must be able to make this distinction, and must be able to cope with user-defined equality.

3-4.2.2 Aliasing

A frequent precondition or invariant is that there should be no aliasing between given names or their subcomponents; a specification language should be able to state this, and the proof system should be able to verify it.

Object-Z, for example, appears not to tackle this issue at all; nor does Larch/Smalltalk. OOSE acknowledges object identity, and Abel contains provisions for preventing unwanted aliasing.

3-4.3 Encapsulation

Encapsulation is the minimisation of the dependencies between units

- so that the work of creating them can easily be partitioned, with minimal communication between the creators of different units;
- so that one unit may be used in conjunction with many others;
- so that the impact of alterations to a unit on its neighbours can readily be assessed and minimised.

3-4.3.1 Units of encapsulation

The obvious unit in OOP is the object; and some languages (such as Smalltalk) provide encapsulation on this basis: no method may access the instance variables of any but its receiver-object. In C++, the unit of encapsulation is the class (and, imperfectly, the program file): a method may access the innards of any parameter belonging to its own type.

The behaviour of objects is determined by their design, and so it makes more sense to encapsulate in the units of design, than the run-time objects. This supports the C++ strategy — but in general, that argument is flawed: a subclass may choose to implement some other way, and whilst a method can be certain of its receiver's class, it can be certain only of other objects' types. E.g.:

```
IntSet:: union (IntSet s2) // s2 is some kind of IntSet
{ size= size + s2.size ; // but if s2∈ IntSet, size is unused ...
```

— though such pitfalls can be avoided by the observance of various 'good programming' rules (such as "no variables in abstract classes & no subclasses of concrete classes" [DT92]).

[Szyperski 92] argues for the separation of encapsulation and classes: encapsulating modules should be groups of classes, and free access should be allowed between the innards of classes defined within a module. Since classes are often defined in groups, this is a very useful idea, realised in OOZE. Further, it may be argued that modules and classes may be quite orthogonal, with modules (or "capsules" in Fresco terminology) able to define new classes or extend existing ones.

Szyperski argues for "no paranoia": you should be able to get at the innards of anything defined within your own module. This freedom suffers from the same problem as per-class encapsulation, unless all possible subtypes are confined within the module. The problem can be avoided by encapsulating both on the object and module levels: variables should be private to objects, whilst there should be messages which are private to modules.

(Multiple dispatching could also prevent this insecurity.)

3-4.3.2 Applicability of encapsulation to specifications

In some respects, encapsulation only applies where there is a reification, whose details are hidden so that clients do not depend on them. But a specification should be fully exposed, since clients use all its properties in their own correctness proofs. Encapsulation is the prevention of dependency between a client and a provider's implementation: so that more efficient or more powerful code may be introduced. Neither of these is an issue for specifications; if a nicer way of stating a specification

is found, it can be used alongside the old one, rather than instead of it. And if the specification has to be changed, then the clients will have to be re-assessed anyway. However, normal functional abstraction remains useful for specifications — if $s \in \text{Stack}$, it is easier to say $s.\text{popTo}(s')$ than $s'.\text{size} = s.\text{size}-1 \wedge \forall i \in 1..s.\text{size} \cdot s[i]=s'[i]$. When an operation has been specified for possible implementation, it is also useful to be able to quote the specification within another specification by quoting the specified method — $s.\text{pop}$ — often known as *promotion*. Z and its derivatives make much use of this technique.

3-4.3.3 Encapsulation and invariants

In conventional specification, there is effectively one state-invariant, which applies to the whole state. Each operation can rely on it being true on entry, and must ensure its truth on exit. In this model, no activity happens in between calls to the system's operations, so nothing can disturb the truth of the invariant.

In OOP, we aim to apply formal methods on a per-class or per-module basis. An invariant typically is quoted within a class, and applies individually in each object of that class. The standard methodology can fail for two reasons:

- There may be ways in which an invariant may be invalidated in between calls to the methods of the class. In particular, if any other operation in the system has access to an object on which the invariant depends, then that may be altered without the use of the class's own methods.
For example, a SortedList class has the invariant that its components are ordered; if a pointer to one of the (mutable) components is available to some other part of the system, that component may be changed so that it is out of order, without the SortedList being aware of that. It is not always efficient to keep such components entirely within the control of the 'owner' object.
- It is conventionally assumed that while an operation is in progress (and the invariants are temporarily violated) no other operation may be called. In an OO system, there is no automatic guarantee that the message passing map will not be circular.

In both cases, the problem is the attempt to encapsulate: in the first case, pointers cross the encapsulation boundary; in the second, the encapsulation prevents us from knowing enough about the things we call to be sure they won't come back to us. In order to prevent these problems, extra specification and verification techniques will be required.

3-4.4 Classes and types

3-4.4.1 Classes and types, subclasses and subtypes

We have seen that there is an important distinction to be made between the specification of an object's behaviour (a type), and its implementation (a class). An object may be a member of many types, but is an instance of one particular class. The definitions of types and classes may each be derived from other definitions by inheritance or parameterisation; in defining a language, we seek to give these derivations useful properties.

In POOL, there are two different notations for classes and types. This makes the distinction clear to the programmer, but makes it more difficult to apply specification

constraints directly to implementation constructs — a class can be asserted to conform to a particular type, but an invariant cannot be applied to the internal variables of a class.

Meyer’s approach recognises the difference between class and type, but attaches type information to classes. This has the advantages that there are fewer pieces of description to worry about; that the specification constructs (pre/postconditions and invariants) can be applied directly to the attributes and methods of the class; and that where it is sensible to provide a common implementation for a supertype, this can readily be done.

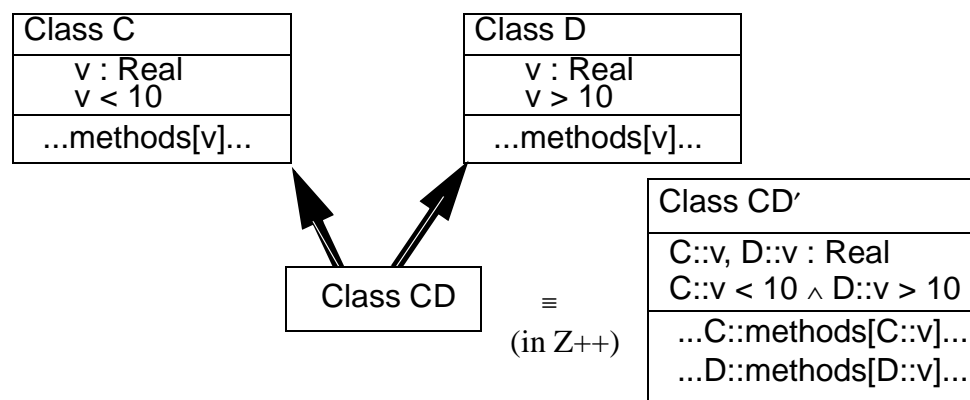
Some authors suggest that every class should either be an abstract class (with no instances of its own) or a leaf class, with no subclasses. The purpose of this seems to be to emphasise the difference between class and type — their abstract classes are really hangers for types; but where there is a clear type specification attached to each class, this seems less relevant.

In Fresco, types and classes are represented by a single syntactical construct, even though subtyping and subclassing are distinct. Every class has an associated ‘home’ type to which every instance of it and its subclasses are intended to belong. In Fresco, the type specification can be fully formalised, so this ideal situation can be realised more consistently than in other languages.

3-4.4.2 Composing types

When two specifications are combined — as for example in multiple inheritance — the result may be interpreted in various ways.

There are two kinds of inheritance in Z++: strengthening, in which an extra invariant and attributes are added to a class; and strict inheritance, in which the attributes and methods of the superclass are renamed to avoid clashes with any other superclasses and the additional material in the subclass.



Z++ -strict inheritance does ensure that the subclass is a proper subtype of the superclass: the latter’s workings are insulated from all other material in the former. But this isolation does make extra work for the specifier, who must then make the connections explicitly by the use of “promotion” — that is, quotation of the appropriate inherited methods in the subclass. Inheritance is therefore little more than import.

An extra complication is that if a variable is defined in a common superclass — e.g. w defined in A which is supertype to both C and D — then it should not be renamed in CD. Work on COLD-K [Jonkers 88] has formalised this in a calculus of ‘origins.’

An alternative approach — espoused by Object-Z and others — is not to bother with renaming, permitting the final specification of an operation to be determined by several statements from various sources. The advantage of this is that it helps modularise the specification.

3-4.4.3 Monotonicity

However, the modularity is not tremendously useful from a software engineering point of view, unless specification-composition has the property of *monotonicity*. That is: if a client can see a specification S of some unit (class, method, ...) M then any inferences that can be made from S will remain true in all compositions of S with other specifications of M . Monotonicity is important in managing the building of a large system, and in the sense of ‘backwards compatibility’ when a system is to be updated.

Z++’s inheritance is certainly monotonic, but requires explicit promotion of operations from superclass to subclass (except where there is no change or clash). It is more appropriate for a specification language than where implementations are involved, and where renaming would be a complication.

Object-Z’s inheritance is not monotonic: predicates may be arbitrarily added to an invariant or method-schema, adding constraints which a client of the superclass would not expect.

In Fresco, there is no renaming, but the conjunction of specifications (of types and methods) is defined to be monotonic. This may result in a specification of *false* where there are inconsistencies. The philosophy is that clients (of **Class C** and **Class D** above for example) are always guaranteed that members of subclasses will behave as expected; but there are inevitably some compositions (such as **Class CD** in the Fresco version, where we would have the invariant $v < 10 \wedge v > 10$) which are unimplementable.

Monotonic composition is used in several ways:

- Software may be documented with a mixture of formal specification and explanatory text. Different aspects of a type may be described separately, in separate boxes. Any client interested in only one aspect may rely on the validity of whatever can be inferred from any of these partial descriptions, without having to check out the rest.
- One class may fulfill the requirements of several clients, which refer to different types to specify their expectations.
- An implementation may be improved. Old clients will continue to work, provided the new aspects augment the old specification monotonically.

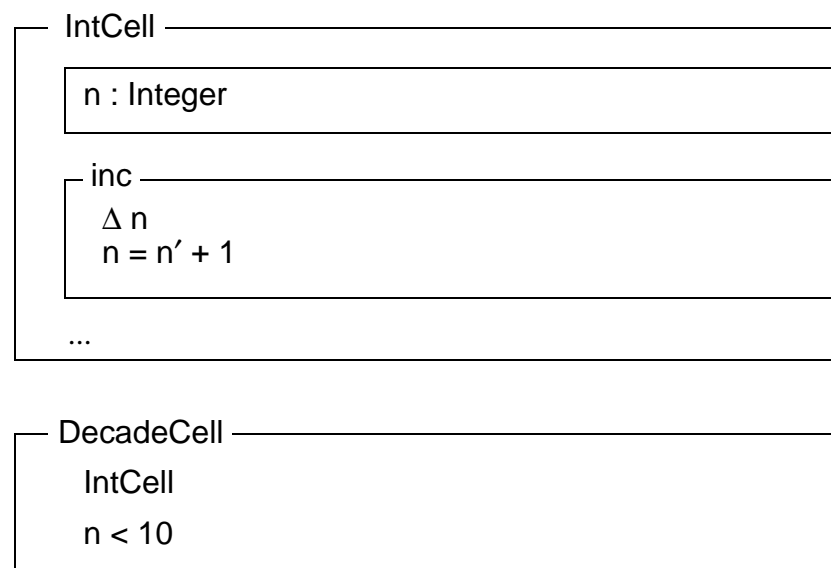
3-4.5 Inheritance and subtyping

It is therefore essential to get right the rules for interpreting inheritance, at least of type definitions. One of the key features of OOP facilitating re-use is polymorphism. As software engineers, we are therefore crucially interested in substitutability; and so in Fresco, **A** is a supertype of **B** if and only if all objects which have been shown to operate according to the rules determining membership of **B** will always conform to all clients’ expectations of **A**-members.

Class definitions may inherit merely as a convenience of construction; but it leads to a more readable and maintainable system if every subclass relationship is associated with a subtype relationship. In C++, this corresponds to the intention of “public” subclassing. The occasional utility of “private” (non-subtyping) subclassing is provided for in C++; but many authorities argue against its use (see page 38).

Type definitions could also be derived one from another without the result generating a subtype, as may happen in Object-Z; but this seems likely to lead to confusion. The absence of explicit preconditions in Z leads to this folly. If a new invariant restricts the state space, then clients are supposed to know that they should not call any operations which would take an object into the forbidden space.

For example,



DecadeCell is interpreted as being like **IntCell** with a restricted state space. Users of `inc` are supposed to know when it is valid to apply it — preconditions must be inferred: the valid ‘before’ states are those for which there is a valid ‘after’ state that meets the postcondition:

$$\text{pre}(\sigma) = \text{inv}(\sigma) \wedge \exists \sigma' \cdot \text{post}(\sigma, \sigma') \wedge \text{inv}(\sigma')$$

In a world of immutable values, invariant-strengthening always constructs a subtype. But for mutable objects, subtyping depends on the operations.

By contrast, in a language with preconditions (based on VDM), the restriction would be likely to render some operations unimplementable: for example, if `IntCell::inc` had a `true` precondition, then `DecadeCell::inc` — and hence the whole class — would be unimplementable, because it could not deal with `n=9`. Thus in this latter interpretation, any non-subtyping inheritance would be caught in the attempt to verify an implementation.

(A weaker interpretation of preconditions says that they should be considered in conjunction with invariants; but this seems to render them pointless.)

3-4.5.1 Subranges are not subtypes (for objects)

It is the mutability of objects which is important in this respect. To take another example, an immutable rectangle (think of a cardboard one) is undoubtedly a kind

of immutable quadrilateral: rectangles conform to all the criteria one could write down to characterise quadrilaterals. But a mutable Rectangle (think of four telescopic radio aerials welded at right angles) is *not* a kind of mutable Quadrilateral: you would expect to be able to bend the hinges of the latter, and stretch its edges asymmetrically. So whilst in the world of immutable values, rectangles \sqsubseteq quadrilaterals, the same is not true of their mutable counterparts. The Z approach works well when considering the behaviour of a community of objects all of whose specifications are known to the designer: but for re-use of code, we require a clear separation of the concerns of clients and providers, and so the VDM interpretation is more appropriate.

In an applicative world dealing with immutable values, a subtype is a subset of the value space, and can always be obtained by strengthening the constraints. When dealing with objects, the important characteristics are not the static properties of the object in a particular state, but the set of possible histories — sequences of states — through which an object's operations could take it. Viewed in this light, a subtype is still a subset: but a subset of the possible histories, not of any individual state. To facilitate re-use of program components dealing with mutable objects, we must adopt this notion of subtyping.

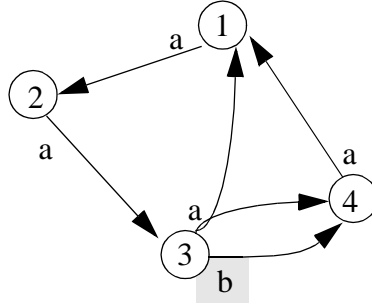
The Z-tradition interpretation works well for subtyping of values: a stronger invariant reduces the state space to a subset. The VDM-tradition interpretation works well for subtyping of objects: strengthening the invariant may lead to an unimplementable type, since the real emphasis is on the opsspecs.

3-4.5.2 Reducing nondeterminism

Strengthening an invariant *is* valid where its only effect is to cut down non-determinacy: in that case, all the operations still have ways of satisfying the clients' expectations.

Cycle-B	
a :	() Δ
b :	() Δ
x :	Nat
{ x < 3 :- x = \bar{x} + 1 } a	
{ x = 3 :- x = 4 \vee x = 1 } a	
{ x = 3 :- x = 4 } b	
{ x = 4 :- x = 1 } a	

Two types are illustrated above, **Cycle** (ignore the shaded parts) and **Cycle-B** (with the shaded parts). **Cycle-B** is clearly a subtype of **Cycle**, since it is the same but for an extra theorem. They have this state diagram:

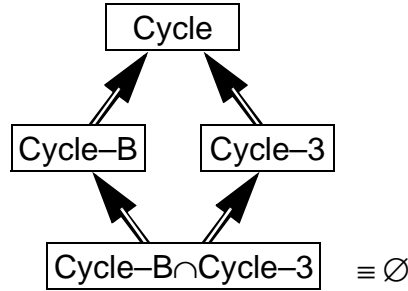


The **a** exit from state 3 is not determined by **Cycle**. If we forget **b** for a moment, then an acceptable refinement of **Cycle** would be to remove state 4, either by altering the opspects of **a**, or by adding a new invariant:

Cycle3 ::+ Cycle
$\{ x < 4 \}$

Clients of **Cycle** have to allow for the possibility that the result of **a** could sometimes be 4, but they may never rely on the appearance of state 4. Clients of **Cycle** can therefore operate successfully with objects which are actually members of **Cycle3**.

But if we recall **b**, the same new invariant added to **Cycle-B** would render the resulting **Cycle-B** \cap **Cycle3** unimplementable: there is no object which could both satisfy the opspect on **b** and keep $x < 4$.



What makes the difference between an invariant which reduces nondeterminacy, and one which goes beyond that and introduces an inconsistency? The former satisfies the following for each opspect $\{ P(\sigma) :- R(\bar{\sigma}, \sigma) \}$ op which is an axiom of the type:

$$\text{nmt:} \quad \forall \sigma:T \cdot P(\sigma) \wedge \text{inv}_i(\sigma) \Rightarrow \exists \sigma':T \cdot R(\sigma, \sigma') \wedge \text{inv}_i(\sigma')$$

where inv_i is the conjunct of all of the type's invariants. (This allows for initialisation into any state, though strictly we need only deal with those accessible by closure over all sequences of operations from a permitted set of initial states.) This may be shown by induction over the object's model: it helps if an enumeration function has been defined which generates all valid states.

It may not always be useful to demonstrate **nmt**, since the appropriateness of an invariant will become apparent when implementation is attempted. Furthermore, its

satisfaction is often obvious. However, if there are to be several following stages of refinement from a complex invariant, it would be wise to check early on.

The Fresco environment should provide for the gathering of the invariants and generation of `nmt`, and its incremental proof after a new invariant is added. It should ensure that users are reminded that new operations need to be checked against existing invariants, as well as the other way around. (See chapter 7.)

3-4.5.3 Values can be range-restricted

Having made plain the pitfalls in attempting to restrict the range of states into which an object can get, it is worth re-emphasising that there is nothing wrong with forming subranges of immutable values (or objects which serve to represent them). This is because any individual member of the type (e.g. 7) is not expected by any client to be able to mutate into any other member. Certainly, there are operations which yield results which are outside the restricted range, but that is not a problem: they are different objects. So for example, if we extract the typing constraint on `Integer::succ`:

Integer
<code>succ(self) ∈ Integer</code>

Now define a restricted range:

OctDigits
<code>self ∈ Integer</code> <code>self < 8</code>

This does not change the fact that `succ(7)` is an `Integer`. 7 is a member both of `Integer` and of `OctDigits`, and `succ(7)` happens not to be. By contrast, if we had defined a mutable `Cell` containing an integer, with an increment operation, then a restricted version `OctCell` would not be a subtype of `Cell`.

3-4.6 Generic definitions

Modules may be derived by supplying arguments to parameterised definitions. The interesting issues are

- what to parameterise — types, classes, or larger modules
- how monotonicity is achieved
- the constraints on the type parameters

3-4.6.1 Subtyping amongst generics

It is worth noting that even though `Nat` is a subtype of `Int`, `Set(Nat)` is not a subtype of `Set(Int)`: you would expect to be able to add new `Int`s to the latter. However, it is possible to define a generic type `Container` such that every `Set(T)` is a subtype of every `Container(T)`.

3-4.6.2 Constraints on type parameters

`SortedList(T)` requires that `T` should have an ordering relation. A type may be defined which has an ordering relation and nothing else:

Ordered
$_ \leq _ : \text{Ordered} \times \text{Ordered} \rightarrow \text{Boolean}$ <i>...required properties of \leq ...</i>

but the requirement of `T` is not that it is a subtype of `Ordered`: we would not wish to exclude `Int` because it is unable to compare itself with all other `Ordered`s, such as `String`. Instead, there is a substitution relationship between this and the types we wish to admit: Palsberg and Schwarzbach [PS91] elaborate on such a scheme.

Abel and OOZE use their separate “property” or “theory” modules as constraints on generic type parameters. The properties are applied via a morphism or “view” which identifies, say `Int` with every occurrence of `Ordered` within the property module. In OOZE, both generics and their arguments are modules, not types. This means that every occurrence of the type-name `Ordered` in a module would be replaced with the argument type.

It is interesting to observe that a variant of the OOZE approach works well if sub-typing were defined between modules rather than types. Suppose modules are defined like multi-sorted ADTs: several types, perhaps with internal model variables, may be defined in a module; and operations are not attached to any specific type, but to modules. A module `MM` is a submodule of another `M` iff all members of its types behave according to the axioms of `M`. Now a rule like “ $x \in T_1 \wedge y \in T_2 \Rightarrow \text{op}(x,y) \in T_3$ ” should be interpreted as follows: for any refinement `MM` of `M`, there will be types `MM::TTn` corresponding to the types `M::Tn`; in `MM`, the axiom of `M` hold, with `MM::TTn` substituted for `M::Tn`; for any type `Tx` not bound within `M`, the axioms in `MM` use `Tx` unsubstituted. The same substitution rules apply to constants. So, for example, we may write rules about the interaction between models, views and controllers, without any hassle about saying that `SquareViews` are not expected to have to interact with `RoundViews`. Where we do mean `M::Tn` to remain unadulterated, we can write it that way explicitly. In this scheme, it is easy to write `Ordered` as a module, of which any `SortedList`’s argument should be a submodule. This reduces the concept-count a little.

The Fresco solution is detailed in §6-4 — p.94.

3-4.7 Assertions and proofs

3-4.7.1 Wide-spectrum imperative OO languages

Eiffel uses programming language for its assertions. There are several advantages:

- Assertions can be executed for debugging purposes. (Fanatics of verification do not believe in “**while** test fails **do** fiddle with code” approach; nevertheless, it is much cheaper to discover bugs by testing than by attempts at proofs; and since bugs are likely, it makes sense to begin by testing — verify once you think you’ve got it right.)
- Programmers only have to learn one language.

- The fundamental concepts of the languages used for analysis, design and coding are integrated, making it unnecessary to construct and continually trip over complex models of (say) pointers.
- The formal process can be taken right down to the code.
- As the library of re-usable modules grows, it extends the power not only of the implementor, but of the specification-writer too.

But the language must be designed to be wide-spectrum. In Eiffel, the expressiveness of assertions is severely restricted by the absence of quantifiers.

There are also semantic difficulties: what does an assertion mean if it has side-effects? Eiffel has an insecure partitioning between procedures (with side-effects) and functions (without). Abel manages to achieve this partitioning strictly. In Fresco, expressions with side-effects are allowed in assertions, as there is no intention to provide any facility for executing them. Even so, an assertion with more than one subexpression with side-effects may be ambiguous: but that is just as it is in programming languages. It is the specifier's responsibility to write meaningful specifications.

3-4.7.2 Expressiveness of the specification component

The purpose of a supertype is to define just those properties on which clients will depend, minimising communication between modules. Since polymorphic code is to be constructed, it must be possible to write partial specifications. General relations are therefore more appropriate than equations: so the model-oriented methods seem better suited to specifications of OO modules.

Since we already have a programming language, there seems no point in having executable specifications, so an equational interpreter such as OOZE's would be superfluous in Fresco.

3-4.7.3 Verification

The proof system should be one in which both refinement of data types and decomposition of method code can be combined with the generation of a proof, in stepwise fashion. Morgan's specification statements seem ideal for this. They are also straightforward to compose when generating subtypes.

Where code is inherited, the necessity for re-verification should be minimised.

The proof methods and tools should permit proofs to be outlined and subgoals justified informally where appropriate. The term-rewriting systems which fit the equational systems best (such as Abel and OOZE) are less well suited to this, as subgoals are less easy to isolate.

The B tool [Sorenson91] has been applied successfully to object-oriented versions of Z, and Mural [Mural91] to VDM. It is Mural that is the basis of Fresco's proof assistant.

3-4.8 Larger units of design

The purpose of encapsulation is to ensure that one part of a design is not spuriously dependent upon another: this will help limit the complexity of design decisions, keep manageable the ramifications of change, and increase the portability of individual chunks of design effort.

In some OO languages, encapsulation is applied on a per-object basis, and in others on a per-class basis. In the former regime (exemplified by Smalltalk), a method may only access the inner structure of its receiver; in the latter (e.g. C++), a method may access the innards of any object which is a member of its home class (that is, the class of the receiver).

In OOP, the unit of design-effort is not the object, but the class or groups of interrelated classes. It is therefore arguably unnecessary to encapsulate per-object — Szyperski’s “no paranoia” principle [Szyperski92]. If I am going to re-implement a class, then I will re-implement all the objects of that class, and since I know what is inside any object within that class, there is no point in hiding it from myself.

Modularity is about:

- the division of design-effort into *manageable* pieces (both from the point of view of the initial effort, and subsequent reading and modification)
- the division of designs into separately *portable* chunks.

[Wills91] and [Szyperski92] propose that the two purposes of classes should be separated: the class should be a set of objects with a common implementation, whilst encapsulation should reside in an orthogonal modularising concept, called the “capsule” in Fresco (§7). In practice, it is rare to incorporate a separable feature or subsystem in a single class. The statistical functions form a unit of design effort which should augment an existing **Number** class, rather than forming a new class **NumbersWithStats** — we want them to work for all the existing subclasses of **Number**. A subsystem which supervises the presentation of objects to the user (such as Smalltalk’s MVC) is designed as a *framework* of interacting classes, with defined *contracts* between them [HHG90].

3-4.9 Programming language issues

Smalltalk was chosen as the initial basis for the Fresco language:

- The entire lack of any “type-checking” (signature checking) leaves this as a part of verification. The rules of any prescribed system (such as those of C++) might conflict with those derived from the verification rules. (C++ allows dynamic binding to be turned off for chosen selectors, so that speed can be improved by determining the methods that will be called by each message at compile time. This is not OO, of course. [JGZ 88] and subsequent work have shown how to do such determinations automatically; though at length and across encapsulation boundaries.)
- Persistence of data through a software change: essential to responsive prototyping, experimental development and frequent delivery of updates. Smalltalk and some of the newer OO databases do this.
- Smalltalk’s minimal language with generous predefined kernel of classes, and the ability to compose a system by ‘filing in’ software updates, form an excellent model upon which to build the Fresco capsule system.
- Similarly, the class browsing facilities, which can readily be extended to deal with type specifications.

(Smalltalk was also chosen as the implementation language, since Mural was built in it and again because of its ready modifiability.)

3-4.10 Semantics

[Wolcko] describes a semantics for Smalltalk in denotational terms. It would clearly improve the foundation of the present work to relate it to some such theory. The present Fresco semantics, in the practical form of a set of proof rules, is open to inconsistencies; however, it does make it easy to omit the difficult parts, such as some of the more unpleasant uses of Blocks.

Utting and Robinson provide a semantics for their work in terms of object histories, and a minimal similar model is used here.

3-5 How Fresco tackles these issues

This is, of course, the subject of the rest of this thesis; but to summarise:

- Fresco method-specifications are like those of VDM:
 - a client is guaranteed that if the precondition is met, the operation will succeed
 - postconditions are relations on the before & after states
- Constructed subtypes always inherit their ancestors' specifications, which conjoin with any they have in their own right.
- Conjunction of specifications is such that a proof based on any conjunct is always valid for the conjunction; this applies in particular to subtypes, so there is no "yo-yo problem".
- The semantics is based on object histories: subtypes are subclasses of possible histories. So subtyping is about true substitutability.
- The language is a wide-spectrum extension of Smalltalk (with an adapted concrete syntax). Smalltalk's clear distinction between identity and equality is preserved. A scheme for dealing with aliasing is provided.
- Modules (called capsules) and classes are orthogonal. The capsule forms the main unit of knowledge.
- Verification can proceed stepwise with program development, and steps at every level in the natural-deduction-style proofs can be left informally justified. The aim is 'rigorous' proof, in which every informal justification could be rendered formal, given the motivation and effort.

3-6 Summary

A number of terms have been defined: in particular, local meanings of *type*, *class*, *abstract class* have been distinguished, together with *inheritance*, *subclass*, *subtype*.

The key issues to be addressed here have been described. *Concurrency* will be avoided. Software engineering considerations motivate a concern with reliable *polymorphism* in parametric and inclusion forms, and *monotonic system construction*. The orthogonality of system-design modules and classes will be addressed. Fresco

should be integrable with popular methods of OO analysis and design, though that integration is not part of the present work.

The virtues and drawbacks of other systems have been surveyed. The proof system and tools provided by Mural seem at least as applicable as others. Of logical foundations, LPF deals most easily with the sometimes undefined propositions that arise in programming. In object-oriented program specification, algebraic methods tend to be too deterministic, whilst current derivatives of the widely popular Z seem to be converging with VDM in many respects. The Smalltalk language and programming environment is a very flexible and minimal *tabula rasa* upon which to build an experimental system such as Fresco.

The selection of VDM, Mural and Smalltalk as the bases for Fresco could have been founded on careful reasoning about their virtues in relation to the goals in mind; but that would be a gross post-rationalisation. However, the same reasoning applies to being pleased in retrospect that these choices were imposed by circumstances.

4 Theories and proofs

This chapter describes the proof system upon which support for formal methods in Fresco is founded. It is an adaptation of the Mural system [Mural].

A model of the proof system is presented here in the form of a hierarchy of types, as outlined in the chapter 2.

4-1 Theories

Theories are the basic portable units of knowledge: types, capsules, and proofs are all varieties of **Theory**.

Theory	
var label \in	Symbol
var symbols \in	Binder set
var theorems \in	Theorem set
var axioms \in	Theorem set
var knownSymbols \in	Binding set
var knownThms \in	Theorem set
fn context \in	Context set
$\text{axioms} \subseteq \text{theorems} \wedge \forall a \in \text{theorems} \cdot a \in \text{axioms} \Leftrightarrow a.\text{justification} = \text{nil}$	
$\text{context} \in \text{Theory set}$	
$\forall t \in \text{theorems} \cdot t.\text{context} = \{\text{self}\}$	
$\text{knownSymbols} = (\bigcup \text{context}).\text{knownSymbols} \cup \text{symbols}$	

A theory is a set of declared symbols, and a set of axioms over those symbols; and the body of other theorems which are derivable from those axioms¹. For example, the theory of propositional logic with its symbols **True**, **False**, \wedge , \vee , \neg and the usual axioms (including proof rules) over them. There will in general be an infinity of derivable theorems, and so it is those which the user has explicitly discovered which are recorded in a Fresco theory, together with the justifications for believing in their membership. This is not necessarily a subset of the set of true theorems, since justifications may be partly informal and therefore wrong. The proof tool tracks dependencies between theorems and can highlight the transitive dependency of any proof on an incompletely justified theorem.

Most useful theories *inherit* the symbols and axioms (and therefore theorems) of other theories, which form its *context*: for example, predicate calculus is built upon propositional logic, and number theory may be built upon set theory. Contexts must

1. The axioms are not constrained to be equations, nor even propositions, since at the most basic level, there is no interpretation of the theorems. All we know at this level is that theorems can be derived from other theorems.

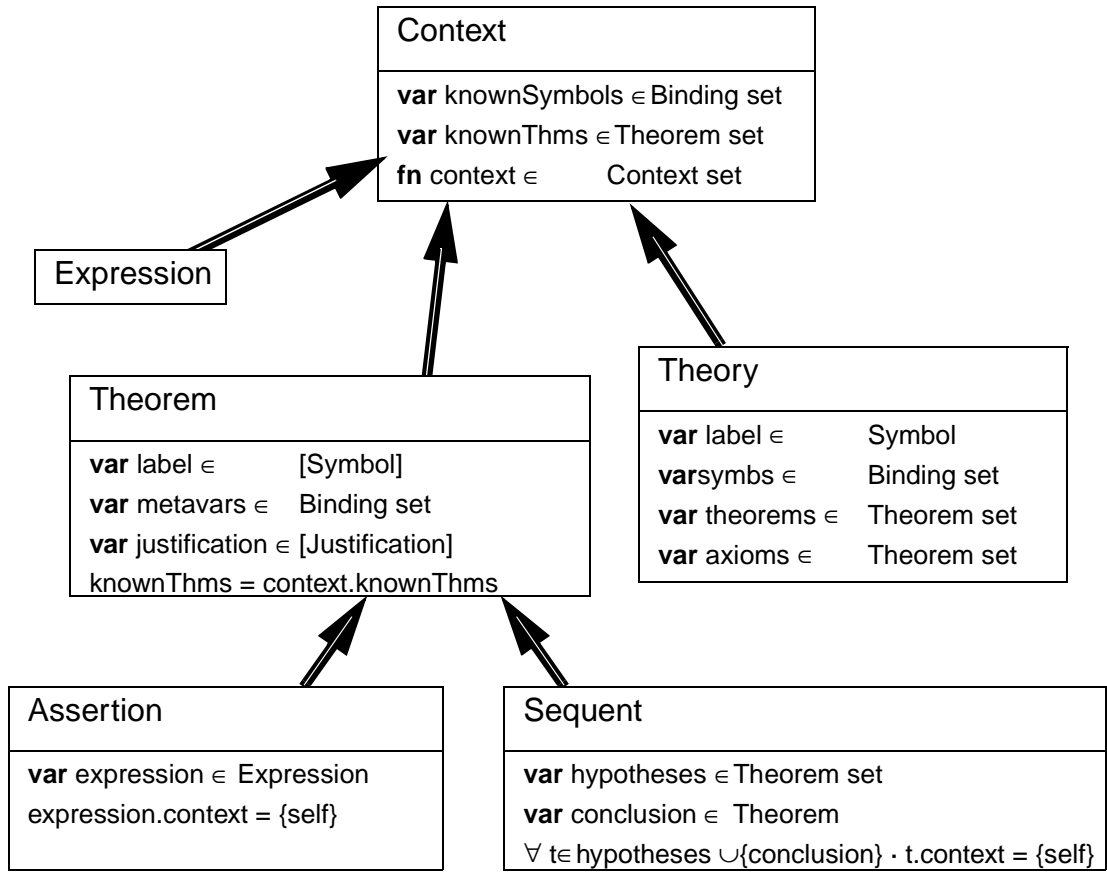


Fig. 5. Theories, theorems, and proofs

form an acyclic graph. There is no guarantee that a Theory will not be vacuous, since it is perfectly possible to define conflicting axioms, either directly or by inheritance.

The theorems of a theory may refer to any of the **knownSymbols**, which include both those bound in the local theory and those inherited from the contextual theories. Symbols with the same signature from different theories are identified, so the theorems from different theories effectively conjoin.

4-1.1 Symbols

All expressions in Fresco are built out of Symbols declared in Theories, and the spelling, arity and kind of the Symbols determine how. These attributes, its *signature*, are fixed in a Symbol's binding declaration in a Theory or Theorem.

Binding	
var spelling \in	Symbol
var arity \in	Kind mapTo: Nat
var kind \in	Kind

Fresco has five kinds of Symbol (to date).

- $\forall, _+ _$ Object-symbols represent pure expressions (such as parameter or variable names). Written in lower case; and operators.
- T** Type-Symbols represent types. Upper case.
- $\forall, \exists, \{ \dots \}$ Binder-symbols, which bind variables locally within an expres-

sion.

- E Expression-symbols, standing for any code expression which may have side-effects. Upper case.
- A_i Expression-list-symbols, standing for any complete argument list. Upper case with subscript.

Chapters 5 and 6 elaborate these distinctions and demonstrate the different uses.

The arity of a symbol determines how many arguments it may take of each kind — so an arity of $\{\text{ObjectK} \rightarrow 1\}$ — let's take 0 as the default — would be a unary function, whilst an arity of $\{\}$ would be a constant.

The spelling of each Symbol is unique within its declaring Theory or Theorem. It defines its concrete syntax: if it is declared just as a name but has an arity of $\{X \rightarrow n\}$ then it is written with round or square parenthesis — like $s(a, b, c)$ or $P[a, b, c]$. By convention, type-symbols and expression-symbols are spelled with initial capitals; object-symbols aren't.

The full declaration syntax of a symbol will be

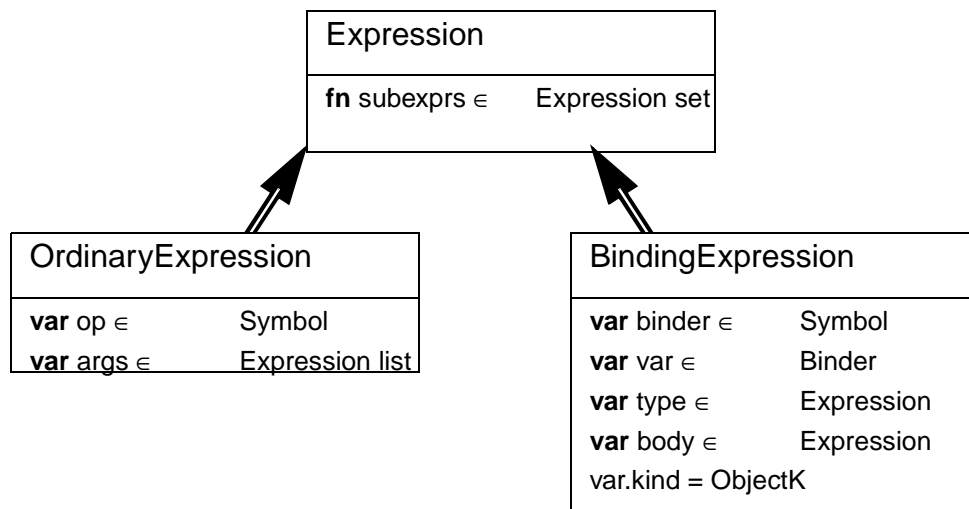
Decl ::= *spelling* [: *Kind* { *Arity* }]

Kind ::= ObjectK / ExpK / ExpLK / TypeK / BinderK

(If the kind is omitted, the default arity is $\{0, 0\}$ and the default kind is inferred from the spelling and the usage.)

4-1.2 Expressions

An expression is an instance of a symbol known in its context: that is, all those bound by a containing Theorem or Theory. There are two kinds of expression:



4-1.2.1 Operator and message expressions

The concrete syntax of an expression is determined by the declared spelling within certain limits. There are seven (!) variations of concrete syntax for expressions; the first five are for OrdinaryExpressions:

$\text{ZeroArity} ::= \text{Symbol}$
 $\text{MixFix} ::= \text{Receiver } \text{selPart1: } \text{arg1} [\text{selPart2: } \text{arg2} \dots]$
 $\text{PostFix} ::= \text{Receiver} . \text{Symbol} [(\text{argument} [, \dots])]$
 $\text{BinaryOperation} ::= \text{Receiver OpSymbol Argument}$
 $\text{UnaryOperation} ::= \text{OpSymbol Receiver}$

Binary and unary operations are reserved for symbols which are not formed from alphanumeric characters, for which users may define syntactic precedence (in the capsule in which that spelling is first used). The kernel definitions include symbols for the usual operators for predicate calculus, sets, maps, and lists.

These are examples of expressions:

```

21          ab
ab.adjust   ab.add(21, x.result)
z>42+i.raised(2) whileTrue: [z.reduce(i)]

```

The MixFix syntax permits Smalltalk-like expressions to be written. The PostFix syntax is used in the kernel rules which define the semantics of the language, for matching general operations.

4-1.2.2 Binding-expressions

A binding expression, such as

$$\forall x \cdot f(x)$$

declares a $\{ \}$ -arity symbol which may be used in its body. It must define a type for the variable. The signature of a binder-symbol is therefore always

$$\text{BinderK } \{ \text{TypeK} \rightarrow 1, \text{ObjectK} \rightarrow 1 \}$$

In the design of Mural, there was some debate about whether it would be useful to loosen this to allow binders such as `let`, which binds to a specific value rather than a type; however, this is not tackled here. The common binders such as \forall , \exists , $\{ \dots \}$ are covered.

The variable bound by a binder must always be an object-symbol of arity $\{ \}$: that is, binders may not define types, other binders, metavariables, or operations or functions.

4-1.2.3 Blocks

A block represents a parameterised segment of code, written:

$$[: \text{parm1} : \text{parm2} \mid \text{code}]$$

The semantics of Smalltalk blocks is complex and not dealt with in full here. However, rules can be written covering its uses in particular ways: for example, conditionals and loops are designed as higher-order operations taking blocks as parameters — for example,

$$\text{aCollection do: } [: \text{item} \mid \text{codePerItem}]$$

4-1.3 Theorems

Theorem	
var label \in	[Symbol]
var metavariables \in	Binder set
var justification \in	[Justification]
knownVars = (\bigcup context).knownVars \cup metaVars	
knownThms = (\bigcup context).knownThms	

A theorem is a statement which, given some interpretation of its symbols, represents some fact. An axiom is a theorem, in a context in which it is assumed without needing justification. In Mural and in Fresco, proof rules are identified with theorems: new rules can be derived by proving theorems. The term ‘theorem’ will henceforth include ‘proof rule’.

Theorems come in two varieties: Assertions and Sequents. An Assertion is an expression of some sort; a Sequent is a conditional assertion. Both forms may have local *metavariables*, and the expressions are formed from these and symbols defined in the context.

An Assertion is written

$$[\text{label} :] [\text{Decl} , \dots] \text{Expression}$$

A Sequent is written in either of two equivalent forms:

$$[\text{label} :] [\text{Decl} , \dots] \text{hypothesis} [, \dots] \vdash \text{conclusion}$$

or alternatively,

$$[\text{label} :] [\text{Decl} , \dots] \frac{\text{hypothesis, hypothesis} \quad \text{hypothesis, ...}}{\text{conclusion}}$$

where the hypotheses and conclusion are Theorems.

Theorems are generic statements, valid for all well-formed consistent substitutions of Expressions for each metavariable.

A Sequent states that in any interpretation and context in which the hypotheses are known to be valid, then the conclusion is also valid. The interpretation of any Theorem is such that it retains its validity whenever its metavariables are replaced by expressions of an appropriate kind.

A theorem may appear as a conclusion: $A \vdash (B \vdash C)$. If A is proven, then in any context in which B is also proven, C may be inferred. This amounts to the same as saying that in any context in which A and B are both proven, then C may be inferred; so we have the general rule

$$\text{decurry-inf: } A, B, C \cdot (A \vdash (B \vdash C)) \vdash (A, B \vdash C)$$

A theorem may appear as a hypothesis: $(A \vdash B) \vdash C$. This says that C may be inferred in any context in which the additional assumption of A will prove B . For example:

arith-induct: $i, P \cdot i \in \text{Integer}, P[0], (j \cdot j \in \text{Integer}, P[j] \vdash P[j+1]) \vdash P[i]$

4-1.3.1 Substitution

The metavariables are substituted by appropriate expressions in order to apply the theorem to particular situations. For example, in this theorem, metavariables x and y stand for any values:

add-comm: $x, y \cdot x \in \text{Integer}, y \in \text{Integer} \vdash x+y = y+x$

(Sometimes the metavariable declaration clause will be omitted to reduce clutter; let us use the convention that any variables not known in the environment and free in a theorem are its metavariables, where this is obvious.)

An expression metavariable may be parameterised (with square brackets); for example:

univ-elim: $n, P:\text{ExpK}\{\text{ObjK} \rightarrow 1\} \cdot \forall i \cdot P[i] \vdash P[n]$

$P[i]$ stands for any expression in which i occurs; $P[n]$ represents the same expression, with n appearing in place of i .

$P[i]$ does not specifically match a function call. For example, $P[2]$ matches $\text{sqrt}(2)$, with $\{P[z] \rightarrow \text{sqrt}(z)\}$; but it also matches $5 \times 2 + 3$ with $\{P[z] \rightarrow 5 \times z + 3\}$.

Notice that matching does not depend on judgements of equality: it is entirely based on the symbols in the expressions.

Rules about executable code have to distinguish between expressions which may have side-effects, and those which definitely do not. For example, it may be that $x \text{ add: } y$ adds the value of y to x ; but only if the expression we substitute for y does not itself add something to x — $x \text{ add: } (x \text{ add: } 2)$ might have a rather different effect. Object-symbols (written in lower case) may be used as metavariables to represent pure expressions (so the example substitution would be invalid), whilst expression-symbols (written in upper case) match expressions with possible side-effects.

The interpretation of the use of an object-metavariable is that it matches only a parameter or variable (in the code), and a rule involving one can only be applied after a notional transformation of a complex expression to a series of assignments. Object-symbols must have arity $\{ \}$ when used as metavariables, again since we do not deal in higher-order matching.

4-1.4 Special constructs for the kernel

The following notations will only be used in defining a few of the kernel rules: they do not seem likely to be needed for ordinary design work, and will not be available to Fresco users in the construction of new definitions.

4-1.4.1 Two-way theorems

The forms $a, b \cdot P1, P2 \vdash Q1, Q2$ or $a, b \cdot \frac{P1, P2}{Q1, Q2}$ will sometimes be a useful abbreviation for the set of theorems

$a, b \cdot P1, P2 \vdash Q1$

$a, b \cdot P1, P2 \vdash Q2$

$a, b \cdot Q1, Q2 \vdash P1$

$a, b \cdot Q1, Q2 \vdash P2$

4-1.4.2 Theorem schemata

$$\bigwedge f \cdot P[f] \quad \text{— for all } f \text{ such that ...}$$

is equivalent to a set of theorems, repeated for the values of f satisfying the side-condition. The purpose is to express behaviour ‘wired in’ to Fresco which cannot be defined in the general notation. It is applicable where there is a finite number of instantiations satisfying the side-condition. For example,

$$\bigwedge v_i \cdot P[v_i] \quad \text{— for every private variable } v_i$$

4-1.5 Contexts

Context
var knownSymbols \in Binding set var knownThms \in Theorem set var context \in Context set var allContext \in Context set

Theorems and theories are all Contexts. Any Context has a set of known symbols which may be used to form expressions within it; and a set of known Theorems to which justifications may refer, provided there are no circularities.

4-2 Proofs

A derived theorem has a *justification*, which supports the belief in its validity. For example:

```

1:      juice  $\vee$  porridge                                // a precursor of 2
2:      porridge  $\vee$  juice                                // derived from 1
      from 1 by or-comm with {A $\rightarrow$ juice, B $\rightarrow$ porridge} // justification

```

The *by...from...[with...]* clause is one form of justification, which may be a match, a subproof, an appeal to an oracle, or an informal text: these are dealt with in the sections below. All justifications refer to a set of precursors — the theorems from which this one is derived. No theorem should be among its own transitive precursors.

A complete proof is a network of intermediate justified theorems generated for the purpose of deriving some goal theorem. A proof may be constructed in any mixture of forwards or backwards modes: ‘forwards’, seeking what theorems can be justified from the theorems known so far; or ‘backwards’, finding what would theorems would be needed to justify the goal required.

4-2.1 Informal and rigorous proofs

An *informal justification* is one taking the form of an intuitively-based argument in natural language for the belief that a theorem holds. The intuitions may be based on the current interpretation of the theory, and may possibly not be provable with the theorems available.

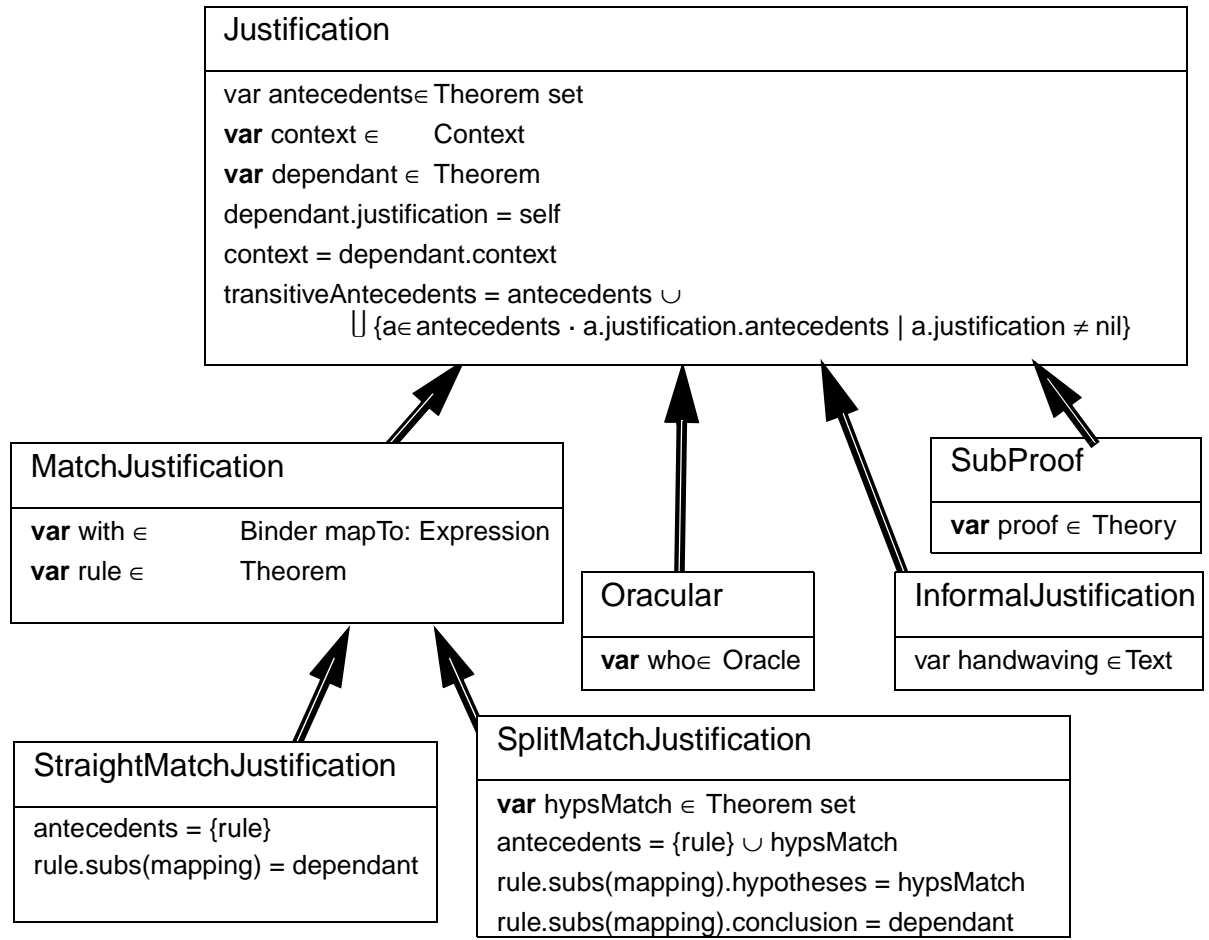


Fig. 6. Justifications and proofs

A *rigorous justification* is an informal justification, in which the intention is that, should there be any query (for example at a design review) about its validity, it could, with sufficient effort, be proven with the theorems available.

A *rigorous proof* is a mixture of rigorous and formal justifications. Most of the proofs done with Fresco should be expected to be of this form. In general, the strategy is to make an outline proof, beginning by stating key theorems, and giving rigorous or formal justifications connecting them ‘forwards’ to the theorem of interest, and ‘backwards’ to axioms or other believed theorems.

4-2.2 Matching

A theorem may be proven by showing that it is equal to the result of substituting for metavariables in an antecedent theorem: this is the Straight Match Justification. E.g.

and-comm: $A, B \vdash A \wedge B \vdash B \wedge A$

\vdash sprouts is orful \wedge peas is nice \vdash peas is nice \wedge sprouts is orful

by and-comm **with** $\{A \rightarrow \text{sprouts is orful}, B \rightarrow \text{peas is nice}\}$

The Split Match Justification supports a match not of the complete rule, but of its conclusion; but only if a match to the hypotheses can be found. Again, metavariables may be substituted consistently. For example,


```

subs-eq:      A, B, P · A=B, P[A] ⊢ P[B]
1:            n = 2
2:            2 × 3 = 6
3:            n × 3 = 6      from 2 by subs-eq with {A→n, B→2, P[i]→i×3=6}

```

(The details of the **with** clause are usually holophrasted.)

The symbols in the range of of the **with** clause are obviously to be interpreted in the context of the Justification, and not of any more local context: this may require some respelling in order to display the result unambiguously. For example,

```

      n, x · ...           // n and x defined and used in this context
1:      x=n
2:      s ∧ ∀ n · n>x ∨ f(x,n)  // local context in here
3:      s ∧ ∀ n' · n'>n ∨ f(n,n') by subs-eq from 1, 2 with {a→x, b→n}

```

4-2.3 Subproofs

SubProof
var proof ∈ Theory proof.axioms = dependant.hypotheses dependant.conclusion ∈ proof.theorems.rng dependant.metavars ⊆ proof.symbols

A sequent may be proven by showing that the conclusion (as a theorem on its own) is provable in a context in which the hypotheses (as theorems on their own) are assumed. It is generally necessary to prove intermediate theorems, rather than getting to the conclusion in one step. The context in which the steps are documented is itself a Theory.

The axioms of the theory must be just the hypotheses of the theorem to be justified, and the declarations of the theory should be those of the theorem. (Except that local names which are not metavariables of the theorem may be declared, and extra axioms may be introduced, provided they do no more than define values for these local names. This constraint can only practically be enforced by restricting such axioms to the form *new-symbol* = *expression* .)

Each theorem which appears as a step in a proof is annotated with a justification which supports its validity. The most common justification is the split match, instantiating a theorem from the context, such that the hypotheses match named antecedents selected from prior steps in the proof or theorems in the environment.

The example below also demonstrates a nested **NatProof** (lines labelled 4).

The concrete syntax of a proof displays the steps (that is, theorems in the internal context of the proof) linearised with respect to the chain of justification, and with systematic labelling. Justifications are displayed with each step. For example, a proof of **add-comm** consists of the declarations, hypotheses and conclusion of the theorem, separated out into separate lines, together with intermediate theorems:

decl	x, y	
h1	$x \in \text{Integer}$	
h2	$y \in \text{Integer}$	
1	$0+x = x$	add-0-l(h1)
2	$x+0 = x$	add-0-r(h1)
3	$x+0 = 0+x$	subs-eq(1, 2)
4decl	$n \cdot$	
4h1	$x+n = n+x$	
4h2	$n \in \text{Integer}$	
4.1	$(x+n)+1 = (n+x)+1$	determinacy(4h1)
4.2	$x+(n+1) = n+(x+1)$	add-assoc(4.1, h1, 4h2)
4.3	$n+(x+1) = (n+1) + x$	add-defn(h1, 4h2)
4.4	$x+(n+1) = (n+1) + x$	subs-eq(4.2, 4.3)
5	$x+y = y+x$	arith-induct(h2, 3, 4)

The whole proof exists in a context in which the invoked theorems `add-0-l` etc are known. The conclusion of each justifying theorem matches the line it justifies, and the parenthesised labels show which lines are matched to its hypotheses. Hypotheses form part of the context of the proof, and need no justification. The chain of justification may not contain loops. Omitted from the justifications as presented here are the mappings from theorem metavariables to expressions in the application context. For example, if `add-0-l` is defined in the context as $e \cdot e \in \text{Integer} \vdash 0+e = e$, then metavariable `e` is mapped to the expression consisting of the single local variable `x`. The declarations of the variables local to the proof are in the `decl` lines.

4-2.3.1 Proof construction

The order in which a proof is constructed has no relevance to the validity of the final result, and strategies for proof construction are a matter of the detail of the proof construction tool and its user.

As an example of a step in ‘backwards’ mode, in the example above, the instantiation of `arith-induct` to match its conclusion with line 5 gives

$$\begin{array}{l}
 y \in \text{Integer}, \\
 x+0=0+x, \\
 (n \cdot n \in \text{Integer}, x+n=n+x \vdash x+(n+1)=(n+1)+x) \\
 \vdash \quad x+y=y+x
 \end{array}$$

(with $i \rightarrow y$, $P[i] \rightarrow x+i=i+x$) of which we have the first hypothesis, but must still prove the other two.

4-2.4 Oracles

The justification of a theorem may appeal to an oracle — a ‘wired in’ procedure for deciding whether an inference is valid. There are two reasons for providing oracles:

- Fast deterministic application of a set of ordinary theorems — for example, simplification in the propositional calculus, or arithmetic.
- Implementation of fundamental rules which are difficult or impossible to express within the proof system; including:
 - those with special side-conditions;
 - rule schemata which are instantiated appropriately for the context.

4-3 Context operations

4-3.1 Union of contexts

The union of two or more contexts is important for the monotonic composition of specifications:

$$\begin{array}{l} c1, c2 \cdot c1 \in \text{Context}, c2 \in \text{Context}, \\ \{t.\text{label} \mid t \in c1.\text{knownTheorems}\} \cap \{t.\text{label} \mid t \in c2.\text{knownTheorems}\} = \emptyset \\ \vdash \\ (c1 \cup c2).\text{knownSymbols} = c1.\text{knownSymbols} \cup c2.\text{knownSymbols} \wedge \\ (c1 \cup c2).\text{knownTheorems} = c1.\text{knownTheorems} \cup c2.\text{knownTheorems} \end{array}$$

Notice that any name declared in both Contexts is identified: this means that in the union, a quantity represented by some variable can have more strict constraints on it than in either of the origins. The union of contexts with clashing theorem-labels is undefined, but Fresco adds qualifications to theorem-labels where necessary.

When a context is to be extended, for example inside a proof or subproof, the enclosing context is unified with the complete set of the theorems contained within the proof. Each of the theorems is available to each of the others for justification, though an invariant on justified theorems prevents theorems from depending on themselves.

4-3.2 Extraction from context

A theorem which has been proven in a given context may be extracted from that context provided the antecedents on which it depends are attached to it as hypotheses, and the declarations are brought out as metavariables. For example, we could extract 4.2 from the example subproof into a line of its own in the main proof:

$$\begin{array}{l} 6 \quad n \cdot n:\text{Integer}, (x+n)+1 = (n+x)+1 \vdash x+(n+1) = n+(x+1) \\ \text{add-assoc}(h1, 4h2) \end{array}$$

or we could go further, noting that 4.1 depends in turn on 4h1:

$$\begin{array}{l} 6 \quad n \cdot n:\text{Integer}, x+n = n+x \vdash x+(n+1) = n+(x+1) \\ \text{determinacy, add-assoc}(h1) \end{array}$$

We can then extract the line from the context of the proof:

$$\begin{array}{l} x, n \cdot x:\text{Integer}, n:\text{Integer}, x+n = n+x \vdash x+(n+1) = n+(x+1) \\ \text{determinacy, add-assoc} \end{array}$$

At each stage, we have carried along the justifications which tell us what remains to be prepended to the hypotheses. We are finally left with the named rules which link the steps in the original proof. These can also be prepended: for example,

$x, n \cdot (a, b, P \cdot a = b \vdash P[a] = P[b]), x:\text{Integer}, n:\text{Integer},$
 $x+n = n+x \vdash x+(n+1) = n+(x+1)$

add-assoc

4-4 Comparison with Mural's proof system

4-4.1 Theories and proofs

Chapter 4 of the Mural Book [Mural] elaborates on the foundations of the proof system and the reasoning leading to the choices which were made from it. Fresco builds on that work, and adapts it in some respects:

- Mural's separate sequents and theorems are unified in Fresco. This was given some consideration in Mural but avoided because it would be slightly less general [Mural p125]. This simplification effectively means building-in the rule:

$$a \cdot \frac{a \in A \quad x \cdot x \in A \vdash P[x]}{P[a]}$$

- Proofs and Theories are identified in Fresco. This is simply a conceptual economy. The number of different kinds of variable is thereby reduced.
- There is a distinction between metavariables which match general expressions, and metavariables which match pure expressions (such as single parameter or variable names).
- Mural distinguishes at the syntactic level expressions which yield types and expressions which yield type-members. This is not done here, since this means much duplication in the presentation. Instead, constraints on the results of the expressions are used where necessary.

[Mural] goes into considerable detail about the properties of the proof system: it would be unprofitable to repeat that here.

Fresco takes over much of Mural's standard population of theories (Fig. 7.) (although there is some mechanical translation to do).

4-4.2 Logic

VDM is founded on LPF (§3-1.6 — p.37). Axioms for LPF and for the use of equality are defined in Mural: *Propositional LPF* declares symbols **true**, \neg , \vee , and defines in terms of them \Leftrightarrow , \Rightarrow , **false**, and \wedge . LPF is designed to deal with the possibility of the falsity of the conventional axiom $e \vee \neg e$. The other axioms follow conventional logic.

4-5 Summary

Knowledge which can be applied to reasoning is represented in theories. A theory is a collection of symbol-declarations and theorems about those symbols. A set of

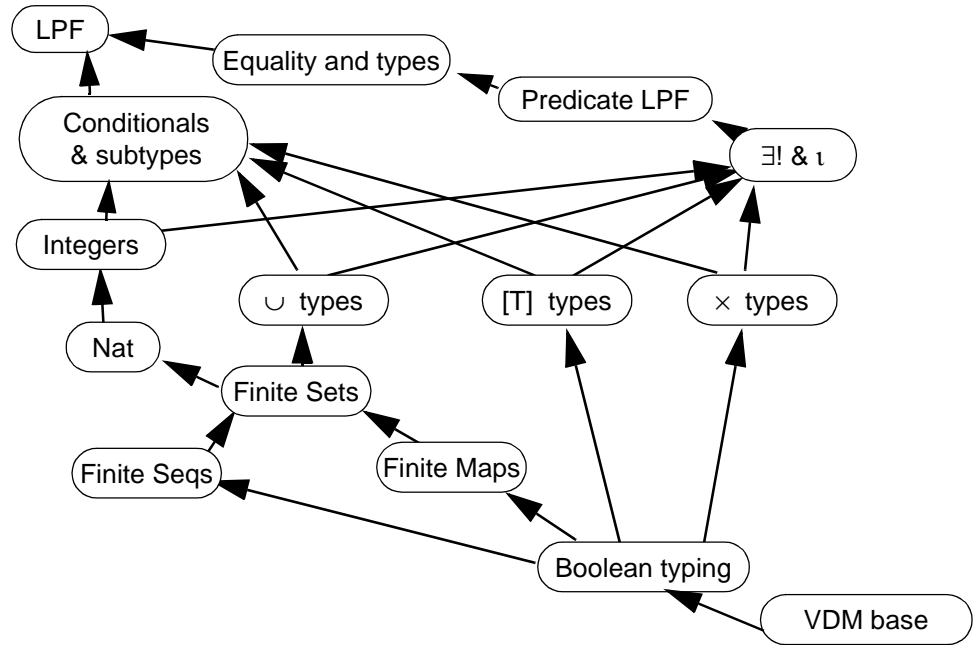


Fig. 7. The standard Mural basis of theories for VDM

known or assumed theorems may be used to support a further theorem, by one of a few fixed justification schemes. A theory may be defined wholly or partly by inheritance from one or more others.

Formal justification is based on the idea of specialising a theorem by substituting for its metavariables; a theorem states that its conclusion is valid if its hypotheses are valid. Sequents are proven by constructing a theory in which the hypotheses are the only axioms, and the conclusion appears as a theorem.

A theorem may be removed from its context by prepending as extra hypotheses a set of theorems from which it can be derived.

This chapter has described the foundations of Fresco's reasoning system. The next three will show how it is used to specify and verify methods, classes and capsules.

5 Statements and Assertions

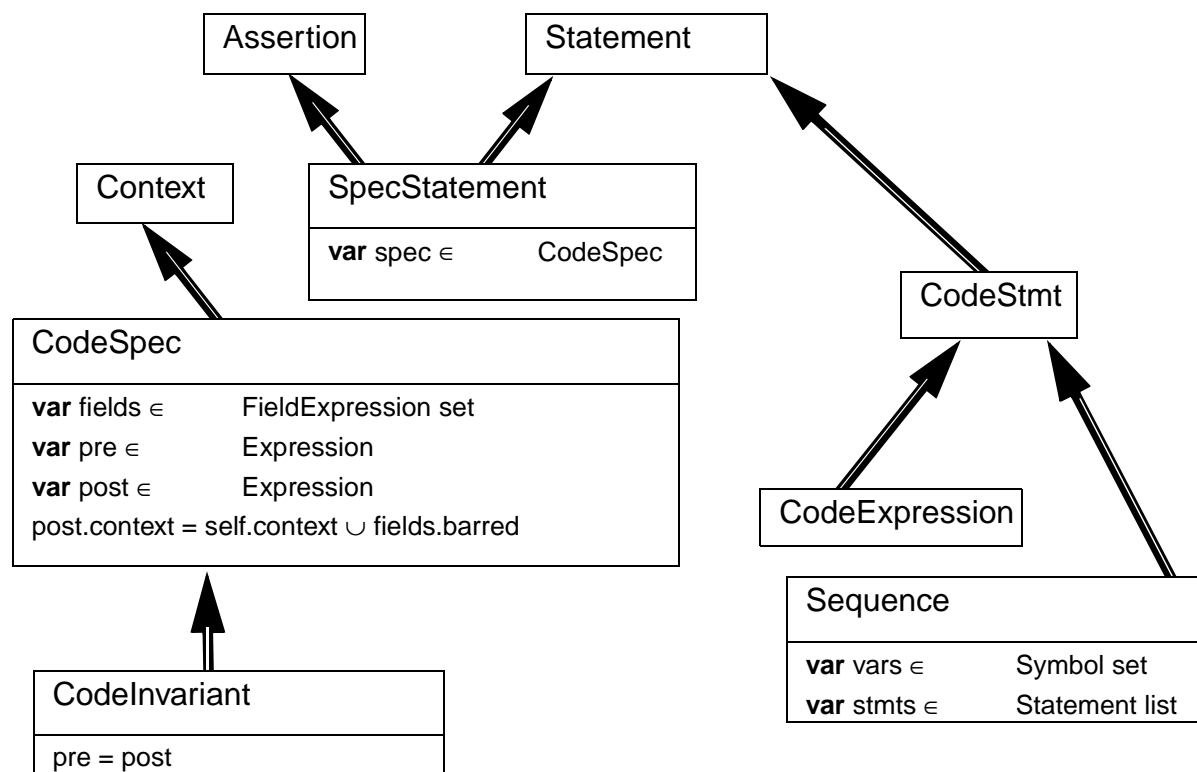
5-1 Specifications and code

Statements represent the executable part of the Fresco language; **assertions** are the not always executable constructs from which theorems are built (See Fig. 8.) The two languages overlap considerably: many expressions (e.g. $2+3$) are both statements (i.e. executable) and assertions (i.e. OK to use in a theorem). The **SpecStatement** links the two: each SpecStatement is an assertion that a particular statement conforms to a particular **CodeSpecification**. Whilst a Statement determines a relationship between successive pairs of states by prescribing how the machine shall achieve the transition, the CodeSpecification expresses this as a predicate over components of the two states.

5-1.1 Code

The executable statements of Fresco are essentially those of Smalltalk dressed up in a concrete syntax more convenient for integration with specification constructs. They are an executable variant of the Expressions, together with Sequences;

Fig. 8. Statements



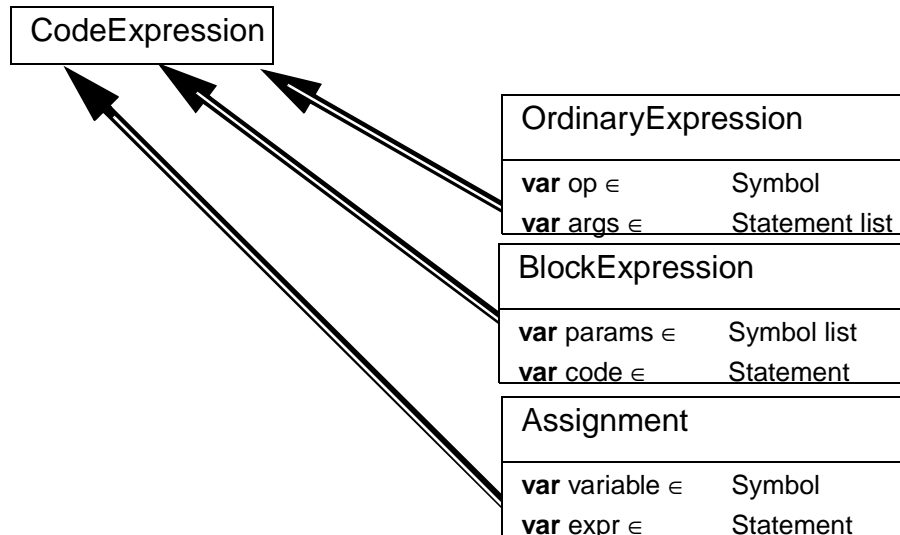
loops, conditionals and other control constructions are formed from calls to higher-order operations.

A sequence may have local variables. There is no special provision in the syntax for imposing a type constraint on a variable; invariants inserted in the code may make assertions about the type of a variable, but that is a matter for proof rather than mechanical type-checking.

Sequence ::= ([**var** *Symbol* [...] ·] *Statement* [; ...])

CodeExpression ::= *Assignment* | *Block* | *OrdinaryExpression*

Block ::= [[:*Symbol* [...] |] *Statement*]



5-1.2 Specification Statements

A spec-statement is an *Assertion*, and may therefore appear in a theorem or a proof; and it may also be a statement within code. The general form is:

$\{ \textit{spec} \} \textit{code}$

If embedded in a method, the spec-statement is executed by executing its *code* part. *Code* may itself be a specification-statement: read $\{S1\} \{S2\} C$ as $\{S1\} (\{S2\} C)$.

The statement asserts that the *code* behaves according to the *spec*. It is for the designer to ensure that this is so: that is, the theorem must be proven valid within the context in which it is stated. In the Fresco support environment, each such statement has an attached proof, which the designer must complete with the assistance of the proof tool: Fresco ensures that all such proofs are complete before a capsule may be certified. (Capsule certification is discussed in Chapter 7 – *System composition* (p.110).)

Although several code-specs may apply to the same piece of code, clients may assume that anything that may be inferred from a spec-statement will not be negated by another found somewhere else: spec-statements compose monotonically. However, one must take care not to imagine that anything not specified doesn't happen.

The axioms of the type which a class is intended to implement must be observed by its operations: and so the outermost block of statements of every method forms a specification-statement whose specification-part is the conjunction of all axioms

applicable to that method. (Alternatively, there may be separate proofs for each relevant axiom.) Proof of this outer specification-statement proceeds by recursive decomposition using rules like **seq** above, in the style of [Morgan90].

5-1.2.1 Code invariants

$$\{ \text{inv} \} \text{code}$$

This asserts that *if* *inv* is true before executing *code*, then it will also be true after. (If the execution of *inv* would alter the state of the system, this has no effect on the meaning of the *code*.)

E.g.

$$\begin{array}{l} x, y, \text{sum} \cdot \\ \{ x \in \text{Int} \wedge y \in \text{Int} \wedge \text{sum} \in \text{Int} \wedge x+y = \text{sum} \} (x:=x-1; y:=y+1) \end{array}$$

Like other theorems, spec-statements may involve metavariables. As well as being used in rules, they may be used to represent the results of expressions evaluated in particular states. For example,

$$n_0 \cdot \{ s = n_0! - n! \} (s := s \times n; n := n-1)$$

Here n_0 may represent any object at all — as is always the case with metavariables; the interesting values are those for which the invariant is true when applied to a particular case.

The code invariant may be defined in terms of the more general code spec:

$$\text{code-inv-defn: } \text{inv}, \text{Code} \cdot \frac{\{ \text{inv} :- \text{inv} \} \text{Code}}{\{ \text{inv} \} \text{Code}}$$

5-1.2.2 Code specs

$$\{ \text{pre} :- \text{post} \} \text{code}$$

This asserts that if *pre* is true just before *code* is executed, then *code* will terminate and *post* will be true of the relation between the states just before and just after. If *pre* is omitted, it is assumed to be **true**. (Don't confuse $\{ :- e \}$ with $\{ e \}$.)

The *fields* define which program variables (or components thereof) may be altered by the *code*. Within *post*, any of the fields may be quoted barred, denoting the value it represented before execution. E.g.

$$\begin{array}{l} \{ n \in \text{Integer} \wedge n > 0 :- s = \bar{n}! \} \\ (s := 0; [n \neq 0] \text{ while True: } [n_0 \cdot \{ s = n_0! - n! \} (s := s \times n; n := n-1)]) \end{array}$$

5-1.2.3 Opspecs

The opspec is a specialisation of the pre/post-assertion in which the *Code* is restricted to a single *Message* whose parameters are restricted to constants or metavariables. Opspecs are used for stating theorems about particular operations.

For example:

$$\{ z > 0 :- \uparrow \times \uparrow = z \} (z.\text{sqrt})$$

The special name \uparrow refers to the value returned from the operation.

5-2 Decomposition proofs

Decomposition is the development of code which will meet a given operation specification. A set of basic rules is given here for the fundamental coding constructs of sequential, alternative and loop execution, and for the construction of expressions (with possible side-effects) and assignments. Ideally, opsspecs serve directly as proof-rules for the employment of the operations they specify, though this works well only under certain restrictions (as we shall see).

5-2.1 Basic rules

These are reformulated from [Morgan] and [Jones86a] to suit the Fresco axiomatic style. Questions of framing and possible aliasing are omitted at this stage, and dealt with separately below.

5-2.1.1 Strengthening

This combines the usual weaken-precondition and strengthen-postcondition rules:

$$\text{stren:} \quad P, P1, R1, R, S \cdot \frac{\begin{array}{c} P \vdash P1 \\ \bar{P}, R1 \vdash R \\ \{ P1 :- R1 \} S \end{array}}{\{ P :- R \} S}$$

For the second hypothesis, you don't have to prove that R always follows from $R1$ —just that it will do in those cases where P is satisfied by the *prior* state.

Since code specs can be nested, *stren* may be paraphrased:

$$\text{stren':} \quad P, P1, R1, R, S \cdot \frac{\begin{array}{c} P \vdash P1 \\ \bar{P}, R1 \vdash R \end{array}}{\{ P :- R \} \{ P1 :- R1 \} S}$$

This statement is satisfiable by any S which terminates if P holds:

$$\text{terminates:} \quad P, S \cdot \{ P :- \text{true} \} S$$

This specification says nothing:

$$\text{bottom:} \quad R, S \cdot \{ \text{false} :- R \} S$$

No S satisfies this specification:

$$\text{miracle:} \quad P, S \cdot \{ P :- \text{false} \} S$$

5-2.1.2 Spec-statement conjunction

A statement may be called upon to satisfy more than one theorem (from different parent types, for example). Each theorem applies to those situations in which its precondition is true. It's perhaps worth considering carefully the consequences of this if two theorems apply to the same statement. They may be conjoined into a theorem which applies if either of the preconditions are true — that is, the precondition of

the new theorem is the disjunction of the originals; and the result will depend on which or both of them applied.

$$\text{spec-conj: } P1, P2, R1, R2, S \cdot \frac{\frac{\{ P1 :- R1 \} S \quad \{ P2 :- R2 \} S}{\{ P1 \vee P2 :- (\overline{P1} \Rightarrow R1) \wedge (\overline{P2} \Rightarrow R2) \} S}}{\{ P1 \vee P2 :- (\overline{P1} \Rightarrow R1) \wedge (\overline{P2} \Rightarrow R2) \} S}$$

Conversely, suppose we believe the lower half of **spec-conj** to apply to some **S**: then **stren** may be used to show $\{ P1 :- R1 \} S$ (which is OK because in writing this theorem, we don't care what happens if **P2** is true or false, nor what happens if **P1** is false).

Consequences of this include:

$$\text{post-conj: } P, R1, R2, S \cdot \frac{\frac{\{ P :- R1 \} S \quad \{ P :- R2 \} S}{\{ P :- R1 \wedge R2 \} S}}{\{ P :- R1 \wedge R2 \} S}$$

and:

$$\text{pre-disj: } P1, P2, R, S \cdot \frac{\frac{\{ P1 :- R \} S \quad \{ P2 :- R \} S}{\{ P1 \vee P2 :- R \} S}}{\{ P1 \vee P2 :- R \} S}$$

5-2.1.3 Sequence

For a sequence of statements $S_1; \dots; S_n$ whose postconditions M_i govern variables x_j , and which contain no other barred variables:

$$\text{seq: } P, M_i, S_i, x_j \cdot \frac{\frac{\{ P :- M_1[\bar{x}_j] \} S_1 \quad x_{0j} \cdot \{ M_{i-1}[x_{0j}] :- M_i[x_{0j}] \} S_{2 \leq i \leq n}}{\{ P :- M_n[\bar{x}_j] \} (S_1; S_2; \dots; S_n)}}{\{ P :- M_n[\bar{x}_j] \} (S_1; S_2; \dots; S_n)}$$

The trick is to give intermediate names x_{0j} to the original values of all the variables x_j ; thereafter, the barred variables have to be removed from the successive M_i .

For example, suppose we wish to verify a sequence of three statements which manipulate variables x , y , and z . Begin with their specifications:

- a: $\{ :- x = \bar{x} f1 \wedge y = \bar{y} \wedge z = \bar{z} \} S1$
- b: $\{ :- x = \bar{x} f2 \wedge y = \bar{x} g2 \wedge z = \bar{z} \} S2$
- c: $\{ :- x = (\bar{x} f3; \bar{y}) \wedge y = (\bar{y} g3; (\bar{x}, \bar{z})) \wedge z = \bar{x} h \} S3$

It is also important that these functions can be shown to depend only on their arguments (§8-3.3 — p.141)

df1: $f1$ transparent

df2: $f2$ transparent

df3: ...and so on for $f3, g2, g3, h$...

Now augment **b** so that it fits with the postcondition of **a** and eliminate bars:

bp1: $x = x_0 \text{ f1 } \wedge y = y_0 \wedge z = z_0 \vdash \text{true}$ /* precondition of b */ by true-intro
 bp2: h1: $\frac{x = \bar{x}_0 \text{ f1 } \wedge y = \bar{y}_0 \wedge z = \bar{z}_0}{x = \bar{x} \text{ f2 } \wedge y = \bar{y} \text{ g2 } \wedge z = \bar{z}}$ /* barred version of new precondition */
 h2: $\frac{x = \bar{x}_0 \text{ f1 } \wedge y = \bar{y}_0 \wedge z = \bar{z}_0}{x = \bar{x} \text{ f2 } \wedge y = \bar{y} \text{ g2 } \wedge z = \bar{z}}$
 1: $\frac{x = \bar{x}_0 \text{ f1 } \wedge y = \bar{y}_0 \wedge z = \bar{z}_0}{x = x_0 \text{ f1 f2 } \wedge y = x_0 \text{ f1 g2 } \wedge z = z_0}$ from bp2.h1, df1 by unbar
 b2: $\{ x = x_0 \text{ f1 } \wedge y = y_0 \wedge z = z_0 \vdash x = x_0 \text{ f1 f2 } \wedge y = x_0 \text{ f1 g2 } \wedge z = z_0 \}$ S2
 from bp1, bp2, b by stren

and similarly for c:

cp1: $x = x_0 \text{ f1 f2 } \wedge y = x_0 \text{ f1 g2 } \wedge z = z_0 \vdash \text{true}$ by true-intro
 cp2: h1: $\frac{x = \bar{x}_0 \text{ f1 f2 } \wedge y = \bar{x}_0 \text{ f1 g2 } \wedge z = \bar{z}_0}{x = (\bar{x} \text{ f3: } \bar{y}) \wedge y = (\bar{y} \text{ g3: } (\bar{x}, \bar{z})) \wedge z = \bar{x} \text{ h}}$
 h2: $\frac{x = (\bar{x} \text{ f3: } \bar{y}) \wedge y = (\bar{y} \text{ g3: } (\bar{x}, \bar{z})) \wedge z = \bar{x} \text{ h}}{x = (x_0 \text{ f1 f2 f3: } x_0 \text{ f1 g2}) \wedge y = (x_0 \text{ f1 g2 g3: } (x_0 \text{ f1 f2, } z_0)) \wedge z = x_0 \text{ f1 f2 h}}$
 c2: $\{ x = x_0 \text{ f1 f2 } \wedge y = x_0 \text{ f1 g2 } \wedge z = z_0 \vdash x = (x_0 \text{ f1 f2 f3: } x_0 \text{ f1 g2}) \wedge y = (x_0 \text{ f1 g2 g3: } (x_0 \text{ f1 f2, } z_0)) \wedge z = x_0 \text{ f1 f2 h} \}$ S3
 from cp1, cp2, c by stren

and finally, we can apply seq:

$\vdash \{ \vdash x = (\bar{x} \text{ f1 f2 f3: } \bar{x} \text{ f1 g2}) \wedge y = (\bar{x} \text{ f1 g2 g3: } (\bar{x} \text{ f1 f2, } \bar{z})) \wedge z = \bar{x} \text{ f1 f2 h} \}$
 $\vdash (S1; S2; S3)$ from a, b2, c2 by seq

Typing clauses are omitted here for clarity. The explicit preservation clauses $y = \bar{y} \wedge z = \bar{z}$ are required because we have not yet considered framing.

5-2.1.4 Condition

Smalltalk's blocks carry no problems provided we deal only with specific uses of them. The basic conditional rule is:

$$\text{if: } P, R, C, S1, S2 \cdot \frac{\{ P \wedge C \vdash R \} S1 \quad \{ P \wedge \neg C \vdash R \} S2}{\{ P \vdash R \} C \text{ ifTrue: } [S1] \text{ ifFalse: } [S2]}$$

but this assumes that C has no side-effects. If C does cause changes, then the effect is like $(\{v\} \vee v := C. v \text{ ifTrue: } [S1] \text{ ifFalse: } [S2])$, so the rule becomes

$$\text{if-se: } P, R, C, S1, S2 \cdot \frac{\{ P \vdash M[\bar{x}_i] \} C \quad \{ M[xx_i] \wedge C \vdash R[xx_i] \} S1 \quad \{ M[xx_i] \wedge \neg C \vdash R[xx_i] \} S2}{\{ P \vdash R[\bar{x}_i] \} C \text{ ifTrue: } [S1] \text{ ifFalse: } [S2]}$$

5-2.1.5 Loop

$$\text{loop: } v, \text{inv}, C, S \cdot \frac{v \cdot \{ v \in \text{Int} \wedge \text{inv} \wedge C \vdash \text{inv} \wedge 0 \leq v \wedge v < \bar{v} \} S}{\{ \text{inv} \vdash \text{inv} \wedge \neg C \} [C] \text{ whileTrue: } [S]}$$

v is a *variant*, any expression chosen so that it reduces monotonically, but not beyond 0: and therefore guarantees termination. It does not, of course, need to be realised in an actual code variable.

Some styles of programming use **C** as a substantial piece of code with side-effects; in which case, the rule is

loop-se: $v, \text{inv}, \text{PT}, \text{C}, \text{S} \cdot$

$$\frac{\begin{array}{l} r \cdot \{ \text{inv}[x_0] \wedge \neg \text{PT} :- R[\bar{x} \backslash x_0] \wedge r = \text{false} \} r := \text{C} \\ r \cdot \{ \text{inv}[x_0] \wedge \text{PT} :- M[\bar{v}, x_0] \wedge r = \text{true} \} r := \text{C} \\ v_0 \cdot \{ v \in \text{Int} \wedge M[v_0, x_0] :- \text{inv}[x_0] \wedge 0 \leq v \wedge v < v_0 \} \text{S} \end{array}}{\{ \text{inv}[x] :- R[\bar{x}] \} ([\text{C}] \text{whileTrue: } [\text{S}])}$$

All barred variables \bar{x} in the concluding R must be substituted by other names x_0 in the decomposition. PT is whatever condition gives rise to C ultimately evaluating to true. (If it's too difficult to characterise this when applying this rule — for example, if C is not sufficiently deterministic — try making $R \equiv M$ and forget about PT .)

For example, to the fairly useless piece of code

$[a := a \times 2. a < b] \text{whileTrue: } [b := b/2]$

the rule can be applied with these substitutions, creating a_0, b_0 as names for the initial values:

$$\begin{array}{l} \text{inv}[a_0, b_0] \equiv a \times b = a_0 \times b_0, \\ \text{PT} \equiv a \times 2 > b, \\ v \equiv (b/a) \text{floor}, \\ M[a_0, b_0] \equiv v = \bar{v}/2 \wedge a \times b/2 = a_0 \times b_0, \\ R[a_0, b_0] \equiv a \times b/2 = a_0 \times b_0 \wedge a > b \end{array}$$

hence

$$\{ :- a \times b/2 = \bar{a} \times \bar{b} \wedge a \geq b \} [a := a \times 2. a < b] \text{whileTrue: } [b := b/2]$$

5-2.1.6 Inline form

The rules can also be written and applied in a style more suited to the integration of program code and development, in which specification-statements can be nested. (See §10-1 – *Fresco development language FST* (p.165))

$$\begin{array}{ll} \text{stren: } P, R, P1, R1, S \cdot & \{ P :- R \} ((P \vdash P1), (\bar{P}, R1 \vdash R), \{ P1 :- R1 \} S) \\ \text{seq: } P, M_i, x_i, S_j \cdot & \{ P :- M_n[\bar{x}_j] \} x_{0j} \cdot (\{ P :- M_i[\bar{x}_j] \} S_1; \{ M_{i-1}[x_{0j}] :- M_i[x_{0j}] \} S_i; \dots) \\ \text{if: } P, R, C, S1, S2 \cdot & \{ P :- R \} (C \text{ ifTrue: } [\{ P \wedge C :- R \} S1] \text{ ifFalse: } [\{ P \wedge \neg C :- R \} S2]) \\ \text{loop-se: inv, C, v, S} \cdot & \{ \text{inv}[x] :- R[\bar{x}] \} \\ & ([\quad (\{ \text{inv}[x_0] \wedge \neg \text{PT} :- R[x_0] \wedge \uparrow = \text{false} \}, \\ & \quad \{ \text{inv}[x_0] \wedge \text{PT} :- M[\bar{v}, x_0] \wedge \uparrow = \text{true} \}) C] \\ & \text{whileTrue: } [\{ v \in \text{Int} \wedge M[v_0, x_0] :- \text{inv}[x_0] \wedge 0 \leq v \wedge v < v_0 \} S]) \end{array}$$

The example given under §5-2.1.3 – *Sequence* (p.74) now looks like this:

$x, y, z, S1, S2, S3 \cdot$

$\{ \vdash x = (\bar{x} f1 f2 f3: \bar{x} f1 g2) \wedge y = (\bar{x} f1 g2 g3: (\bar{x} f1 f2, \bar{z})) \wedge z = \bar{x} f1 f2 h \}$

$x_0, y_0, z_0 \cdot ($

$\{ x = \bar{x} f1 \wedge y = \bar{y} \wedge z = \bar{z} \} S1 ;$

$\{ x = x_0 f1 \wedge y = y_0 \wedge z = z_0 \vdash x = x_0 f1 f2 \wedge y = x_0 f1 g2 \wedge z = z_0 \}$

$((x = x_0 f1 \wedge y = y_0 \wedge z = z_0 \vdash \text{true}),$

$(h1: \bar{x} = x_0 f1 \wedge \bar{y} = y_0 \wedge \bar{z} = z_0 ,$

$h2: x = \bar{x} f2 \wedge y = \bar{x} g2 \wedge z = \bar{z} ,$

$\vdash x = x_0 f1 f2 \wedge y = x_0 f1 g2 \wedge z = z_0 \quad \text{from h1, h2 by subs= },$

$\{ \text{true} \vdash x = \bar{x} f2 \wedge y = \bar{x} g2 \wedge z = \bar{z} \} S2) ;$

$\{ x = x_0 f1 f2 \wedge y = x_0 f1 g2 \wedge z = z_0$

$\vdash x = (x_0 f1 f2 f3: x_0 f1 g2) \wedge y = (x_0 f1 g2 g3: (x_0 f1 f2, z_0)) \wedge z = x_0 f1 f2 h \}$,

$(h1: \bar{x} = x_0 f1 f2 \wedge \bar{y} = x_0 f1 g2 \wedge \bar{z} = z_0 ,$

$h2: x = (\bar{x} f3: \bar{y}) \wedge y = (\bar{y} g3: (\bar{x}, \bar{z})) \wedge z = \bar{x} h ,$

$\vdash x = (x_0 f1 f2 f3: x_0 f1 g2)$

$\wedge y = (x_0 f1 g2 g3: (x_0 f1 f2, z_0)) \wedge z = x_0 f1 f2 h ,$

$\{ \text{true} \vdash x = (\bar{x} f3: \bar{y}) \wedge y = (\bar{y} g3: (\bar{x}, \bar{z})) \wedge z = \bar{x} h \} S3)$

)

The rules given here are derived from [Morgan], but

- **stren** combines several of Morgan's rules in a convenient manner
- **seq** conveniently handles more than one statement
- **loop-se** permits side-effects in the condition
- the scopes of effects of statements are dealt with explicitly here, since a more complex framing scheme than Morgan's will be introduced presently, to cope with possible aliasing.

5-2.2 Assignment

$$\text{assignment: } P, R, x, E \cdot \frac{\{ P: \neg R[\uparrow, x] \} E}{\{ P: \neg \exists x_0 \cdot R[x, x_0] \wedge \uparrow == x \} x := E}$$

or in inline form:

assignment: $\{ P: \neg \exists x_0 \cdot R[x, x_0] \wedge \uparrow == x \} x := \{ P: \neg R[\uparrow, x] \} E$

and if x does not occur in E , this can be simplified to:

assignment': $\{ P: \neg R \wedge \uparrow == x \} x := \{ P: \neg R \} E[x \setminus]$

The assignment rule takes account of the possibility that E is not pure. Compare with Morgan's, in which $R[E]$ would be difficult to interpret if E has side-effects:

Morgan Law 6.6: $P, R, y, E \cdot (y = \bar{y} \wedge P \vdash R[E]) \vdash \{ P: \neg R[y] \} E$

For example:

$$\begin{array}{lcl}
x, y \cdot & h1: & \{ x \in \text{Set} :- x = \bar{x} \cup \{y\} \wedge \uparrow == y \} \{ x \text{ add: } y \\
& 1: & \{ x \in \text{Set} :- \exists X \cdot X = \bar{x} \cup \{y\} \wedge x == y \wedge \uparrow == x \} \{ x := x \text{ add: } y \\
& & \text{by assignment from h1} \\
\vdash & & \{ x \in \text{Set} :- x == y \wedge \uparrow == x \} \{ x := x \text{ add: } y \} \text{ by } \wedge\text{-elim, } \exists\text{-redundant}
\end{array}$$

5-2.3 Operation invocation and results

We have already seen how the theorems specifying each operation act as proof rules for their clients. If invoked from outside the type in which it is defined, an operation's specifying theorems need to be context-extracted from the type (§6-3.1 — p.88). For example,

Point
fn diff \in (Point) Point diff: $\{ p \in \text{Point} :- \uparrow = \text{Point}(x - p \ x, y - p \ y) \} \text{ diff}(p)$

yields the theorem (after some renaming):

$$\begin{array}{l}
\text{Point}::\text{diff: } pa, pb \cdot \\
\quad \{ pa \in \text{Point}, pb \in \text{Point} \\
\quad :- \uparrow = \text{Point}(pa \ x - pb \ x, pa \ y - pb \ y) \\
\quad \} pa \text{ diff}(pb)
\end{array}$$

and in well-behaved cases, this can be applied easily:

$$\begin{array}{lcl}
\dots :- \uparrow = \text{Point}(p1 \ x + p2 \ x, p1 \ y + p2 \ y) \{ p1 \text{ add}(p2) & & \text{by Point::add} \\
\dots :- \uparrow = \text{Point}(\text{Point}(p1 \ x + p2 \ x, p1 \ y + p2 \ y) \ x - p2 \ x, & & \\
\quad \text{Point}(p1 \ x + p2 \ x, p1 \ y + p2 \ y) \ y - p2 \ y) \{ \{ p1 \text{ add}(p2) \text{ diff}(p2) \} & & \\
\quad \text{by Point::diff } [pa \setminus p1 \text{ add}(p2), pb \setminus p2] & & \\
\dots :- \uparrow = \text{Point}((p1 \ x + p2 \ x) - p2 \ x, (p1 \ y + p2 \ y) - p2 \ y) \{ \{ p1 \text{ add}(p2) \text{ diff}(p2) \} & & \\
\quad \text{by Point(Real,Real)-defn} & & \\
\dots :- \uparrow = \text{Point}(p1 \ x, p1 \ y) \{ \{ p1 \text{ add}(p2) \text{ diff}(p2) \} & & \text{by arith} \\
p1, p2, pr \cdot \{ p1 \in \text{Point} \wedge p2 \in \text{Point} :- \uparrow = \text{Point}(p1) \} \{ p1 \text{ add}(p2) \text{ diff}(p2) \} & & \\
& & \text{by Point(Point)-defn}
\end{array}$$

(Notice that we have to conclude not that $p1$ results, but that a copy of it results.)

However, in some cases, the arguments themselves have side-effects, and these must be taken into account. One way is to state an equivalence between an operation-invocation and a sequence of statements. If p_i stand for variables and E_i for expressions with possible side-effects:

$$\begin{array}{l}
\{ RE[X] :- R[X] \} r := p_0 \text{ op } (p_i) \\
\{ P :- RE[\bar{x}] \} (\text{var } v_0, v_1, \dots \cdot v_0 := E_0; v_1 := E_1 \dots; r := v_0 \text{ op } (v_i)) \\
\hline
\{ P :- R[\bar{x}] \} r := E_0 \text{ op } (E_i)
\end{array}$$

(This is a simplification, as it would be more generally useful not to assume this order of evaluation of arguments.)

From the assignment' and sequence rules, we can derive this rule, in which M_i must be free of barred variables:

$$\text{opcall: } P, M_i, p_i, x_{0j}, x_j \cdot \frac{\begin{array}{l} \{ P :- M_0[\uparrow, \bar{x}_j] \} e_0 \\ \wedge_i \{ M_{i-1}[p_{i-1}, x_{0j}] :- M_i[\uparrow, x_{0j}] \} e_i \\ \{ M_n[p_n, x_{0j}] :- M_{op}[\uparrow, x_{0j}] \} p_0 \text{ op } (p_i) \end{array}}{\{ P :- M_{op}[\uparrow, \bar{x}] \} e_0 \text{ op } (e_i)}$$

in which p_i stand for the values of the arguments as they are computed, and x_{0j} are the original values of all other variables.

For example:

- 1 $\{ x \in \text{Set} :- x \in \text{Set} \wedge \uparrow == x \wedge x = \bar{x} \wedge y = \bar{y} \} x$ by var-exprn
- 2.1 $\{ x \in \text{Set} :- x = \bar{x} \cup y \wedge \uparrow == y \wedge y = \bar{y} \} x \text{ add: } y$ by Set-add-defn...
- 2 $\{ x \in \text{Set} \wedge p_0 == x \wedge x = x_0 \wedge y = y_0$
 $:- x \in \text{Set} \wedge p_0 == x_0 \wedge x = x_0 \cup y \wedge \uparrow == y \wedge y = y_0 \} x \text{ add: } y$ by stren
- 3.1 $\{ p_0 \in \text{Set} :- p_0 = \overline{p_0} - p_1 \wedge \uparrow == p_1 \wedge p_1 = \overline{p_1} \} p_0 \text{ sub: } p_1$ by Set-sub-defn
- 3: $\{ p_0 \in \text{Set} \wedge p_0 = x_0 \cup \{p_1\} \wedge p_1 == y \wedge y = y_0 \wedge p_0 == x_0$
 $:- p_0 == x_0 \wedge \uparrow == y \wedge p_0 = x_0 \wedge y = y_0 \} p_0 \text{ sub: } p_1$ by stren...
- 4: $\{ x \in \text{Set} :- p_0 == x \wedge \uparrow == y \wedge p_0 = \bar{x} \wedge y = \bar{y} \} x \text{ sub: } (x \text{ add: } y)$ by opcall
- c: $\{ x \in \text{Set} :- \uparrow == y \wedge x = \bar{x} \wedge y = \bar{y} \} x \text{ sub: } (x \text{ add: } y)$ from 4 by subs==

opcall is required only where there are possible side-effects: clearly, it's easier to use pure expressions where possible!

5-2.3.1 Yield of an expression

yield: $P, Q, x, \text{op} \cdot \{ P :- Q[\uparrow] \} x \text{ op } \vdash Q[x \text{ op}]$

5-3 Issues in the use of programming language in assertions

The language of specification in Fresco is just an extension of the implementation language, for simplicity and close integration. Unlike some wide-spectrum languages, the programming component is practical and imperative, which raises some questions about possible ambiguities when used for specification.

5-3.1 Underdetermined expressions

Since a postcondition is a relation which can be as loose as you like, functions defined by opspecs may not have precise values. For example, `sqrt` (specified above) has two possible meanings for most receivers. Some caution is therefore required in employing such functions within the specifications of other operations. We must avoid making assumptions such as $4 \text{ sqrt} = 4 \text{ sqrt}$ — otherwise, it could be shown that $2 = -2$, and hence that Bertrand Russell and I are the Pope etc.

The rule for making inferences about opspec-defined functions is therefore slightly circumspect:

$$\text{use-pure:} \quad G, F, R, a, S \cdot \frac{F[a] \Delta \emptyset \quad a \cdot \{ P[a] :- R[\uparrow, a] \} F[a] \quad \{ \Delta s \cdot PS :- P[a] \wedge G[F[a]] \} S}{\{ \Delta s \cdot PS :- \exists n \cdot G[n] \wedge R[n, \bar{a}] \} S}$$

— must be applied separately for each occurrence of $F[a]$ in G .

The rather ugly side-condition is necessary because a separate new variable is required for each occurrence, to allow for them having different values. (Side-conditions can be coped with in the basic rules, since they can be ‘wired-in’ to the proof tool.) The first hypothesis states that $F[a]$ must have no side-effects — see §8-3 — p.133.

So, for example:

```

h1:  a · { a ∈ Real ∧ a > 0 :- ↑ = a sqrt × a } a powerThreeHalves           // defn
h2:  b · { b ∈ Real ∧ b > 0 :- ↑ × ↑ = b } b sqrt                          // defn of sqrt
0:   a · { a ∈ Real ∧ a > 0 :- a ∈ Real ∧ a > 0 ∧ ↑ = a sqrt × a } a powerThreeHalves
                                     from h1 by carry-pre
1:   a · { a ∈ Real ∧ a > 0 :- ∃ n · ↑ = n × a ∧ n × n = a } a powerThreeHalves
      from h2, 0 by use-pure
      with  G[e] → ↑ = e × a, F[e] → e sqrt, R[e1, e2] → e1 × e2 = e2,
            a → a, S → a powerThreeHalves
2:   { 4 ∈ Real ∧ 4 > 0 :- ∃ n · ↑ = n × 4 ∧ n × n = 4 } 4 powerThreeHalves by 2
3:   { :- ↑ = 2 × 4 ∧ 2 × 2 = 4 ∨ ↑ = -2 × 4 ∧ -2 × -2 = 4 ∨ ↑ = 43 × 4 ∧ 43 × 43 = 4 ∨ ... }
      4 powerThreeHalves                                     by expansion of ∃ from 2
4:   { :- ↑ = 8 ∨ ↑ = -8 } 4 powerThreeHalves by arith & simplication from 3

```

The underdetermination is carried through.

5-3.2 Promotion

The rule above applies only to pure functions. No rule is available for direct use of expressions with side-effects.

However, any opspec can be *promoted* — quoted, via the name of its operation, in other theorems:

$$\begin{aligned} & \{ p \in \text{Point} \wedge v \in \text{Point} :- p \cdot x = \bar{p} \cdot x + v \cdot x \wedge p \cdot y = \bar{p} \cdot y + v \cdot y \} p \text{ move: } v \\ & \{ d \in \text{Drawing} \wedge fx \in d \text{ points} \wedge fx \neq f :- fx = \bar{fx} \} d \text{ update: } f \\ & \{ art \in \text{Drawing} \wedge f \in art \text{ points} :- \llbracket f \text{ move: } v \rrbracket \wedge \llbracket art \text{ update: } f \rrbracket \} art \text{ move: } f \text{ by: } v \end{aligned}$$

This permits theorems to be neatly factored: in this example, *update* is an operation quoted by many of the opspecs for this type, and its purpose is to ‘carry’ those predicates common to them all. Modularisation is also encouraged by this mechanism: this operation has the effect on f stipulated by its *move* operation, but we have left it to the specification of f ’s type to say what that is; if the meaning of *move*ing individual components of the *drawing* changes, then so automatically does the meaning of *move*ing the current type.

The semantics is given by:

$$\text{promote: } P, R, S \cdot \frac{v_i \cdot \{ P[v_i] :- R[v_i, v_i] \} S[v_i]}{x_i \cdot \llbracket S[x_i] \rrbracket \Rightarrow (P[x_i] \Rightarrow R[\bar{x}_i, x_i])}$$

This means that if we can find any opspec which relates to *op*, then we can interpret it as part of the specification of *S*. There may be several such opspecs, and we may not know all of them, but any inferences made from any one will not be invalidated by later discoveries.

There is no inference to be made from $\neg \llbracket \dots \rrbracket$: so there's no point in designers trying to write negative promotions. This is essential, because our compositionality aim is never to rely upon having seen all the rules which might apply to a particular operation until coding time, and so should not be able to say what it might mean definitely *not* to comply with its spec.

Asserting [in]equality between promotions is equally useless, with the same justification.

However, the result of conjoining two promotions parallels that of stating two opspecs for one operation:

$$\frac{\begin{array}{c} \{ P_A :- R_A \} S_A \\ \{ P_B :- R_B \} S_B \end{array}}{\llbracket S_A \rrbracket \wedge \llbracket S_B \rrbracket \Rightarrow ((\bar{P}_A \Rightarrow R_A) \wedge (\bar{P}_B \Rightarrow R_B))}$$

(Parameterisation of the statements has been omitted here, for clarity.)

Disjunction of two promotions is useful for expressing exception-handling. Here, we guarantee that (provided both their preconditions are met) the specifications of at least one of them will apply:

$$\frac{\begin{array}{c} \{ P_A :- R_A \} S_A \\ \{ P_B :- R_B \} S_B \end{array}}{\llbracket S_A \rrbracket \vee \llbracket S_B \rrbracket \Rightarrow (\bar{P}_A \wedge \bar{P}_B \Rightarrow R_A \vee R_B)}$$

so we could envisage composing a specification from several partial operation-specs:

$$\llbracket \text{compileAndLink} \rrbracket \vee \llbracket \text{reportErrors} \rrbracket \wedge \llbracket \text{deleteObjFiles} \rrbracket$$

(The preconditions for each of these would not include possible end-user errors like syntax mistakes in the compiler input: recall that the point of a precondition is to document what ought *always* to be true on entry to a procedure if the design is to be considered correct; preconditions are not for documenting cases which may arise at run time. Syntax of the compiled language is therefore part of *compileAndLink*'s postconditions.)

Given that other logical operators between promotions are not useful, it would be interesting to investigate other compositions that are possible inside the promotions; for example:

conjoin: $\{ PA \vee PB :- (PA \Rightarrow RA) \wedge (PB \Rightarrow RB) \} (SA \mid \wedge \mid SB)$
disjoin: $\{ PA \wedge PB :- RA \vee RB \} (SA \mid \vee \mid SB)$
intersect: $\{ PA \wedge PB :- RA \wedge RB \} (SA \mid * \mid SB)$
fallback: $\{ PB :- (RA \neq RB) \wedge (\neg PA \Rightarrow RB) \} (SA \mid / \mid SB)$
seq: $\{ PA :- \exists s' \cdot RA(s, s') \wedge RB(s', s) \} (SA ; SB)$

The last is, of course, familiar, and is included to demonstrate uniformity between this executable composition of statements and the others, which though inexecutable, are meaningful within $\llbracket \dots \rrbracket$: for example,

$$\llbracket [SA] \wedge [SB] \rrbracket \mid - \llbracket [SA] \wedge [SB] \rrbracket$$

5-3.2.1 Satisfying Promotions

Promotions are Useful for factoring specifications, but in general there is less advantage when it comes to implementation. All the opspects relevant to a promoted message must be gathered and conjoined; and the implementation of the flattened result may or may not use the implementations of the promoted messages. Whenever an opspect is added to a promoted message, its uses must be traced and reimplemented.

The implementation of a composition may use the components' implementations under certain circumstances:

choose: $\{ :- \llbracket [SA] \rrbracket \vee \llbracket [SB] \rrbracket \} (C \text{ ifTrue: } [SA] \text{ ifFalse: } [SB])$
except: $\{ :- \llbracket [SA] \rrbracket \vee \llbracket [SB] \rrbracket \} (\text{Exception new within: } [SA] \text{ handle: } [SB])$
seq- \wedge :

$$\frac{\begin{array}{c} \{ P_{Ai} :- R_{Ai} \} S_A \\ \{ P_{Bi} :- R_{Bi} \} S_B \\ \{ \bigwedge R_{Ai} \} S_B \\ \bigwedge \bar{P}_{Ai} \wedge \bar{P}_{Bi} \wedge R_{Ai} \vdash \bigwedge \bar{P}_{Bi} \end{array}}{\{ \llbracket [SA] \rrbracket \wedge \llbracket [SB] \rrbracket \} (S_A ; S_B)}$$

5-4 Summary

This chapter has dealt with the use of expressions in executable code and in theorems. Rules have been given for procedural decomposition proofs.

Whilst the decomposition proofs owe a great deal to the [Morgan], the rules are somewhat more complicated, to allow for the possibilities of side-effects.

An important property of the theorems which specify statements is monotonic conjunction.

6 Types and Classes

6-1 Types

6-1.1 Type theories

- The *theory* of a Fresco type T is a set of theorems $A_{T,i}$ over a set of message-selectors.
- An object x is a member of type T , written $x \in T$, iff all the theorems of the theory of T are valid when x is substituted for **self** (and after making **self** explicit as a prefix to attribute names — **self.x** rather than just x):

$$\frac{\begin{array}{c} A_{T,1}[\text{self}x] \\ \dots \\ A_{T,n}[\text{self}x] \end{array}}{x \in T}$$

Type definitions will be interpreted in such a way that all the axioms are predicates over object-behaviour, rather than individual states. Theorems therefore apply to the whole behaviour of an object; so an object's membership of a type does not change with time.

An object may be a member of many types.

The fundamental way of proving type membership is to prove the type's theorems. Conversely, if an object is known to belong to a particular type, then we may infer that that type's theorems are true of it.

The theorems of a type may be partitioned into *axioms*, those given by the designer as defining the type; and *derived theorems*, provable from the axioms. To prove that an object is a member of a given type, it is therefore only necessary to prove that the type's axioms are valid for that object.

If one of the theorems of a type is unsatisfiable, then the type is necessarily empty. However, this has no effect on any other types there may be in the system. For example, if one of the theorems of type $T1$ contains the conjunct

$$\forall y \in T2 \cdot P(y)$$

this places no extra obligation upon the members of $T2$ additional to $T2$'s own definition; rather, this just means that $T1$ is empty unless the term does happen to be always true for all possible members of $T2$.

The set of axioms implies the set of message-selectors which the type understands, and the set of model-variables in terms of which the type is described: they are just those which can be found within the axioms. An operation in the signature would be no use without at least one axiom to define its effects.

6-1.2 Membership of types

Fresco types are sets of object histories.

6-1.2.1 Object Histories

An object passes through a sequence of states in its lifetime.

ObjectHistory
var transitions \in List(Transition) $\forall i \cdot i \in 1..transitions.length-1 \Rightarrow$ transitions[i].after = transitions[i+1].before

Transition
var before \in State var msg \in Message

Each transition is brought about by a message.

Message
var selector \in Name var args \in List (ObjectTransition)

Each argument may be altered by the message, so an **ObjectTransition** is a segment of an object history, representing the states of the argument before and after.

ObjectTransition
var before \in State var after \in State var between \in ObjectHistory between.first = before \wedge between.last = after

6-1.2.2 Type membership

Every type definition T can be rendered as a set of opspecs

$$A_{T,n} \equiv \{ P_n(\sigma, p) :- R_n(\bar{\sigma}, \sigma, \bar{p}, p, \uparrow) \} op_n(p)$$

where \bar{p}, p are vectors of **States**, and $\bar{\sigma}, \sigma, \uparrow$ are **States**; and an invariant

$$A_{T,i} \equiv \Box inv(\sigma)$$

(where \Box is a convenient borrowing from temporal logic, indicating that the invariant is true of every state in the history). There may be more than one opspecc applicable to each message (so $op_{n1} = op_{n2}$ is not excluded); and there may be several invariants.

An **ObjectHistory** h is a *weak member* of T , written $x : T$, if and only if

$\forall i \in h.transitions.indices \cdot$

let $\bar{\sigma} = h.transitions[i].before, \sigma = h.transitions[i].after,$
 $m = h.transitions[i].msg, \bar{a} = m.args.before, a = m.args.after$ **in**

$$\bigwedge n \cdot P_n(\bar{\sigma}, \bar{a}) \wedge m.selector = op_n \wedge inv(\bar{\sigma}) \Rightarrow R_n(\bar{\sigma}, \sigma, \bar{a}, a, m.return) \wedge inv(\sigma)$$

An **ObjectHistory** h is a *strong member* of T , written $h \in T$, if and only if

$$h : T \wedge \forall i \in h.\text{transitions.indices} \cdot \text{let } \sigma = h.\text{transitions}[i].\text{before} \text{ in } \bigwedge i \cdot \text{inv}_i(\sigma)$$

Weak type membership says that if an object satisfies the invariant, then it will behave as expected and the invariant will remain true, but may behave as it likes otherwise: every opspec's pre and postconditions are conjoined with the invariant. Strong type membership asserts that the invariant is indeed true in all states; and is the usual requirement — assume this variety of typing by default. The use of the distinction is discussed in §8-1 — p.125. A difference arises only during execution of a message or in some cases where aliasing disturbs an invariant.

Notice that although this forces predictable behaviour for any message about which there is an axiom whose preconditions are fulfilled, the invariant is the only constraint on the results of any message for which there is no axiom, or which does not meet any axiom's preconditions. This is a useful property when types are to be composed.

An object is said to belong to a type ($x \in T$ or $x : T$) if it can be shown that all its possible histories belong to the type ($h \in T$ or $h : T$).

- *Type monotonicity theorem.* Each axiom which is added to a type-definition restricts the type to a subset of what it was before.
It is clear that no new transitions are permitted by a new axiom, since every transition still has to conform to the axioms which existed before. Hence a new axiom can only restrict the set of histories, or at least leave it as it was.

6-2 Subtypes

- A type S is a *subtype* of a type T , written $S \subseteq T$, iff every member x of S is also a member of T .

$$\frac{x \in S \vdash x \in T}{S \subseteq T}$$

A type may have many subtypes and supertypes. A subtype S may be defined as a subtype of T by inheritance; or it may be proven to be so. Proofs are discussed in a later section; the principle is to demonstrate that all the axioms of T are valid within S .

The purpose in determining subtyping is to check that the behaviour expected by a client as expressed in type T is provided by a reified or more detailed specification as expressed in type S . If $S \subseteq T$, then the client can treat members of S exactly as if they were members of T ; and need not know about S .

6-2.1 Defining a subtype by inheritance

If TT is defined as an inheritor of T , written $TT::+T$, then its axioms are those of T plus any explicitly defined for TT .

- *Subtype by inheritance theorem.* $TT::+T \vdash TT \subseteq T$.
 TT is made up by adding new axioms to T . By the type monotonicity theorem, the result is a subtype of T .

Model-variables and axioms with the same names in \mathbb{T} and T are identified; new names may also be introduced.

A type may be defined to inherit from more than one type — in which case it inherits the axioms and features of them all.

6-2.2 Type product

- $x \in T_1 \cap T_2 \Leftrightarrow x \in T_1 \wedge x \in T_2$

The product or intersection $T_1 \cap T_2$ of two types is their greatest common subtype: $T_1 \cap T_2 \subseteq T_i$.

If a type is defined as $\mathbb{T} ::+(T_1, T_2)$ then it inherits the axioms of both T_1 and T_2 .

- *Multiple inheritance product theorem.* Inheriting from both T_1 and T_2 is equivalent to inheriting from $T_1 \cap T_2$.
The minimal case is when \mathbb{T} adds no axioms of its own. \mathbb{T} is made up by adding the axioms of T_1 to T_2 : by the type monotonicity theorem, the result must be a subtype of T_2 ; and of T_1 by symmetry; hence $\mathbb{T} \subseteq T_1 \cap T_2$. Clearly, since \mathbb{T} restricts the set of histories with no extra axioms, $\mathbb{T} = T_1 \cap T_2$.

Type products are the basis of monotonic composition, because if a client c_1 expects some object $x \in T_1$, then we may supply an object $x \in (T_1 \cap T_2)$ — that is, with not only the behaviour c_1 is expecting, but some further rules derived from another contract, or an improved specification, or the practicalities of implementation. All the reasoning that can be done by a client about its use of T_1 works by inference from the type's axioms; it is not possible to infer anything from the absence of a theorem. Hence when new axioms are added, all existing inferences remain valid.

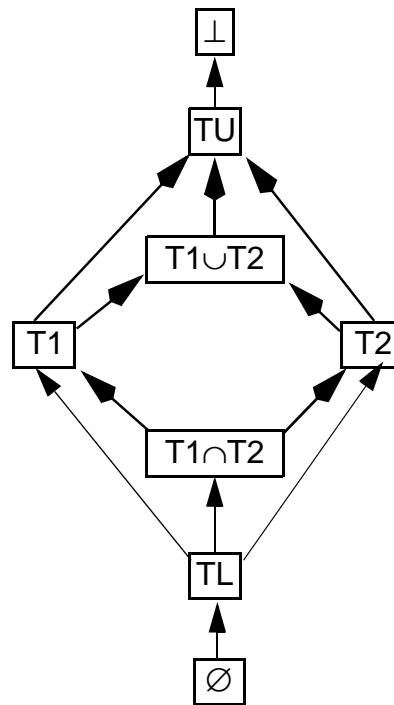
There is no guarantee that the intersection of two types will not be empty, forming an unimplementable empty type with inconsistent axioms.

6-2.3 Co-product and type category

The co-product or union $T_1 \cup T_2$ of two types is their least common supertype: $T_i \subseteq T_1 \cup T_2$. $x \in T_1 \cup T_2 \Leftrightarrow x \in T_1 \vee x \in T_2$. A member of a union conforms to either of the components: so that a union is the least common supertype of its components; and is subtype of any other common supertype (TU in the lattice shown). The only theorems (and hence the only operations) that one can be sure of in a union are those implied by the ' \vee ' of any pair of theorems from the two types; this includes, of course, any theorems which the types have in common.

Where an alternative between several types is expected in some context, designers are encouraged to specify the supertype and allow the range of subtype alternatives to be decided at a later stage: for example, specify **Shape** rather than **Parallelogram** \cup **Square**, because that allows more readily for extension at a later stage. But it is on occasion useful to be able to stipulate the precise range of alternatives: for example, in specifying a tree, it may be fundamental to the model, that branches always sprout two of something which is either a leaf or a node, and, short of a major design review, definitely never anything else. Traditional VDM makes extensive use of discriminated unions, in which each alternative 'knows' what type it is.

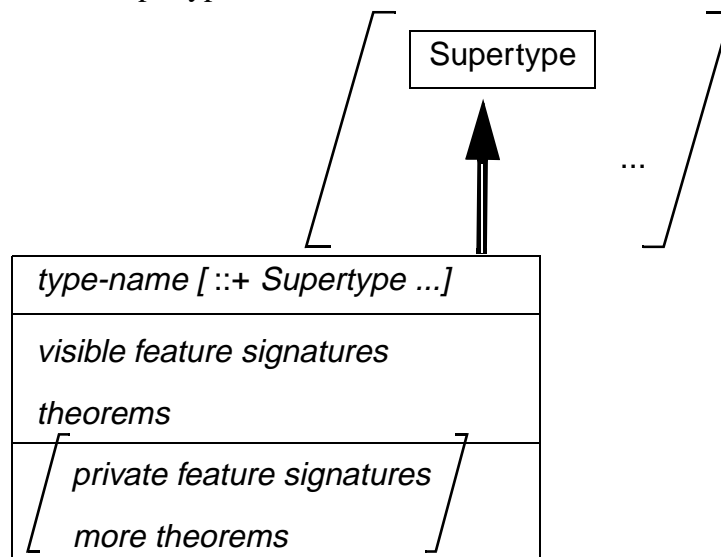
\perp is the axiomless type, supertype of all others; and \emptyset is the unimplementable type, subtype of all others.



TL/TU is an arbitrary common sub/supertype of T1 and T2; it is always a sub/supertype of the intersection/union.

6-3 Type definitions

The ‘display’ syntax for a type description is of this form, with two alternative notations for supertype:



(The disposition of the theorems between the partitions is not formally significant: it’s sometimes useful to distinguish any which make no mention of private components by placing them above the line.)

The abstract syntax:

FrescoType ::+ Context	
var name \in	Symbol
var parents \in	FrescoType set
var axioms \in	Theorem set
var derived \in	Theorem set
var visible \in	Signature set
var private \in	Signature set
var theory \in	Theory
$\forall d \in \text{derived} \cdot d \text{ justification} \neq \text{nil}$	

axioms are the defining opspects of the type; **derived** are further theorems proven from the **axioms**. The Fresco system allows anyone to extend the set of **derived** theorems attached to a type (provided they are proven), since they increase the ease with which the type may be understood, and provide a less primitive base upon which clients may base their own proofs.

visible lists the operations or functions which clients may expect to apply to members of this type; **private** permits the construction of a model, and are usually simple attributes.

private attributes are not hidden from clients: they may be referred to in a client's reasoning. However, they are encapsulated, and cannot be used in code.

Signatures are prefixed with **op**, **fn**, or **var**: operations may change the state and arguments; functions only yield values without changing anything; and variables are equivalent to zero-arity functions. The type-constraints are effectively extra axioms.

6-3.1 Type context extraction

Within the context of a type theory, the name **self** is conventionally used for the variable that represents the receiver object in an opspect. Thus for example, in the theory of **Points**:

$$\{ \text{:- } x = \bar{x} + p \ x \wedge y = \bar{y} + p \ y \} \text{ self move: } p$$

The Fresco convention is to omit **self** as the receiver of an operation: it can systematically be restored by prefixing each term which does not begin with a metavariable or local variable of the context (in a method):

$$p \cdot \{ \text{:- self } x = \overline{\text{self}} \ x + p \ x \wedge \text{self } y = \overline{\text{self}} \ y + p \ y \} \text{ self move: } p$$

The theorems of a type **T** are stated within a context in which **self** \in **T** is axiomatic. We can apply the context-extraction rule to any theorem **A[*self*]**, at the same time renaming **self**, to produce:

$$a \cdot a:\text{T} \vdash A[a]$$

So the example becomes:

$$a, p \cdot a \in \text{Point} \vdash \{ :- a x = \bar{a} x + p x \wedge a y = \bar{a} y + p y \} a \text{ move: } p$$

6-3.2 Model-oriented type definitions

In a conventional model-oriented specifications of mutable objects, each theorem is either an OpSpec or an invariant. The invariants document constant relationships among the attributes, while the opsspecs determine the behaviour of the operations in terms of the attributes.

TrafficLight	
op next \in	()
var red \in	Boolean
var amber \in	Boolean
var green \in	Boolean
$\neg \text{green} = (\text{red} \vee \text{amber})$	
{ red \wedge \neg amber $:-$ red \wedge amber } next	
{ red \wedge amber $:-$ green } next	
{ green $:-$ \neg red \wedge amber } next	

One operation may have to conform to several opsspecs. From a client's point of view, it is not obligatory to satisfy any particular precondition; but the applicability of each opspeg is determined by whether its precondition is fulfilled. So you had better satisfy the precondition of at least one theorem applicable to the operation you propose to call, if you are to have any idea of what the result will be.

6-3.3 Type invariants

Ordinary expressions (not spec-statements) may be used to restrict a model and to define redundant components and auxiliary functions. There may be any number of invariants in one type definition and inherited from parent types, but the invariant of a type is the conjunction of all of these.

In the early stages of analysis, it is common to deal with specifications of immutable concepts (like **FrescoType**, above), which can be written as sets of functions constrained by invariants.

An invariant may be used to constrain the model to exclude inapplicable combinations of states, as in the **TrafficLight** example above. (Thus \neg green is not explicitly required in the postcondition of the first axiom.)

An invariant may be used to add extra ‘views’ of a model:

Point	
op move	∈ (Vector)
op rot	∈ (Real)
var x	∈ Real
var y	∈ Real
var r	∈ Real
var ω	∈ Angle
r ≥ 0	
r × r = x × x + y × y	
cos(ω) × r = x ∧ sin(ω) × r = y	
{ :- x = \bar{x} + v.x ∧ y = \bar{y} + v.y } move(v)	

Here two pairs of variables, x, y and r, ω, are each sufficient as models. Each model is convenient for the specification of a different operation, but the two are coupled by the invariants.

6-3.4 Functions

Public or model functions may be defined. E.g.:

Point	
fn _- _	∈ (Point) Point
p2 ∈ Point ⊢ { :- r ∈ Point ∧ r x = x - p2 x ∧ r y = y - p2 y } r := self - p2	
var distance	∈ (Point) Real
p · p ∈ Point ⊢ distance(p) = (self - p) r	

6-3.5 Property-oriented specifications

Fresco makes no big distinction between property-oriented and model-oriented methods of specification: the difference comes down to whether the axioms make use of internal model components. It is possible to derive non-component-using theorems from component-using axioms. [Fitzgerald90] demonstrates this in connection with non-OO specification modules. A purely axiomatic style is unavoidable for the most primitive types, since they don’t have anything more primitive to build models with.

E.g.

Set[T]	
fn $\emptyset \in$	Set[T]
fn $_<+_ \in$	(Set[T], T) Set[T]
fn $_\cup_ \in$	(Set[T], Set[T]) Set[T]
fn $_\cap_ \in$	(Set[T], Set[T]) Set[T]
$x \cdot x \in T \vdash \neg(x \in \emptyset)$	
$x, y, s \cdot x \in T, y \in T, s \in \text{Set}[T] \vdash y \in (s <+ x) = (y=x \vee y \in s)$	
$x, s1, s2 \cdot x \in T, s1 \in \text{Set}[T], s2 \in \text{Set}[T] \vdash x \in (s1 \cup s2) \Leftrightarrow x \in s1 \vee x \in s2$	
$x, s1, s2 \cdot x \in T, s1 \in \text{Set}[T], s2 \in \text{Set}[T] \vdash x \in (s1 \cap s2) \Leftrightarrow x \in s1 \wedge x \in s2$	

6-3.6 Type Box composition

A type's definition may be spread between more than one type-definition box, for several reasons:

- to separate concerns in documentation; for example, where a type is developed as the intersection of several 'contracts' [Helm].
- in successive versions of a published design, a type may be extended by adding new boxes and changing the implementation.
- in stepwise design, stages of refinement may be represented either by different types, or by extensions to a single type.

The default composition of two boxes is type intersection; a client's knowledge of any box will remain valid no matter what other boxes are added. On the other hand, an implementor must gather specifications from all the boxes applying to any one type, and must of course repeat the procedure whenever any new boxes are added.

The Fresco system ensures that the implementor of an operation has to document proofs of all the relevant axioms from all the relevant boxes; if any relevant axiom is changed or added, the implementation is flagged as needing re-doing. But each client is only dependent on the theorems it uses in its proofs, and a change in a type box flags for re-inspection only such dependent clients. Changes in the implementation of a type which aren't caused by any change in the specifications therefore raise no spurious alarms on the clients. This strict use of dependency information provides accurate configuration control, and provides the maximum benefit of encapsulation.

6-3.7 Signatures

The purpose of the feature signatures is to document which of the operations dealt with by the theorems are visible to clients and therefore must be implemented. Signatures may be interpreted as qualifying the theorems in the rest of the definition.

For example:

Point
$\text{fn } _+ _ \in \quad (\text{Point}) \text{ Point}$ $p, r \cdot \{ \text{:- self} = p - r \} r := \text{self} + p$

Point
$\text{fn } _+ _ \in \quad (\text{Real}) \text{ Point}$ $m, r \cdot \{ \text{:- } r \cdot x = x + m \wedge r \cdot y = y + m \} r := \text{self} + m$

Point
$\text{fn } _+ _ \in \quad (\text{Line}) \text{ Line}$ $n, r \cdot \{ \text{:- } r = n + \text{self} \} r := \text{self} + n$

We can modify the type context extraction rule of §6-3.1 to add signature information to each theory as it is extracted. If ‘ $T_b::$ ’ prefixes a theorem or signature which originates in a box T_b describing a type T , then:

$T_b, R, P_i, n, f, \text{pre}, \text{post} \cdot$

$$\begin{array}{c}
 T_b:: f:(P_i \delta) R \\
 T_b:: r, p_i \cdot \{ \text{pre} \text{:- post} \} r := f(p_i) \\
 \hline
 p_i, x, r \cdot \{ x \in T \wedge p_i \in P_i \wedge \text{pre} \text{:- } r \in R \wedge \text{post} \} r := x.f(p_i)
 \end{array}$$

Notice that this is not the same as translating the signature to some theorem and then composing that theorem with the opspec: preconditions would not be conjoined in such a procedure. Rather, the signature supplements each of the stated opsps. This scheme should be applied to every combination of opspec and theorem in a given box whose operation-names and parameter-counts match.

Notice that this gives more significance in the semantics to

- the type box, which is no longer just a partitioning of the information in a unified type description;
- the opspec, which is no longer just a special case of a specification-statement.

Signatures and subtypes

The above rule clarifies some questions about signature-conformance. If $TT \subseteq T$, then $TT::f$ (that is, an operation f defined for type TT) must conform to any signature-theorems of $T::f$; so any further theorem about $TT::f$ must extend rather than conflict with the inherited theorems. If we have

$$TT:: f \in (P1, P2, P3) RR$$

then the returned result when f is applied to any member of TT must belong to $R \cap RR$, which must therefore be non-empty. $R \cap RR \subseteq R$, of course; in programming languages which handle this issue properly (such as Eiffel), the compiler would insist that $TT::f$ should be specified as yielding some subtype of R . (In C++,

no alterations in result type are allowed, because of implementation difficulties [C++, p.211].)

If we have

$$TT_b:: f \in (PP_1, PP_2, PP_3) R$$

then f applied to instances of TT is able to accept parameters of type PP_i as well as P_i : that is, the i th parameter is of type $PP_i \cup P_i$. This again corresponds to the usual programming language rule, since $P_i \subseteq PP_i \cup P_i$: a function in a subtype has to be able to deal at least with the parameters its parent deals with.

We also allow functions to deal with different parameter types in different ways. Leaving aside the signature notation for a moment, we can have two theorems in the same type-definition which specify different results for different preconditions:

$$\{ p \in P_1 :- \text{post1} \} f(p)$$

$$\{ p \in P_2 :- \text{post2} \} f(p)$$

The two theorems can be implemented by separate operation code-bodies ‘overloading’ the same name f , though if P_1 and P_2 overlap, then f is implementable only if it is possible to satisfy $\text{post1} \wedge \text{post2}$ in the case where $p \in P_1 \cap P_2$. To make the best of the signature notation here, we have to write the two cases in separate type definition boxes:

T
op $f \in (P_1)$ $\{ :- \text{post1} \} f(p)$

T
op $f \in (P_2)$ $\{ :- \text{post2} \} f(p)$

The effect of a signature does not extend beyond the box it is defined in because it is non-monotonic (see 3-4.4.3): if the signature in the left-hand box had an effect on the meaning of the right-hand box, then clients would need to see both in order to be able to use T .

6-3.8 Type inference

The usual methods of inferring that $x \in T$ are:

==type:	$a, b, T \cdot$	$a == b, a \in T \vdash b \in T$	by subs-==
Assign-type:	$\{ y \in T :- x \in T \} x := y$		by assignment
Type-Gen:	$x \in S, S \subseteq T \vdash x \in T$		
Dynamic-special:	$(x \in T, P[x] \vdash x \in T_1) \vdash \{ x \in T \wedge P[x] :- R \} S \vdash \{ x \in T_1 \wedge P[x] :- R \} S$		
yield-type:	$\{ P :- \uparrow \in T \} x \text{ op} \vdash x \text{ op} \in T$		by yield

By conventional axiom within a type-definition of T , $\text{self} \in T$.

The type definitions of many types will include definitions of creation functions such as

$$x \in T \vdash T(x \ a, x \ b, x \ c)$$

and by convention, only creation functions have the same name as a type. So $T(\dots) \in T$.

6-4 Generic types

6-4.1 Constant parameters

It is frequently the case that we specify a type in ideal terms, with no limits on size or other parameters, and yet must implement it on a real machine. If we wish to be honest about this and document the restriction, what is the best way of doing it, and how do we reconcile this with the idea of subtyping? As discussed in §3-4.5 — p.47, it is not appropriate just to add an extra invariant.

Parameterising a type definition with a constant makes it possible to design a type, derive theorems and so on, whilst deferring certain specifics. E.g.:

Stack of: n
fn s ∈ List fn n ∈ Nat { s len < n :- s = \bar{s} <+x } push(x) ...

Given this definition, it would be straightforward and useful to show that

$$n, m \cdot n \in \text{Nat}, m \in \text{Nat}, n \leq m \vdash (\text{Stack of: } m) \subseteq (\text{Stack of } n)$$

so that a longer-capacity Stack can be provided where a shorter one is required.

Only clients and implementors need give particular values for n.

In Fresco, the favoured solution in respect of the numbers and extensible collections is to make believe that the implementation's range is infinite: the axioms will reflect this pretence.

6-4.2 Type parameters

Just as a type is defined by a theory of the membership of a specific type, a generic type is defined by a theory of the membership of the type determined by a type expression: in effect, an axiomatic semantics for type expressions.

We will write type expressions in postfix or infix notation, using type names as constants. The aim is to be able to write expressions like **Set of: Int**, **SortedList of: String**, **List of: Char**, **Map from: Symbol to: Key**. There is first the distinction of mutability to be made. Consider these four kinds of set:

- **Set** is a type of immutable values (or at least, objects representing immutable values): \cup, \cap, \in yield values (= create new objects representing the new values), but do not alter the states of their operands.
- **Set of: T** Like **Set**, but all the members of such a set are of type T.
- **SetContainer** — no parameter — is Smalltalk's mutable **Set**, into which you may add or remove any mixture of types of object.
- **SetContainer of: T** is a type of mutable sets, with restricted preconditions on the add operations. You may only add objects of type T to one of these.

6-4.3 Generic types of immutable values

Set is sufficiently fundamental to merit description using axiomatic methods:

Set
fn $\emptyset \in \text{Set}$ $\forall x \cdot \neg x \in \emptyset$ $\forall x \cdot x \in \text{Set}(x)$ $\forall x \cdot x \in S1 \vee x \in S2 \Leftrightarrow x \in S1 \cup S2$...

Set of: T can be specified as a restriction of **Set**:

Set of: T
$\text{self} \in \text{Set}$ $\forall x \cdot x \in \text{self} \Rightarrow x \in T$

The extra invariant is OK here, because there are no operations which alter the state of any member of the type.

Recalling the definition under 6-1.1, p83, there is no difficulty modifying it to cope with parameters:

$$\frac{
 \begin{array}{c}
 A_{G(T_j), 1} [\text{self} \backslash x] \\
 \vdots \\
 A_{G(T_j), n} [\text{self} \backslash x]
 \end{array}
 }{
 x \in G(T_j)
 }$$

Notice that \emptyset is a member of **Set of T** for any T (since $\emptyset \in \text{Set}$ and $\forall x \cdot x \in \emptyset \Rightarrow x \in T$), and that there is no ground for any discomfort in this (as some bulletin-board commentators have expressed in connection with a variety of languages): \emptyset is the point at which these types all overlap. Notice also that **Set of: \perp** = **Set**, if \perp is the common supertype of all others.

The generic type could of course have been written all in one description, without using an auxiliary unrestricted type.

6-4.4 Generic types of mutable objects

SetContainer is not difficult to specify using the techniques we have seen so far. It can be specified axiomatically from scratch, or in terms of a component of type **Set**; it will contain mutating operations such as:

$$\begin{array}{l}
 x \cdot \{ \} \text{ :- } x \in \text{self} \ \} \text{ add } (x) \\
 x \cdot \{ \} \text{ :- } x \neq y \Rightarrow (\bar{y} \in \text{self} = y \in \text{self}) \ \} \text{ add } (x)
 \end{array}$$

But we cannot specify **SetContainer of: T** as a subtype of plain **SetContainer** for the reasons discussed in §3-4.6 — p.51: clients of the unrestricted **SetContainer** rightfully expect to be able to **add** any mixture of objects, so no member of **Set-**

Container of: T will be suitable (unless $T = \perp$). Hence **SetContainer** of: T must be written from scratch:

SetContainer of: T	
op add \in	(T)
op remove \in	(T)
fn contains \in	$(T) \text{ Bool}$
var $s \in$ Set $\{ x \in T : - s = \bar{s} \cup \text{Set}(x) \} \vdash \text{add}(x)$...	

Plain **SetContainer**, if it is required, can now be defined as

SetContainer \triangleq **SetContainer** of: \perp

6-4.5 Generic types with restricted parameters

SortedList of: T will only work if T has a \leq operator. Its implementation(s) will depend on the properties of \leq between members of T , and so they must be documented as axioms in the specification:

SortedList of: T	
op add \in	(T)
fn get \in	$(\text{Nat}) T$
...	
var $s \in$ List $x \cdot x \in T \vdash x \leq x$ $x, y \cdot x \in T, y \in T \vdash x \leq y \wedge y \leq x \Leftrightarrow x = y$ $x, y, z \cdot x \in T, y \in T, z \in T \vdash x \leq y \wedge y \leq z \Rightarrow x \leq z$ $i, j \cdot i < j \wedge j < \text{self len} \Rightarrow s[i] \leq s[j]$...	

The axioms relating to \leq do not mention **self**, but they nevertheless constrain the implementation, which must satisfy them; so **SortedList** of **Compiler** is just not implementable, and has no members.

It would be tedious to repeat the axioms of \leq in every parameterised type that uses it, so we seek a method of encapsulating those properties in a type which would describe all those objects which can be ordered. The immediately obvious solution does not work, in general:

TotalOrdering	
fn $_ \leq _ \in$	$(\text{TotalOrdering}) \text{ Bool}$
$x \cdot x \in \text{TotalOrdering} \vdash x \leq x$	
...	

The snag is that this implies that any **TotalOrdering** (say, a number) is comparable with any other (say, a string). But whilst these must be disallowed, it must also be

possible for integers and reals, say, to be comparable. The solution (using a trick described in [Meyer]) is another generic type:

TotalOrdering of: T	
fn $_ \leq _$	\in (TotalOrdering of: T) Bool
$x \cdot x \in \text{TotalOrdering of: T} \vdash x \leq x$	
$x, y \cdot x \in \text{TotalOrdering of: T}, y \in \text{TotalOrdering of: T} \vdash x \leq y \vee y \leq x$	
$x, y, z \cdot x \in \text{TotalOrdering of: T}, y \in \text{TotalOrdering of: T}$ $z \in \text{TotalOrdering of: T} \vdash x \leq y \wedge y \leq z \Rightarrow x \leq z$	

Now this type makes no mention of **self**: it is only a wrapper for a theory. Iff the theorems in such a **selfless** type are valid, then it is satisfied by any object — it is equivalent to \perp ; but if not, then it is empty. Asserting that anything — it doesn't matter what — is a member of this type is therefore a way of stating its axioms and thereby importing them into a type definition. This gives a way of abbreviating SortedList of T:

SortedList of: T	
op add	\in (T)
fn get	\in (Nat) T
...	
var s :	List
self \in TotalOrdering of: T	
$i, j \cdot i < j \wedge j < \text{self len} \Rightarrow s[i] \leq s[j]$	
...	

or using one of the subtyping notations:

SortedList of: T ::+ (TotalOrdering of: T)	
op add	\in (T)
fn get	\in (Nat) T
...	
var s :	List
$i, j \cdot i < j \wedge j < \text{self len} \Rightarrow s[i] \leq s[j]$	

which puts the parameter-constraint neatly in the header.

It would clearly be useful to have a facility for renaming operations within types, so as not to be tied to the name \leq for the ordering operation: I wish to form, for example, SortedList[\leq /outranks] of Window.

Lastly, it is worth noting that a **SortedList** does not need to be generic: it only needs to be constrained to accept only comparable objects whilst non-empty:

SortedList	
op add \in	(T)
op remove \in	(Nat)
...	
var s :	List
$i, j \cdot i < j \wedge j < \text{s len} \Rightarrow \text{s}[i] \leq \text{s}[j]$	
$\{ \text{s len} = 0 \vee \text{s}[0] \leq x \vee x \leq \text{s}[0] :- \dots \} \text{ add}(x)$	

If this seems dubious, recall that in an axiomatic system, the validity of an expression is just a question of whether it can be proved. For the case in which a number is to be added to a list already containing strings, an attempt to prove the precondition of **add** will fail, so that the axiom shown cannot be applied: and therefore no client can confidently perform that operation in that case. The fact that the precondition has no defined Boolean value in that case is not relevant.

(It is interesting to ponder whether the application of such a style throughout would make 3-valued logic as in [BCJ84] superfluous.)

6-4.6 Subtyping among generics

$T \subseteq TT$ does not imply that $G \text{ of } T \subseteq G \text{ of } TT$, nor the reverse: a **Set of: Int** supplied instead of a **Set of: Number** is incapable of storing **Reals**; whilst a **Set of: Number** supplied in lieu of a **Set of: Int** may yield contents with which the client cannot deal. However, $(G \text{ of } TT \subseteq GG \text{ of } TT, T \subseteq TT) \vdash G \text{ of } T \subseteq GG \text{ of } T$ (for example when $TT = \perp$).

6-5 Creation and verification of subtypes

6-5.1 Subtyping proofs

Subtyping proofs are applicable:

- to show that a class implements a given type
- to show that a member of a given type is suitable where some other type is expected — as a parameter, or in a variable

Subtyping proofs may often be reduced by inheriting proofs:

- from a superclass which implements a supertype;
- from an earlier version of a class which implemented a supertype.

Subtyping may be proven by showing the axioms of the supertype to be theorems of the subtype. This is complicated where the models of the two types are different; in which case a retrieval relation must be defined between the two models.

There are other, non-subtyping, varieties of relationship between superclass and subclass (i.e. a **ted** which inherits methods and instance variables), and these can

sometimes be interesting from the point of view of proof re-use; but these are not dealt with in detail here.

In a language with function-name overloading (such as C++), it is necessary to decide in which case a method is applicable (so as to free the designer from having to prove the axioms for methods which will not be called). This is not difficult (at least in C++), since all such discrimination is done at compile time, being only signature-dependent.

6-5.2 Varieties and purposes of subtyping

The subtyping assertion $\text{self} \in T$ may either be an axiom or a derived theorem of a type TT .

Asserted subtyping ensures that TT is a subtype of T by fiat of the designer; all the axioms of T are thereby included as if they were axioms of TT .

Derived subtyping represents the conjecture that the TT is a subtype of T , and should be supported by a proof founded on the axioms of TT .

Axiomatic subtyping is used to construct one type from another: all that is necessary is to check that the result is *implementable*. Methods attached to the supertype are usually inherited at the same time, and must be checked for conformance to the new axioms.

Derived subtyping is used when it is appropriate to construct an entirely new model, closer to a feasible implementation.

In either case, subtyping may be *operation-strengthening*, *state-restrictive*, *extensional*, or any combination of the three.

Operation-strengthening adds extra opsspecs to the specification which apply to existing operations, so that they deal with a broader range of prior states, or produce more strongly determined results.

State-restrictive subtyping constrains the state-space of the members by adding extra invariants: so that for example, a general shape is constrained to be a square.

Restrictive subtyping usually appears where a supertype is used to characterise the common aspects of several types — as in a polymorphic system; it is therefore more common in OOP than in traditional development.

Extensional subtyping provides more operations and possibly provides more detail for the state-space, differentiating individual states into substates. For example,

ColouredPoint ::+ Point
fn colour ∈ Colour

$x \in \text{Point}$ implies that the possible states of x can be enumerated by working through the combinations of co-ordinates (to some finite precision!) But **Point** does *not* tell us that its members do not have some other attributes; if we now discover $x \in \text{ColouredPoint}$, then for each state we enumerated before, we can now distinguish variants for every possible colour.

There are three principal purposes of extension:

- again, to specialise a common supertype into variants
- to provide additional features for clients

- to move towards an implementation by adding redundant state components such as caches.

6-5.3 Subtyping and inheritance

Fresco leaves unchanged the normal code-inheritance mechanism of the host programming language. Two kinds of inheritance are permitted: conformant and non-conformant. A conformant subclass is expected to implement a subtype of its parent(s), so subtyping is asserted; non-conformant inheritance is of code only, and no such requirement is imposed.

6-5.4 Implementability

When a type is specified (either new or by inheritance from another) there is no automatic guarantee that there are no contradictions between the axioms — both those explicit in the type definition, and inherited from supertypes. A contradiction would be discovered on attempting an implementation; but since that may happen some way down the development road, it may be wise to perform an implementability check as soon as the type is specified. Implementability is not a proof obligation in Fresco, but the facilities to verify it are provided.

Implementability is not a crucial check from a client's point of view, except that if it is mistakenly asserted in advance of implementation, clients might waste work in the expectation of undeliverable goods; however, no incorrect code will result.

6-5.4.1 Implementability from scratch

The Shape supertype, which documents the common features of several types of mutable shape-representing objects, provides a suitable example.

Shape	
op move	\in (Vector)
op rotate	\in (Angle)
fn position	\in Vector
fn v1, v2, v3, v4	\in Vector //edges
fn p1, p2, p3, p4	\in Vector // vertices
op set_p1, ...set_p4	\in (Vector)
$\dot{\vdash} :- \text{position} = \overline{\text{position}} + v \} \text{move}(v)$	
$\{ :- v1=\overline{v1} \wedge v2=\overline{v2} \wedge v3=\overline{v3} \wedge v4=\overline{v4} \} \text{move}(v)$	
$\{ :- v1=\overline{v1}.\text{rot}(\omega) \wedge v2=\overline{v2}.\text{rot}(\omega) \wedge v3=\overline{v3}.\text{rot}(\omega) \wedge \text{position}=\overline{\text{position}} \} \text{rotate}(\omega)$	
$p1=\text{position} \wedge p2=p1+v1 \wedge p3=p2+v2 \wedge p4=p3+v3$	
$v1+v2+v3+v4 = \mathbf{0}$	
$\{ :- p1 = x \} \text{set_p1}(x) \text{ //etc}$	

To prove consistency, it is only necessary to demonstrate that one implementation can fulfill the axioms; but remember that Fresco theorems cover not single states, but whole histories of behaviour, and it is therefore necessary to prove that for each

operation, each accessible prior state has a permissible ulterior state. Permissible states are those allowed by the invariants; accessible states are all those achievable from some permissible initial state by some sequence of transitions via the operations with any parameters allowed by their preconditions.

The accessible states will generally be a subset of the permissible ones, but the permissible ones are much easier to characterise. The strategy is therefore to prove that for every operation op_i governed by axioms $\{ pre_{i,j} :- post_{i,j} \} op_i$ in a type T whose invariants conjoin to give inv ,

$$\forall_i \cdot \forall \bar{\sigma} \cdot \exists \sigma \cdot \forall_j \cdot \overline{inv(\bar{\sigma})} \wedge \overline{pre_i(\bar{\sigma})} \Rightarrow post_i(\bar{\sigma}, \sigma) \wedge inv(\sigma) \vdash \exists x \cdot x \in T$$

(The ‘ $\forall_{i,j}$ ’ are really schematic conjunctions.) Notice that for each operation, one ulterior state must satisfy all the applicable postconditions simultaneously. The $\bar{\sigma}$ and σ are the before and after versions of the set of parameters and variables forming the model of the type.

Where several opsspecs apply to one operation, it often simplifies matters to separate into distinct cases the regions where the preconditions overlap and where they do not. For example, there are three regions here, only one of which has a potential conflict:

$$\begin{aligned} &\{ x \leq 0 :- r \times r = -x \} r := \text{absSqrt}(x) \\ &\{ x \geq 0 :- r \times r = x \} r := \text{absSqrt}(x) \end{aligned}$$

In the case of **Shape**, I claim that the following arguments could be formalised from the theorems of **Vector** and **Angle**. The invariant is the constraint that the vectors must meet up, together with the relationship between the vectors and points. A degrees-of-freedom argument can convince us that *any* four points can determine three vectors; and, given three vectors, we can always determine a fourth which is the complement of their sum: so we really only need worry about the loop closure constraint. For **set_pi**, nothing is said about the other points, which are free to move if required; for **move**, the looping constraint must remain unaffected if the vectors are unchanged, and the four points are free to follow from the new position; for **rotate**, only one point is fixed, whilst only three of the vectors are explicitly rotated.

6-5.4.2 Implementability and restrictive subtyping

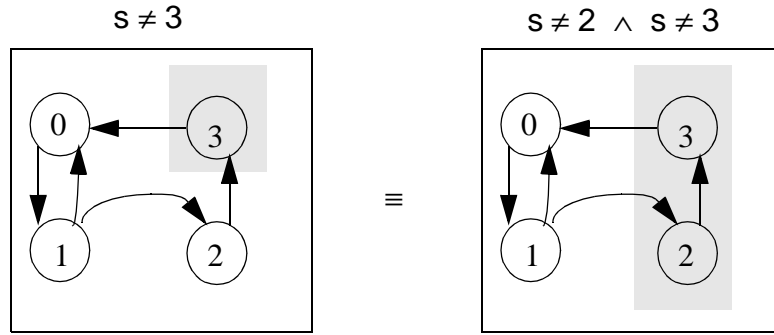
The fact that a state is permitted or accessible in a type does not necessarily imply that it will be permitted or accessible in any subtype. For example:

SquareM ::+ Shape
$v1 + v3 = \mathbf{0} \wedge v1 = v2 \wedge v1 \perp v2$

SquareM conforms to all the theorems of **Shape**, but has fewer states. Whilst it is bound to be a subtype, it is not necessarily implementable. Notice again that a non-empty permissible state-space does not imply implementability, since there may be accessible states for which some applicable postcondition cannot match a subsequent state. Nor is every accessible state necessarily meaningful, since no implementation should allow itself to get up a cul-de-sac:

(No implementation getting into state 2 would be able to satisfy the postcondition for the next call of **op**.)

$$\{:- s = (\bar{s}+1) \bmod 4 \vee \bar{s}=0 \wedge s=1\} \text{ op}$$



Unfortunately, there is little to do but to re-prove implementability as from scratch, for each operation separately. However, it may be that some lemmas proven for earlier implementability checks can be re-used.

6-5.4.3 Implementability and additional opspects

Implementability must be re-proven, but just for the operations affected.

This is the case whether the opspects apply to existing operations or new ones. Both internal consistency, and compatibility with existing invariants and opspects (for the same operation) must be shown.

6-5.4.4 Implementability and extension

If a type has no nondeterminism, then the only subtyping possible is extension: otherwise, restriction of the existing state space or strengthening the postconditions of any of the operations would render it unimplementable.

If the new state components are orthogonal to the old ones — that is, there are no invariants or opspects involving both — then the new components and the operations and any invariants which relate solely to them can be treated in isolation. Adding colour to **Point** is an example.

If there is a relationship between the old components and the existing ones, then it is possible to introduce new invariants over the old components unwittingly:

$$(x>0 \Rightarrow \text{colour}=\text{blue}) \wedge (y>0 \Rightarrow \text{colour}\neq\text{blue})$$

The same applies to postconditions.

6-5.5 Reification

‘Reification’ is the creation of a subtype in which the model differs from that of its supertype. The key to its verification is one or more *retrieval relations* which link new and old model elements. In order to prove subtyping in general, we need to prove that all the axioms AX_T of the supertype T are observed by any member of the subtype ST :

$$\begin{aligned} ST \subseteq T &\Leftrightarrow \forall x \cdot x \in ST \Rightarrow x \in T \\ &\Leftrightarrow (AX_{ST} \vdash AX_T) \end{aligned}$$

The difference in model variables presents a slight complication. It is necessary to choose an extra axiom, the retrieval relation, which interconnects the variables of the two.

retrieve: $\text{vars}_{ST} \cdot AX_{ST} \vdash (\text{vars}_T \cdot \text{retrieval-relation} \vdash AX_T)$

The retrieval must not be such as to constrain T ; so it is also necessary to prove the propriety of the retrieval relation:

adequacy: $\forall \text{vars}_T \cdot \text{INV}_T[\text{vars}_T] \Rightarrow$
 $\exists \text{vars}_{ST} \cdot \text{INV}_{ST}[\text{vars}_{ST}] \wedge \text{retrieve-relation}[\text{vars}_T, \text{vars}_{ST}]$

where INV_T is the conjoined invariants of T .

6-6 Reification example

6-6.1 Supertype: compiler's symbol table

SymbolTable stores associations of Symbols with characteristics (here represented by Ref) within nested contexts, and is a suitable name-server for a compiler of block-structured languages. As the compiler scans a text, declarations are recorded with **define**; entry to and exit from a block should be recorded with **enter** and **exit**; and **find** looks up the first occurrence of a symbol in successively containing contexts. Only one definition of a symbol is allowed per context.

sd	a→r1	h→r4	a→r7	
	b→r2	b→r5		
	g→r3	c→r6		g→r8
	sd@1	sd@2	sd@3	sd@sd.size

Here it is modelled as **sd**, a Stack of Dictionaries each element of which associates Symbols with Refs:

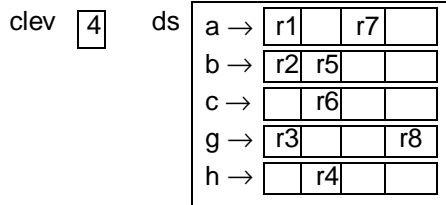
SymbolTable
op enter \in () op exit \in () op define \in (Symbol, Ref) \rightarrow Bool op find \in (Symbol) \rightarrow Ref $\{ \text{sd.size} = \overline{\text{sd.size}} + 1 \wedge (\text{sd} @ \text{sd.size}).\text{size} = 0 \wedge \forall i \in 1..\text{sd.size} \cdot \text{sd}@i = \overline{\text{sd}}@i \} \text{enter}$ $\{ \text{sd.size} \geq 1 \text{ :- } \text{sd.size} = \overline{\text{sd.size}} - 1 \wedge \forall i \cdot i \in 1..\text{sd.size} \cdot \text{sd}@i = \overline{\text{sd}}@i \} \text{exit}$ $\{ s \in (\text{sd} @ \text{sd.size}) \text{ :- } \uparrow = \text{FALSE} \wedge \text{sd} = \overline{\text{sd}} \} \text{define}(s, r)$ $\{ s \notin (\text{sd} @ \text{sd.size}) \wedge \text{ref} \neq \text{Ref.null} \text{ :-}$ $\quad \text{sd.size} = \overline{\text{sd.size}} \wedge \uparrow = \text{TRUE} \wedge \text{sd}@\text{sd.size} = (\overline{\text{sd}}@\text{sd.size}) \uparrow \text{Dict.map}(s, r)$ $\quad \wedge \forall i \in 1..\text{sd.size} - 1 \cdot \text{sd}@i = \overline{\text{sd}}@i \} \text{define}(s, r)$ $\{ \forall i \cdot i \in 1..\text{sd.size} \Rightarrow s \notin (\text{sd}@i).\text{dom} \text{ :- } \uparrow = \text{Ref.null} \wedge \text{sd} = \overline{\text{sd}} \} \text{find}(s)$ $\{ \exists i \cdot i \in 1..\text{sd.size} \wedge s \in (\text{sd}@i).\text{dom} \text{ :- } \text{sd} = \overline{\text{sd}} \wedge \exists i \cdot i \in 1..\text{sd.size} \wedge s \in (\text{sd}@i).\text{dom}$ $\quad \wedge \neg(\exists j \cdot j \in i+1..\text{sd.size} \wedge s \in (\text{sd}@j).\text{dom}) \wedge \uparrow = (\text{sd}@i)@s \} \text{find}(s)$
var sd \in List .of(Dict.of(Symbol, Ref))

(Decoding clues: @ both indexes lists and looks up dictionary contents; n..m is the set of integers in that range; d.dom is the domain of a Dictionary;

$(d1 \upharpoonright d2).dom = (d1.dom \cup d2.dom); k \in d2.dom \Rightarrow (d1 \upharpoonright d2)@k = d2@k;$
 $k \notin d2.dom \Rightarrow (d1 \upharpoonright d2)@k = d1@k .)$

6-6.2 Refinement: Dictionary of Stacks

This is a refinement of **SymbolTable**. The operations and their signatures are the same, and though the internal model is entirely different, we hope to prove that the externally observable behaviour is the same.



The Dictionary of Stacks model illustrated above moves a step towards a more efficient structure, since only one dictionary-lookup will be required per find. Further refinements should be expected before realisation as code, but proceeding in smallish steps makes verification easier. The following type-definition goes one step further: the dictionary-entry for each symbol is reduced from a complete stack to a set of (Nat, Ref) pairs corresponding to the nonblank stack entries; **clev** records the notional size of all the stacks:

StackDict
<p>op enter \in $()$</p> <p>op exit \in $()$</p> <p>op define \in $(\text{Symbol}, \text{Ref}) \rightarrow \text{Bool}$</p> <p>op find \in $(\text{Symbol}) \rightarrow \text{Ref}$</p> <p>$\{ \text{:- clev} = \overline{\text{clev}} + 1 \wedge \text{ds} = \overline{\text{ds}} \} \text{ enter}$</p> <p>$\{ \text{clev} \geq 1 \text{ :- clev} = \overline{\text{clev}} - 1 \wedge \text{ds}.dom = \overline{\text{ds}}.dom \wedge$ $\forall s, i, r \cdot s \in \text{ds}.dom \wedge i \leq \text{clev} \Rightarrow (\langle i, r \rangle \in (\overline{\text{ds}}@s) \Leftrightarrow \langle i, r \rangle \in (\text{ds}@s)) \} \text{ exit}$</p> <p>$\{ \exists rr \cdot \langle \text{clev}, r \rangle \in \text{ds}@s \text{ :- } \uparrow = \text{FALSE} \wedge \text{ds} = \overline{\text{ds}} \} \text{ define}(s, r)$</p> <p>$\{ \neg \exists rr \cdot \langle \text{clev}, r \rangle \in \text{ds}@s \wedge \text{ref} \neq \text{Ref.null} \text{ :- } \text{ds}.dom = \overline{\text{ds}}.dom \cup \{s\} \wedge \uparrow = \text{TRUE} \wedge$ $\forall ss \in \text{ds}.dom \cdot ss \neq s \Rightarrow \text{ds}@ss = \overline{\text{ds}}@ss \wedge \text{ds}@s = (\overline{\text{ds}}@s) \cup \langle \text{clev}, r \rangle \} \text{ define}(s, r)$</p> <p>$\{ s \notin \text{ds}.dom \vee \text{ds}@s = \emptyset \text{ :- } \uparrow = \text{Ref.null} \wedge \text{ds} = \overline{\text{ds}} \} \text{ find}(s)$</p> <p>$\{ s \notin \text{ds}.dom \vee \text{ds}@s = \emptyset \text{ :- } \text{sd} = \overline{\text{sd}} \wedge \exists i \cdot \forall j, rr \cdot \langle i, rr \rangle \in \text{ds}@s \Rightarrow j < i \vee rr = \uparrow \} \text{ find}(s)$</p>
<p>var ds \in $\text{Dict.of}(\text{Symbol}, \text{Set.of}(\text{Tuple.of}(\text{Nat}, \text{Ref})))$</p> <p>var clev \in Nat</p>

6-6.3 Verifying refinement

We wish to prove that any member of **StackDict** is also a member of **SymbolTable**: that is, that **StackDict** is a subtype (in the strict Fresco sense) of **SymbolTable**. This is true if and only if every theorem that a client can infer about **SymbolTable**-members is also true of **StackDict**-members. If the two types had identical models, (or if the subtype's model was an extension of the supertype's) it would be a matter

of proving that the axioms of the supertype were derivable theorems of the subtype.
An appropriate statement of the retrieval relation is:

$$\begin{aligned} & \text{sd.size} = \text{clev} \wedge \\ & \forall i, s, r \cdot i \in 1.. \text{clev} \Rightarrow \\ & \quad (\text{sd}@i@s = r \wedge s \in (\text{sd}@i).\text{dom} \Leftrightarrow \langle i, r \rangle \in \text{ds}@s \wedge s \in \text{ds.dom}) \end{aligned}$$

The illustration below deals with one of the simpler operations from the example.

$$\begin{aligned} \mathbf{h1} \quad & \text{sd.size} = \text{clev} \wedge \forall i, s, r \cdot i \in 1.. \text{clev} \Rightarrow \\ & \quad (\text{sd}@i@s = r \wedge s \in (\text{sd}@i).\text{dom} \Leftrightarrow \langle i, r \rangle \in \text{ds}@s \wedge s \in \text{ds.dom}) \\ \mathbf{h2} \quad & \{ \text{clev} \geq 1 :- \overline{\text{clev}} = \text{clev} - 1 \wedge \text{ds.dom} = \overline{\text{ds}}.\text{dom} \wedge \\ & \quad \forall s, i, r \cdot s \in \text{ds.dom} \wedge i \leq \text{clev} \Rightarrow (\langle i, r \rangle \in (\overline{\text{ds}}@s) \Leftrightarrow \langle i, r \rangle \in (\text{ds}@s)) \} \text{ exit} \\ \mathbf{1} \quad & \text{sd.size} = \text{clev} \quad \text{from h1 by } \wedge\text{-elim} \\ \mathbf{2} \quad & \forall i, s, r \cdot i \in 1.. \text{clev} \Rightarrow (\text{sd}@i@s = r \wedge s \in (\text{sd}@i).\text{dom} \\ & \quad \Leftrightarrow \langle i, r \rangle \in \text{ds}@s \wedge s \in \text{ds.dom}) \quad \text{from h1 by } \wedge\text{-elim} \\ \mathbf{3\cdot h} \quad & \text{sd.size} \geq 1 \\ \mathbf{3} \quad & \vdash \text{clev} \geq 1 \quad \text{from 3-h, 1 by subs-eq} \\ \mathbf{4\cdot h1} \quad & \text{clev} = \overline{\text{clev}} - 1 \wedge \text{ds.dom} = \overline{\text{ds}}.\text{dom} \wedge \\ & \quad \forall s, i, r \cdot s \in \text{ds.dom} \wedge i \in 1.. \text{clev} \Rightarrow (\langle i, r \rangle \in (\overline{\text{ds}}@s) \Leftrightarrow \langle i, r \rangle \in (\text{ds}@s)) \\ \mathbf{4\cdot 1} \quad & \text{clev} = \overline{\text{clev}} - 1 \quad \text{from 4-h1 by } \wedge\text{-elim} \\ \mathbf{4\cdot 2} \quad & \text{ds.dom} = \overline{\text{ds}}.\text{dom} \quad \text{from 4-h1 by } \wedge\text{-elim} \\ \mathbf{4\cdot 3} \quad & \forall s, i, r \cdot s \in \text{ds.dom} \wedge i \in 1.. \text{clev} \Rightarrow (\langle i, r \rangle \in (\overline{\text{ds}}@s) \Leftrightarrow \langle i, r \rangle \in (\text{ds}@s)) \\ & \quad \text{from 4-h1 by } \wedge\text{-elim} \\ \mathbf{4\cdot 4\cdot h1} \quad & i \cdot \quad i \in 1.. \text{sd.size} \\ \mathbf{4\cdot 4\cdot 1} \quad & i \in 1.. \text{clev} \quad \text{from 4\cdot 4\cdot h1, 1 by subs-eq} \\ \mathbf{4\cdot 4\cdot 2\cdot h} \quad & s \cdot \quad s \in (\text{sd}@i).\text{dom} \\ \mathbf{4\cdot 4\cdot 2\cdot 1\cdot h} \quad & r \cdot \quad (\text{sd}@i)@s = r \\ \mathbf{4\cdot 4\cdot 2\cdot 1\cdot 1} \quad & s \in \text{ds.dom} \wedge \langle i, r \rangle \in (\text{ds}@s) \\ & \quad \text{from 4\cdot 4\cdot 1, 4\cdot 4\cdot 2\cdot h, 4\cdot 4\cdot 2\cdot 1\cdot h, 2 by } \forall \Rightarrow \\ \mathbf{4\cdot 4\cdot 2\cdot 1\cdot 2} \quad & s \in \text{ds.dom} \wedge \langle i, r \rangle \in (\overline{\text{ds}}@s) \text{ from 4\cdot 4\cdot 2\cdot 1\cdot 1, 4\cdot 3 by } \forall \Rightarrow \\ \mathbf{4\cdot 4\cdot 2\cdot 1\cdot 3} \quad & s \in \overline{\text{ds}}.\text{dom} \quad \text{from 4\cdot 4\cdot 2\cdot 1\cdot 1, 4\cdot 2 by } \wedge\text{-elim, Set-eq} \\ \mathbf{4\cdot 4\cdot 2\cdot 1} \quad & (\overline{\text{sd}}@i)@s = r \\ & \quad \text{from 4\cdot 4\cdot 2\cdot 1\cdot 2, 4\cdot 4\cdot 2\cdot 1\cdot 3, 4\cdot 4\cdot 1, 2 by } \forall \Rightarrow \\ \mathbf{4\cdot 4\cdot 2} \quad & (\text{sd}@i)@s = (\overline{\text{sd}}@i)@s \quad \text{from 4\cdot 4\cdot 2\cdot 1 by Trans-eq} \\ \mathbf{4\cdot 4} \quad & \vdash \text{sd}@i = \overline{\text{sd}}@i \quad \text{from 4\cdot 2, 4\cdot 4\cdot 2 by Dict-eq} \\ \mathbf{4\cdot 5} \quad & \forall i \cdot i \in 1.. \text{sd.size} \Rightarrow \text{sd}@i = \overline{\text{sd}}@i \quad \text{from 4\cdot 4 by } \Rightarrow\text{-intro} \\ \mathbf{4\cdot 6} \quad & \text{sd.size} = \overline{\text{sd}}.\text{size} - 1 \quad \text{from 4\cdot 1, 1 by subs-eq} \\ \mathbf{4} \quad & \vdash \text{sd.size} = \overline{\text{sd}}.\text{size} - 1 \wedge \forall i \cdot i \in 1.. \text{sd.size} \Rightarrow \text{sd}@i = \overline{\text{sd}}@i \\ & \quad \text{by } \wedge\text{-intro from 4\cdot 6, 4\cdot 5} \\ \vdash \quad & \{ \text{sd.size} \geq 1 :- \text{sd.size} = \overline{\text{sd}}.\text{size} - 1 \wedge \forall i \cdot i \in 1.. \text{sd.size} \cdot \text{sd}@i = \overline{\text{sd}}@i \} \text{ exit} \\ & \quad \text{by refine from h2, 3, 4} \end{aligned}$$

6-7 Creation functions

Creation functions are used to cite an instance of an object in a particular state. The specifier or designer may define a variety of creation functions for one type. A creation function is applied to the type of which it creates members. For example,

$$\begin{aligned} l1 \in \text{Line}, l2 \in \text{Line} &\vdash \text{Point.intersect}(l1, l2).\text{lies_on}(l1) \wedge \\ &\quad \text{Point.intersect}(l1, l2).\text{lies_on}(l2) \\ x \in \text{Real}, y \in \text{Real} &\vdash \text{Point.xy}(x, y) \ x = x \wedge \text{Point.xy}(x, y) \ y = y \\ r \in \text{Real}, r \geq 0, \omega \in \text{Angle} &\vdash \text{Point.rw}(r, \omega) \ r = r \wedge \text{Point.rw}(r, \omega) \ \omega = \omega \end{aligned}$$

Contrast this scheme with VDM, in which there is one creation function for each type, **mk-TypeName**; the types of its parameters are those of the structural components of the type. In Fresco, this would not be convenient, as a type may have many redundant private features, including those inherited from reified types.

It is axiomatic that a creation function creates a new object; this is discussed under §8-3.3.7 — p.145.

If the creation function is intended for the purposes of specification, the result need not be as fully determined as a real instance would be. For example, all implemented members of **Shape** are actually quadrilaterals, squares, etc; but it might nevertheless be useful to define a function **Shape.vertices**(p1,p2,p3,p4) to stand as an abstraction of all possible shapes with those vertices:

$$\begin{aligned} \{ &:- i = \uparrow.p1 \wedge j = \uparrow.p2 \wedge k = \uparrow.p3 \wedge l = \uparrow.p4 \} \text{Shape.vertices}(i, j, k, l) \\ \{ &:- \uparrow = \text{Shape.vertices}(p1+v, p2+v, p3+v, p4+v) \} \text{replicate}(v) \end{aligned}$$

Creation functions are not inherited in any useful sense. If we define **ColouredPoint** as a subtype of **Point**, the theorems about the **Point** creation functions are inherited, but still producing **Points**, not **ColouredPoints**. A new set must therefore be made for every type. The same principle applies if we use a common function name such as **deepCopy** for every copying operation: we need to add information at each subtype about how the new material is copied.

A creation function is not part of a type definition, since it is not part of the behaviour of any object. Axioms defining creation functions are stated within the context of some capsule. It is automatically axiomatic that for every type-name **T** and every function **f**

$$T.f(p_i) \in T$$

and that **T.f** is pure.

6-8 Types and classes

6-8.1 Syntactical considerations

In Fresco, no strong distinction is made between types and classes. (The distinction between inheritance and subtyping is far more important.) One syntactic framework, the *type/class description* (TCD), serves for both. In addition to the syntax we have seen hitherto,

- methods may be attached to the TCD.
- superclasses may be declared. There are three kinds of inheritance:
 - concrete variables and methods only (corresponding to ordinary inheritance in Smalltalk)
 - type information only: supertype by assertion $\text{self} \in \text{supertype}$; any code attached to supertype re-implemented in this
 - conformant inheritance: both type and class inherited

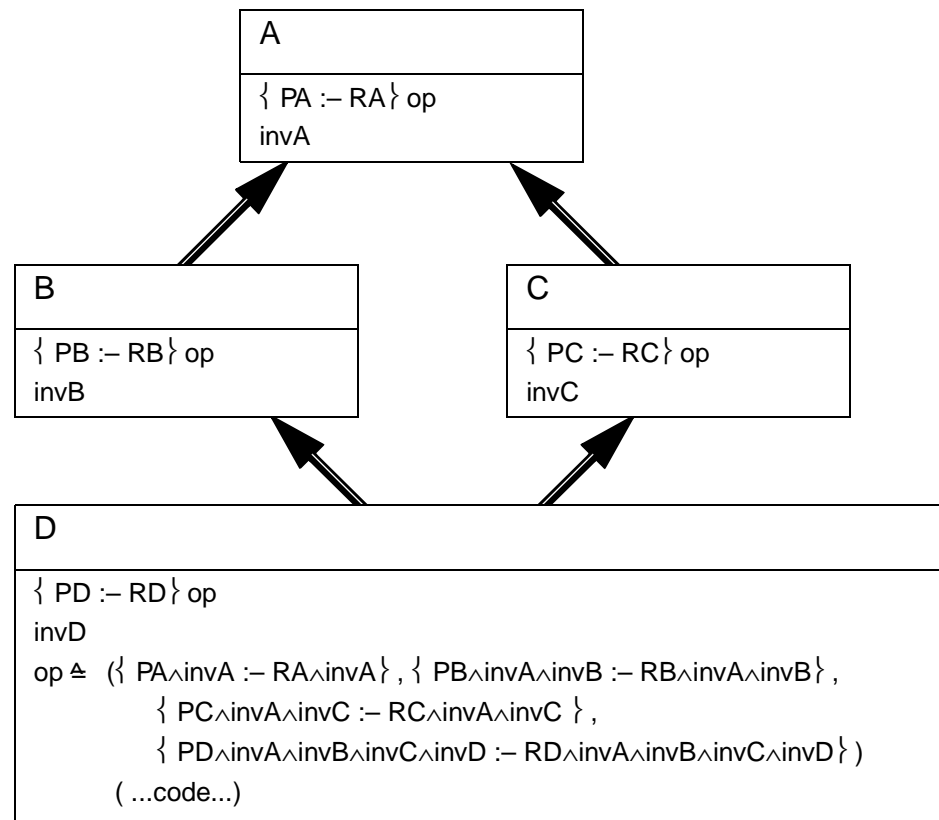
Attached methods, concrete variables and implementation-inheritance are dealt with by the compiler or interpreter just as they are in the unadulterated programming language. Typing constraints in parameter and variable declarations are ignored by the compiler. Conformant inheritance reduces the amount of proof required, and is generally recommended. (The next chapter includes details of how proof obligations are determined.)

6-8.2 Theorems and concrete features

An advantage of mixing types and classes in the same structure (instead of separating them as they are, for example in Abel [Dahl] or POOL [America]) is that opspeccs can be applied to methods, and invariants to concrete variables. (Eiffel has this advantage.)

Each axiom of a type T must be proven to be conformed to by any class that claims to implement it. If subtyping is asserted ($\text{self} \in S$), all the axioms of the supertype S must be conformed to as well (except any which are proven as theorems of T). Where several axioms apply to one operation-name, each must be proven of the

Fig. 9. A method conforms to all the inherited specs



same operation; this may be done by performing separate proofs for each one, or by conjoining them into one axiom first.

Implementation and proofs of operations marked as ‘abstract’ — specified only for the convenience of promotion — may be omitted.

Each type/class description may contain a mixture of specification and implementation. Where there are methods attached to a tcd, they should implement the relevant axioms. A class is *completely implemented* iff all the non-abstract operations specified by its axioms (including those inherited by any subtyping axiom) have methods, and all the invocations to self in its methods are provided for by methods. (If a policy of complete proof is followed, then private functions will also implement axioms, used in decomposition proofs.) Fresco can detect incomplete implementations as a certification check, and should disallow the definition of creation methods in these cases (though abstract creation functions are allowed for the purpose of quoting instances in specifications).

A TCD which provides methods for only some of its public features is an ‘abstract class’, which cannot have instances of its own. Such TCDs are allowed because it is often convenient to provide a partial implementation in a superclass.

6-9 Summary

Type definitions have been given a semantics in terms of theorems incorporating opspecs, which are predicates over behaviour. A type is a set of objects which can be shown to behave always according to a given set of such theorems. Type descriptions may be model-oriented, defining visible behaviour in terms of the mutual effects of externally applied operations and internal components. Model components may be hypothetical, or they may correspond to actual variables. Types may contain methods; the executable component of a type is a class in the ordinary sense.

The composition of type descriptions is based on conjunction, which is monotonic in respect of signature, theorems, and model components. It is therefore easy

- to extend a type description to form a subtype;
- to compose multiple supertypes;
- to extend a type description as a reification;
- to compose partial type descriptions which take part in different contracts;
- to make a compatible extension of a type in a new version of a system.

In all of these cases, anything proven from the original type(s) — for example, the correct implementation of a client — will remain valid for the resulting type. There is no guarantee that such a conjunction is implementable, but this may be proven either by adequacy proofs, or by the process of verified implementation.

We have also seen:

- how to understand conformance of signatures and class behaviour in terms of type extension
- how to understand reification in terms of proof of axioms from the axioms of the reifying type, together with a retrieve relation.

Finally, reification proofs were described.

This completes a framework in which types and classes and verification can be done. Chapter 7 will deal with the packaging of type and class descriptions into capsules, and the mechanics of checking the correctness of capsules. Chapter 8 will discuss the particular problems of reasoning about systems of objects.

7 System composition

The capsule is the basic unit of software design, change, transportation and re-use in the Fresco scheme of things. As we have seen, classes do not always form the best such modules; capsules can carry new software, extensions to existing software, or just specifications.

The way in which a type-description focusses on one particular object ('self') is consonant with the object-oriented style of programming. The principal advantage in programming is that this encapsulation limits the spread of interdependencies between software components, which could otherwise inhibit the reconfigurability and re-use of the components. In specification, the focussing on a particular object has the pragmatic advantage that it is easy to think in the same terms as the programmer; it permits the independent specification of an exportable chunk of software; and again, it tends to limit the scope of the description in order to make descriptions separable.

There are two drawbacks to these forms of encapsulation. Firstly, the natural boundaries of the behaviour you want to specify are not always best drawn around a single object: it may be more natural to describe the relationships between several objects in a 'framework'; Fresco's capsule system does not assume that classes are the natural units of specification or design. Secondly, in order to verify that the prescriptions of the specification of an object are observed, it is sometimes necessary to look beyond the boundaries of that object; especially in connection with possible aliasing.

This chapter defines a semantics for capsules, and shows how a system is composed from capsules.

The constraints on the development, publication, and composition of capsules are described, showing how these constraints (some of which are mechanically enforceable, whilst some are proof obligations) prevent interference between capsules.

The Fresco notation is used here; there is a summary of the notation in Appendix A.

7-1 Systems are compositions of capsules

7-1.1 Systems

There are three interrelated ways of looking at a Fresco system:

System	
var	ev: Execution_View
var	tv: Theory_View
var	cv: Capsule_View
ev = cv.strip \wedge tv = cv.theories	

Capsules are the units whereby design effort is distributed between different users. A capsule may define new type/class descriptions or extend existing ones. The **Capsule_View** of a system is a sequence of **Capsules** — in the order in which they were incorporated into the system.

The **Execution_View** *ev* is what the interpreter or compiler sees: globals and classes with methods and instance variables. This is unchanged from the host programming language; we shall concentrate on Smalltalk in this chapter. In Fresco, everything is defined within some capsule; so *ev* is precisely determined by *cv*. In addition to executable declarations and code, *cv* contains the type specifications and proofs discussed in earlier chapters.

The **Theory_View** is what the developer deals with whilst building and reasoning about a system: the theories are fairly directly related to the information in the capsules.

Capsules can become part of a system in two ways:

- *Creation* and development by a designer; the only way to create new software and theory is in some capsule.
- *Incorporation* (from a library or distribution system) by a prospective client designer.

7-1.2 Capsules

The order of incorporation determines how methods defined in the capsules may overwrite one another. Each capsule must be preceded by its imports. The Kernel is always the first capsule in any system. For convenience, we include in the model both the sequence and a mapping from names.

Capsule_View
<pre> var s : Map from: CapsuleName to: Capsule var ordering : List of: Capsule 'Kernel' ∈ s.dom ∧ ∀ n · n ∈ s.dom ⇒ s(n).fullName.short = n s rng = ordering.elems ∀ i ∈ 2..ordering.length · ordering(i).imports ⊆ {ordering(j).fullName j ∈ 1..i-1 }</pre>

An essential component of a **Capsule** is the informal description, which contains at least some of the formal definitions scattered about in its text (just as this chapter does). The whole set of formal definitions, **defns**, will be accessible with a browsing tool.

Capsule
<pre> var fullName : CapsuleFullName var imports : Set of: CapsuleFullName var status : {Developing, Certified, Published} var description: CapsuleText /* contains Definitions */ var defns : Definitions</pre>

Each capsule has a unique name: the full name includes sufficient information about machine of origin, author and date to ensure uniqueness on a worldwide basis. A short form may be assigned for use in any particular system: either the default abbreviation (which excludes all the origins information) or a name chosen when the capsule is incorporated. The Fresco user interface will translate between the short and full forms.

Once a capsule is fully developed, the designer can ask Fresco to *certify* it: that is, to perform a series of checks upon the consistency and completeness of its proofs. A capsule cannot be altered without losing its certified status. Only a certified capsule should be published, and once published, it cannot be altered (except under a new name); and so a particular name is always guaranteed to refer to the same capsule.

The full names are used to identify imports. Each capsule uses definitions and theorems from the capsules it imports, so a capsule cannot be incorporated into a system until all of its imports have first been incorporated. A capsule may not, of course, transitively import itself.

For capsules KA and KB, “KB«KA” abbreviates “KB imports KA”.

A capsule’s imports list may be edited by the designer who creates it.

7-1.3 Definitions in a capsule

A capsule contains definitions which may be entirely new, or may augment those in an imported capsule.

Definitions			
var	types:	TypeName mapTo: TypeClassDefn	
var	lemmas:	ThmLabel mapTo: JustifiedTheorem	
var	globals:	VarName mapTo: GlobalDefn	
var	methods:	TypeName mapTo: MethodDefn	
$\forall t, l, g \cdot t \in \text{types.dom} \wedge l \in \text{lemmas.dom} \wedge g \in \text{globals.dom} \Rightarrow$ $(\text{types}@t).\text{name} = t \wedge (\text{lemmas}@l).\text{name} = l \wedge$			

The **types** carry specification and data structure whilst the **methods** are separated out for convenience; the **globals** are concrete variables which act as the roots of the system’s data structures; the **lemmas** include any inferences the designer wishes to prove which might be useful to a client.

A designer may not remove anything declared in another capsule.

Whilst type-definitions and theorems always augment those from other capsules, methods overwrite those in preceding capsules (Fig. 10.) The order in which capsules are incorporated into a system is therefore more important in respect of the resulting executable system, than in respect of the specifications.

Everything within a capsule can refer only to types, globals or theorems defined within itself or its imports. (Although there are no restrictions on what operations may be called in a method, its proof will have to refer to theorems about those operations.) A capsule KB must therefore import another KA if:

- KB is intended to implement or refine KA;

- KB is a client of KA.

A **TypeClassDefn** collects together the **TypeBoxes** for a given type scattered throughout the capsule description:

TypeClassDefn	
var	name: TypeName
var	parameters: TypeName list
var	boxes: TypeBox set
$\forall b \cdot b \in \text{boxes} \Rightarrow b \text{ name} = \text{name}$	

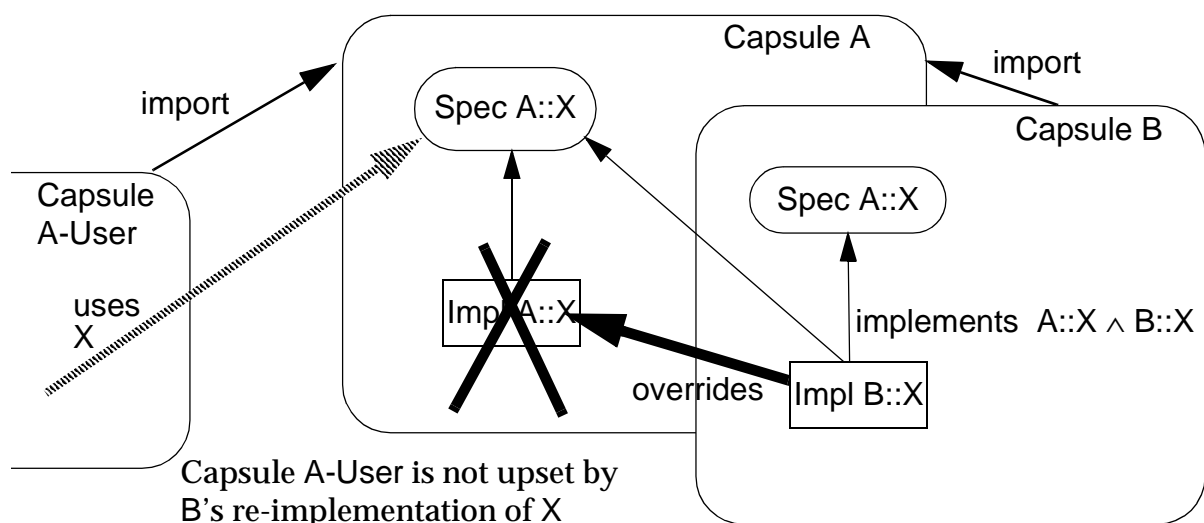
Each **TypeBox** has a public signature and private model (including the supertypes and superclasses respectively), each of which implies axioms in addition to the designer's explicitly defined axioms:

TypeBox	
var	name: TypeName
var	sig: TypeSignature
var	model: TypeModel
var	axioms: Theorem set

Globals will be dealt with in section 7-4 — p.120.

A method is attached to a specific class. In Smalltalk, the class of the receiver is sufficient to determine which method is executed, but typed base languages with overloading also require static type information about the other arguments:

Fig. 10. Capsule composition conjoins specs and overrides implementations



MethodDefn			
var	class:	TypeName	
var	name:	MethodName	
var	signature:	TypeExprn	/* for overloading */

7-2 Capsule composition

7-2.1 The executable system

The compiler/interpreter sees classes and other globals. It represents the static part of the system.

Execution_View	
var	classes: ClassName mapTo: Class

This is derived from the capsule view by superposing successive capsules in the incorporation sequence:

$$\begin{aligned}
kv \cdot kv \in \text{CapsuleView} \quad &\vdash \quad kv \text{ strip} \in \text{Execution_View} \quad \wedge \\
&kv \text{ strip classes dom} = (kv \text{ s all: } [k \cdot (k \text{ defns types dom}) \text{ union}])) \quad \wedge \\
&kv \text{ strip globals} = (kv \text{ s all: } [k \cdot k \text{ defns globals dom}] \text{ union}) \quad \wedge \\
&(\forall cn, c \cdot c = (kv \text{ strip classes @ } cn) \Rightarrow \\
&\quad c \text{ data} = (kv \text{ s all: } [k \cdot (k \text{ defns types @ } cn) \text{ classData}] \text{ union}) \quad \wedge \\
&\quad c \text{ methods} = ((kv \text{ s all: } [k \cdot (k \text{ defns methods @ } cn) \text{ dom}]) \text{ union} \\
&\quad \text{mapAll: } [mn \cdot (kv \text{ ordered msgs: } mn \text{ inClass: } cn) \text{ last}]))
\end{aligned}$$

where classData extracts model information from a type definition. msgs:inClass: extracts the list of methods defined by successive capsules for a particular operation.

$$\begin{aligned}
kl, mn, cn, klmd \cdot &kl \in (\text{List of: Capsule}), klmd = kl \text{ head defns methods} \quad \vdash \\
&(kl \text{ msg: } mn \text{ inClass: } cn) = \\
&\quad ((cn \in klmd \text{ and: } [mn \in klmd @ cn]) \\
&\quad \text{ifTrue: } [List(klmd @ cn @ mn) ++ (kl \text{ tail msg: } mn \text{ inClass: } cn)] \\
&\quad \text{ifFalse: } [kl \text{ tail msg: } mn \text{ inClass: } cn])
\end{aligned}$$

7-2.2 The specifications

Theory_View = Set of: Theory

Every incorporated capsule engenders a new theory:

$$cv \in \text{CapsuleView} \quad \vdash \quad cv \text{ theories} = (cv \text{ s all: } [k \cdot k \text{ name}]) \text{ mapAll: } [n \cdot (cv \text{ s @ } n) \text{ theory}]$$

Recall (§4-1 — p.57) the principal components of a Theory:

Theory		
var	name:	TheoryName
var	imports:	TheoryName set
var	sorts:	SortDecl set
var	consts:	ConstDecl set
var	axioms:	Theorem set

The import structure of the theory is the same as the import structure of the capsules: so everything defined and inferred about the contents of a capsule's imports is also known in its own theory.

$k, th, def \cdot k \in \text{Capsule}, th = k \text{ theory}, def = k \text{ defns} \vdash$
 $th \text{ name} = k \text{ name}$
 $th \text{ imports} = k \text{ imports}$

The sort and constant names are the type and global names. Globals include both variable and global function definitions; in particular, the latter include creation functions. The names of these items are all qualified with the name of the capsule (though this is hidden from the designer by the user interface): the expression $x \text{ in: } k$ represents the definition with all occurrences of names qualified with the name of capsule k .

... $th \text{ sorts} = \text{def types all: } [t \cdot (t \text{ in: } k) \text{ sortDecl}]$
 $th \text{ consts} = \text{def globals all: } [g \cdot (g \text{ in: } k) \text{ constDecl}]$

$t, c \cdot t \in \text{TypeClassDefn} \vdash t = (\text{SortDecl name: name arity: } t \text{ parameters length})$

$g, c \cdot g \in \text{GlobalDefn} \vdash g = (\text{ConstDecl name: name arity: } g \text{ signature parms length})$

The axioms are derived from the definitions:

$k, th, def \cdot k \in \text{Capsule}, th = k \text{ theory}, def = k \text{ defns} \vdash$
 $th \text{ axioms} = (\text{def types all: } [t \cdot (t \text{ in: } k) \text{ axioms}])$
 $\cup (\text{def globals all: } [g \cdot (g \text{ in: } k) \text{ axioms}])$
 $\cup (\text{def methods all: } [m \cdot (m \text{ in: } k) \text{ axioms}])$

Global-definition axioms say that the global belongs to the declared type; method-definition axioms assert the association of a piece of code with an operation within a given class. A requirements axiom is derived from each type definition, asserting equivalence of " $x \in K::T$ " (where $K::T$ is the basic type name qualified with this capsule name) with conformance to the axioms of the type:

$t \cdot t \in \text{TypeClassDefn} \vdash t \text{ axioms} = (\text{Set with: } (t \text{ in: } c) \text{ requirements})$

The theorems are:

- those defined and proved by the user ("lemmas");
- theorems per typebox telling clients what the properties are individually, of form " $x \in K::T \vdash \dots$ ";
- proof expectations generated by Fresco automatically, which the designer has to prove (see below).

$k, th, def \cdot k \in \text{Capsule}, th = k \text{ theory}, def = k \text{ defns} \vdash$
 $th \text{ theorems} = k \text{ lemmas} \cup (\text{def types all: } [t \cdot (t \text{ in: } k) \text{ theorems}])$

7-3 Capsule certification and incorporation

7-3.1 Changes to theorems

In general, if we have a graph of justified theorems, and we remove one of the theorems, then we must find the theorems which are dependent on it, and either remove them or generate alternative proofs for them. By contrast, adding new axioms or proving new theorems creates no such problem.

In Fresco, the central concern is object behaviour, characterised by types. A type definition implies two theorems (§6-1 — p.83): one set of properties used by clients, all of form

$$\text{T-PROP:} \quad x \in T \vdash T::\text{THM}[x]$$

where $T::\text{THM}[\text{self}]$ is any theorem of T ; and a single requirement which implementors and reifiers must fulfill, of form

$$\text{T-RQMTS:} \quad x \in T \Leftrightarrow (\wedge T::\text{AX}[x])$$

where $T::\text{AX}[x]$ are the axioms of T (possibly including subtyping axioms).

If we now add an axiom to T , the old **T-PROPs** all remain valid: no problem. But the old **T-RQMTS** is no longer valid, having an additional hypothesis. All proofs of $x \in T$ would therefore have to be reviewed.

Fresco avoids this problem by creating new types in each capsule, by qualifying the type names with the capsule names.

7-3.2 Capsules and type conformance

If **KB** imports **KA**, it must do so monotonically: that is, the clients of **KA** must not need rewriting (Fig. 10.) This is essential, since once **KA** is published, they may be difficult to discover.

Within **KA** and its clients, a type name T refers to that type as it is defined within **KA**; if **KB** augments that definition, then T refers to the augmented type, within **KB** and its clients. We shall distinguish them as $\text{KA}::T$ and $\text{KB}::T$, though these qualified names need never be presented at the user interface. If **KB** is defined properly, then

$$\text{T-MONO:} \quad \text{KB} \ll \text{KA} \vdash \text{KB}::T \subseteq \text{KA}::T$$

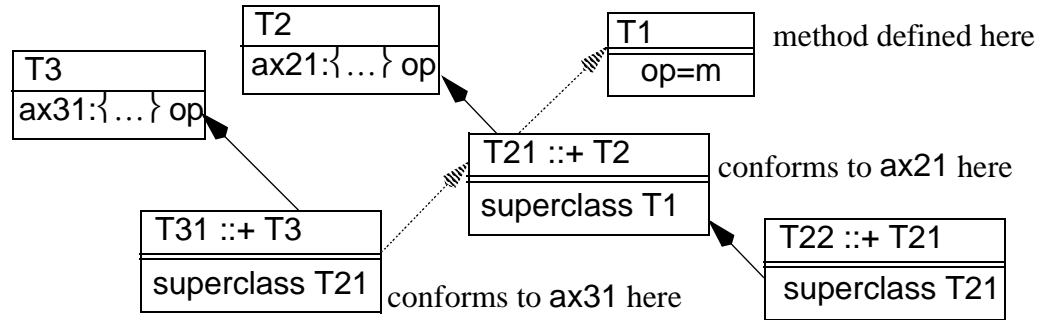
and so the clients need not worry about the distinction. But implementors and refiners of T must be reviewed when **KB** is designed or incorporated, in order to ensure that this theorem is upheld.

7-3.3 Proof expectations

As a system is composed, either by successive incorporation of capsules from a library, or by development of new capsules, Fresco will generate *proof expectations*: theorems which ensure that the system executes according to its specifications. Proof expectations may come ready-proven in an incorporated capsule, or may be generated by Fresco and justified by the designer, during development of a new capsule. Proof expectations include **T-MONO** (above) and the theorem that for every concrete type T (that is, those for which creation functions exist: 7-5 — p.122)

$$\text{T-impl:} \quad x \text{ class} = T \vdash x \in T$$

and hence that the methods attached to or inherited by a type conform to its axioms. For each method which is introduced by a capsule, relevant axioms are stated in its type and its supertypes; and in any type and its supertypes which inherit the method. For example:



An operation definition $op=m$ (defining m to be the method implementing operation op) is said to be *effective* in those classes in which m is activated in response to an invocation of op ; namely:

- the class in which the definition occurs
- every class which does not itself have a definition for op , and which is a subclass of a class in which $op=m$ is effective.

(In C++, the rules are complicated by overloading.)

An axiom AX defined in a type T is *effective* in all the subtypes of T (whether asserted or by reification). An axiom is applicable to an operation op if it is an ops spec for op , or it is an invariant.

To every type which lies within the effectiveness of an axiom $AX[op]$ which is applicable to operation op , and the effectiveness of a definition $op=m$, there is attached a proof expectation $AX[m]$. Many of these expectations will actually be the same, where a subtype is also a subclass: for example, the proof expectation for m is the same in $T22$ (above) as it is in $T21$; Fresco should recognise this and not require duplicated proofs. This reflects the commonplace recommendation (e.g. in [Meyer]) that subclasses should also be subtypes: the requirement for reasoning is much reduced.

When a capsule is incorporated or certified, these proof expectations are searched for and their proofs checked for each new axiom and each new method. The axioms resulting from interpretation of signatures are included in this.

A new supertype can be treated as the introduction of all the axioms known in the supertype.

In a system permitting multiple superclasses, introduction of a new superclass can be treated as the extension of the effectiveness of all the methods effective in the superclass. (If two inherited methods share an operation name without a disambiguating method in the inheriting class, it is not inevitable that there must be a conflict: so long as we stick to the rules, both will be shown to meet all relevant specifications, and it is then acceptable for the execution system to choose one of them arbitrarily.)

A final twist concerns *relevance*. For each type T where there is more than one effective definition of a method for some op , only the definition $K_n::m$ which comes from the most recently incorporated capsule K_n will be executed; and so for T , proof expectations arising from earlier definitions of m are not relevant.

7-3.4 Certification and incorporation checks

Unproven theorems (called *deficits*) in a capsule under development indicate that it is not yet fit to be distributed. A designer may perform a *certification check* on a capsule, which either highlights deficits or marks the capsule as certified; in which case, it may be marked published, protecting it from further alteration.

Informal justifications are allowed, but may be highlighted during the checking process.

The checking is not confined to the assertions and methods visible in one capsule and its imports: for it is important to ensure that conflicts do not arise with other capsules present in the system.

A deficit may also arise upon the incorporation of a capsule into a system, indicating a mutual incompatibility (a *conflict*) with other capsules forming part of that system. For this reason, an *incorporation check* is always performed which duplicates the certification check, but in the context of the incorporating system. It also assures the purchaser that the capsule's certification is not fraudulent! It is much easier and quicker to check proofs than to generate them, so capsules carry all details of their proofs — even though there is a naïve sense in which a proof is no longer required once it has been accomplished.

It should not be necessary for a user composing a system from capsules to perform any further proofs on them. The Fresco scheme ensures that, provided no conflicts are signalled, each capsule will perform as its designer intended, even in the company of unfamiliar fellows, without interference.

Mostly, new implementations and reifications will be supplied together with the specification all in one capsule; but the rules allow them to be supplied separately. Clients need only import the capsules which contain specifications for the aspects of behaviour they are interested in.

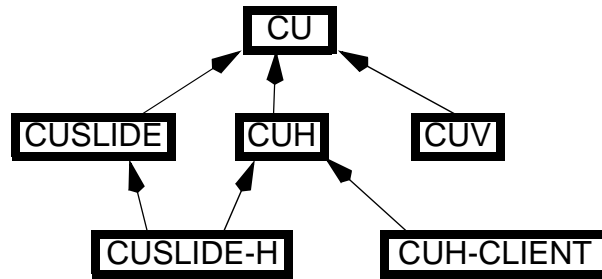
Checks ignore *irrelevant conflicts*: that is, those which apply to methods which have been superseded.

7-3.5 Conflict and resolution

Suppose capsule CU specifies and implements a new operation cui for a user interface, which re-arranges scattered icons into neat rows and columns. Later, an improvement CUH is published which imports CU, tightens the spec and re-implements cui, guaranteeing that the icons will be arranged into alphabetic order by label, reading across rows. Meanwhile, an oriental designer creates an improvement CUV, similar but with vertical ordering. Clearly both cannot be incorporated into our system with a properly working result: on incorporating the second (say CUV), Fresco declares a conflict, because CUV::cui comes without a proof that it conforms to the opspec for cui in CUH.

At this point, Fresco will offer the option of undoing the incorporation of one or other of the conflicting capsules.

Now suppose a further improvement CUSLIDE is designed which imports CU, and makes the rearrangements happen visibly. The specification is compatible with either of CUH and CUV; but nevertheless, we cannot just incorporate CUSLIDE and CUH (say) together, because each of their implementations only takes into account its own spec: Fresco would signal a conflict. But an additional 'fixup'



capsule CUSLIDE-H could be designed which imports both capsules — for their specifications — but re-implements the method cui. This would come with a proof that it conforms to both specs; the conflict is thereby resolved, because for methods, only the most recently incorporated method is relevant (Fig. 10.)

(Notice that:

- It would not be so useful to design a CUSLIDE-H which imported CU directly, since client capsules of CUH (say) would not be satisfied by it.
- It is not appropriate in this situation to design a subtype for each feature.)

It is possible to write a capsule which contains only a partial, or no implementation and acts just as specification.

7-3.6 Renaming

In some cases, a conflict occurs only because two designers have used the same name accidentally for unrelated purposes. When capsules with a naming conflict are incorporated into the same system, the problem can be resolved by systematically changing one of the names.

The situation is easy to identify: it only occurs between capsules which are separately defining a new name. In all cases where a capsule is refining a name, there is a proof expectation that the new item is a refinement of the inherited version; if this proof expectation is not present in a certified capsule, then the definition must be intended to be new.

There is no internal conflict here amongst type names, because they are qualified with the capsule names; but the ambiguity does arise at the user interface, where the qualifications do not appear. And qualification is not done with the *Execution_View*, because the objective is to modify existing code. Hence the necessity to change a name, perhaps by ‘fixing’ the qualification so that (just in this particular system) the capsule name is a permanent part of it.

Instead of incorporating capsule *K*, we are then incorporating *K[x\y1, y\y1]* (where *x, y* are the offending names). Any subsequent capsule which imports *K* will need to have its references to the renamed items doctored; and so the list of renamings should be considered part of the full capsule name, both in the system’s capsule list, and in any list of imports in which *K* appears. Whenever any new dependents of *K* are subsequently published, the reverse mapping is applied before distribution.

7-3.6.1 Hiding

It is desirable to be able to hide names defined within a capsule, both to preclude spurious clashes, and so as to present a limited selection of them to client capsules. This can be done by qualifying the appropriate names with the capsule name, but

presenting only the unqualified name at the user interface. This prevents the item being referred to except from within its own capsule. Hiding can be seen as a special case of renaming.

7-4 Global variables

GlobalDefn
name: VarName type: TypeExprn

A global definition in a capsule declares a variable which may act as the root of a data structure, together with a type and code for its initialisation. It may be an existing or a new variable.

Notice that there are no global invariants — that is, no axioms outside type-definitions and the Fresco fundamental axioms described here.

7-4.1 Initialisation and persistence

Once a capsule is successfully incorporated, the user's permission is sought to invoke the initialisation code of the globals. In some cases, this will be a question of setting up empty new data; in others, of transforming an existing body of data, be it a single menu or a whole database. It is important that when changes are made to a system's software, provision should also be made to bring forward existing data; or at least, that the system should be able to deal with old data.

The initialisation code of global variables is intended for these purposes. Once a capsule has been installed satisfactorily, its initialisations should be performed before the new code is used. For a global g declared of type T , there is a proof expectation of correct initialisation:

$$\text{g-init-type:} \quad \{? :- g \in T\}$$

The precondition is rather ill-defined, since it can depend on just about anything true before the incorporation. It would be unwise to depend upon globals being initialised in any particular order. This is a topic for further investigation.

Assignments to globals are not permitted: only operations may be used to interrogate and change them. This rule ensures that the global continues to have the expected type.

7-4.2 Conformance

An updated global must belong to the same types as it did before, so that old clients can use it. For any global G declared in capsules KA and KB , a proof expectation arises of the form:

$$KB \ll KA, KA::G \in T_1, KB::G \in T_2 \vdash T_2 \subseteq T_1$$

The same issues of conflict, possible resolution and possible renaming arise as for types.

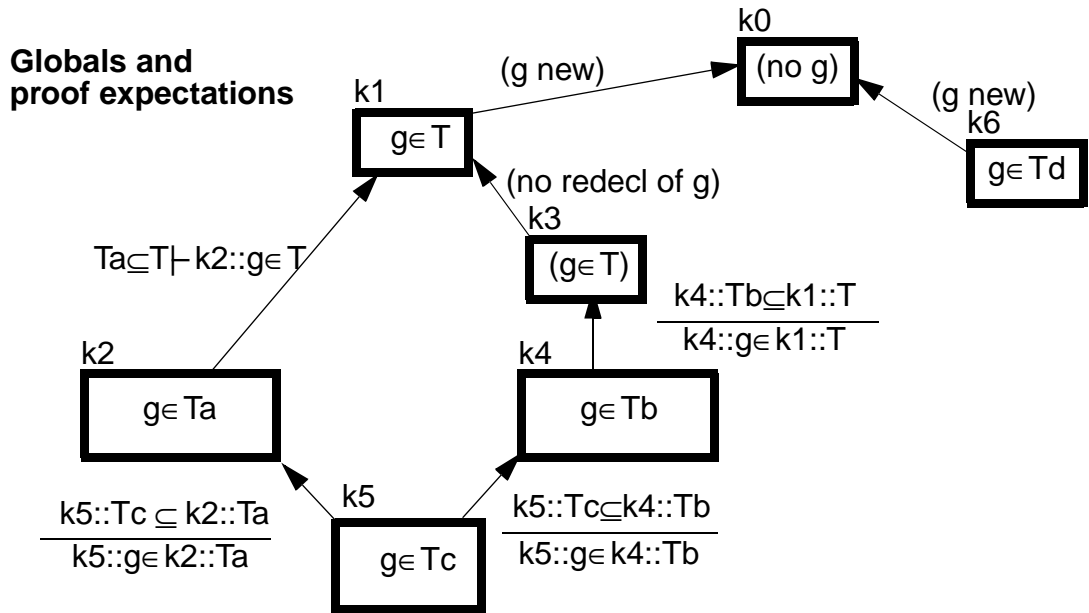


Fig. 11. Globals and proof expectations

7-4.3 Incorporation scenario with globals

The nodes on the diagram (Fig. 11.) represent capsules, and the arcs imports. The arcs are labelled with some of the theorems *provided* by the importing capsules (at the source end of the arrow). Let us suppose that the capsules are all incorporated into one system, in order of their names $k1$ – $k5$.

$k0$ has no global called g . $k1$ imports $k0$ and defines g ; no conformance proof is provided or expected. (There will be a proof that the initialisation code yields a member of T , and a similar proof in each redefinition.)

$k2$ redefines g to conform to Ta , and because $k2 \ll k1$, a proof of $Ta \subseteq T$ is expected. The originating system would have insisted that the designer should provide this before certifying the capsule. $k1$'s software continues to work with g , and even with $k1$ interacting with g , $k2$ works too. The only problem would be if g were to be assigned a value by $k1$ which is a member of T , but not of Ta : for this reason, we do not permit assignment to globals.

$k3$ does not redefine g , but possibly uses it. No conformance proof is expected. In the system in which $k3$ originated, there would have been no $k2$, and so $k3$'s code expects g to be T . Now in the system illustrated, we know that $g \in Ta$: that's OK for $k3$ because $Ta \subseteq T$.

$k4$ redefines $g \in Tb$. In the system in which it was originated, $k2$ was not known, and so $k4$ provides only for the proof expectation that the originating Fresco expected, for $Tb \subseteq T$. However, as far as the incorporating system is concerned, g 's type is Ta , and a proof of $Tb \subseteq Ta$ is what is actually required; the alarm is therefore raised, and the user given the option of undoing the incorporation.

However, this problem has arisen before, and someone has taken the trouble to write $k5$ which imports both $k2$ and $k4$, redefines $g \in Tc$, and provides both the expected proofs. Perhaps it might not be necessary to redefine g , if it is possible to show $Tb \subseteq Ta$: in which case $k5$ contains a proof and nothing else.

Finally, **k6** also happens to define a global called **g**, having been originated in a system in which **k1** was not known. But in our system, we now have $g \in T_C$, so our **Fresco** expects a proof of $T_d \subseteq T_C$. However, since **k6** provides no conformance proof for **g** at all, it clearly has no dependence on any imported definition, and can safely be renamed.

7-5 Creation functions and concreteness

An implemented creation function, for example $T \text{ mk}$ must satisfy the axiom

$$T \text{ mk} \in T$$

so that the proof expectation (if T or mk are [re]defined in capsule K) is:

$$T \text{ mk} \in K::T$$

(where mk stands for any function with or without parameters). Notice that the left-hand T is not qualified with the capsule name: this is because it will appear in actual code, rather than just in proofs, and we don't qualify names in code (because the idea of capsules is to *modify* the code sensibly).

At this point, it becomes useful to distinguish carefully between class and type. (Generally, we can rely on $x \text{ class} = T \vdash x \in T$, and so need not worry about the difference.) We will refer to a class C and a type CT which we hope C will implement. Now we know that

$$T\text{-RQMTS:} \quad x \in T \Leftrightarrow (\bigwedge_i T::AX_i[x])$$

so having created a C -instance, we must prove that all the axioms of CT are observed by it.

This can clearly be done only if there are actually methods supplied in C for all the axioms of CT .

There is an axiom of a primitive function (in **Smalltalk**)

$$\text{basicNew:} \quad C \cdot C \text{ basicNew class} = C$$

The typical creation function will be in this form:

$$\begin{aligned} C \text{ mk} = \{ & \uparrow \in C \wedge Q \} \text{ justification missing here} \\ & \{ \uparrow \text{ INV}[\uparrow] \wedge \uparrow \text{ class} = C \wedge Q \} \\ & \{ x \cdot \{ \uparrow \text{ class} = C \} x \leftarrow C \text{ basicNew.} \\ & \quad \{ x \text{ class} = C \text{ :- INV}[x] \wedge x \text{ class} = C \wedge Q \} x \text{ init.} \\ & \uparrow x \} \end{aligned}$$

where INV is the conjunction of all the invariants of CT , and Q is some particular requirement on the initial state. But this gives rise to the proof expectation

$$\text{INV}[x_0] \wedge x \text{ class} = C \vdash x \in CT$$

or, from $T\text{-RQMTS}$,

$$(\text{INV}[x_0] \wedge x \text{ class} = C \vdash \bigwedge_i CT::AX_i[x]) \vdash x \in CT$$

Clearly this can only be done if we can find a method in C (inherited or not) for each opspec amongst AX_i . Hence a class must be completely implemented — *concrete* — in order for a creation function to be verified and used — which, of course, bears out sensible practice.

The proof depends on the induction over the history of x ; but the designer does not have to elaborate all this explicitly in Fresco: rather, the constraints Fresco imposes ensure its validity. These are:

- The method implementation proof expectations discussed in 7-3 — p.116.
- There is a certification check (an existing compiler constraint in C++) that ensures that creation functions are only invoked on concrete classes.
- Fresco generates the proof expectation that the type invariants are met by the yield of a creation function; for any creation function mk ,

T-INV-CREATION: K::T::INV[T mk]

From this it can be seen that creation functions have a special status in Fresco, just as they have in C++, rather than just being ordinary operations of the metaclass as in bare Smalltalk. Fresco therefore does not support any metaclass operations other than creation functions.

The operation `init` is unusual in not taking the invariant of C as an implicit part of its precondition. The present model of Fresco makes no provision for a special category of initialisation functions — there may be any number per class — so it is up to the designer not to assume the invariant in these cases. Initialisations are typically in this form:

$$C::init = \{ \text{self} \in C \} \{ INV[e1, e2] :- INV[v1, v2] \} \\ \{ \begin{array}{l} v1 \leftarrow e1. \\ v2 \leftarrow e2. \end{array} \}$$

Creation functions themselves have no reference to ‘self’ (unless it is the object representing the class itself in Smalltalk idiom — not a recommended practice).

7-6 The User Interface

A prototype for a Smalltalk-based capsules system without theorem-proving facilities exists. This outline of the projected development environment is therefore somewhat less hypothetical than much of what has gone before.

All development work is done within some capsule; each is represented by a large window within which can be created browsing tools for types, methods, and the details of the capsule.

In the present prototype, the tools look similar to the standard Smalltalk browsers; but much fascinating speculation can be made about the possibilities for integrating the tools for formal material with, for example, a hypertext and diagrammatic system for the informal description, similar to the various object-oriented analysis and design tools which are appearing on the market (supporting, for example, [Rumbaugh]). Ideally, one environment should take the designer all the way from formally-annotated OOA through to verified code.

The designer can look at any of the specification or executable code in the system; but only those items most recently updated by this or an imported capsule can be altered. No alterations can be made to a ‘published’ capsule. Alterations apparently made to the definitions in imported capsules are recorded as changes in the current capsule.

The structure of the system as seen through the browsers reflects the class and type hierarchies rather than the capsule structure; though the originating capsule of each type box and method can be discovered, and it is also possible to see previous versions of a method, from each capsule which defined one.

Theorems can appear both within a type box and in browsing tools of their own; any theorem can be asked to display its proof, which pops up in a separate box. A theorem inside some context such as a proof, code block, type, or indeed a capsule, can be dragged into the enclosing context; and Fresco adds the necessary metavariables and hypotheses.

Method code can be viewed with or without the specification components. If with, then highlighting a specification-statement gives access through menus to its justification, which again can be shown separately.

The encapsulation of software into capsules is about managing the dependencies between software modules. Therefore it would be useful for the system to draw a diagram of dependencies, both between type-boxes and between capsules.

7-7 Summary

Every Fresco system is a composition of capsules, beginning with a Kernel capsule. Each capsule contains definitions of types, global variables, and theorems, which may be new or may augment existing definitions. Every capsule must import those others whose definitions it uses.

Capsules may be composed in configurations other than those in which they were originally developed; but Fresco's proof expectation system ensures that a warning is given if two capsules would interfere.

A capsule is concerned with relatively static matters — specification and code rather than data — but may stipulate initialisation and translations upon persistent data. We have not investigated the constraints which a translation would have to satisfy.

The generation of proof expectations and the process of checking before a capsule is distributed, and on incorporation into a system have been described.

8 Objects and verification

The aims of this chapter are twofold:

- to discuss certain topics, skirted hitherto, which are strongly connected with the partitioning of the system state into objects;
- to try the utility of the concepts presented in the preceding chapters — that is, the interpretation of types as theories about object histories — as an aid to dealing with these issues.

The latter should be seen as the primary goal (even though it is essayed as a fairly hefty attack on the former).

The topics are:

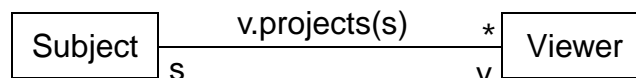
- Constraint maintenance: applying formal methods to the design of programs as frameworks of co-operating classes.
- Framing: determining whether $\{inv\}$ code when the specification of code does not explicitly mention the variables involved in inv .
- Barred expressions: the use and interpretation of variables and expressions denoting prior states in postconditions.
- Equality and subtyping: equality is non-monotonic.

Each section ends with a summary which points out how the Fresco formal notion of types has helped in the discussion. The chapter ends with an overall review.

8-1 Constraint maintenance: co-operating objects

8-1.1 Constrained subsystems

The functional focus in an OO program is often not individual objects, but co-operating clusters of them. [JF88] suggests that types should be documented in ‘frameworks’, or interdependent groups whose instances co-operate. The OO design method of [Rumbaugh91] lays as much emphasis on the attributes and properties of the connections between objects (and therefore the relations between their classes) as on the objects and classes themselves. The typical problem is to maintain an invariant across a link, and in the first stages of design, it is useful to annotate the link with the invariant, deferring the design decision as to how to realise it, or whether to attribute it to a particular class:



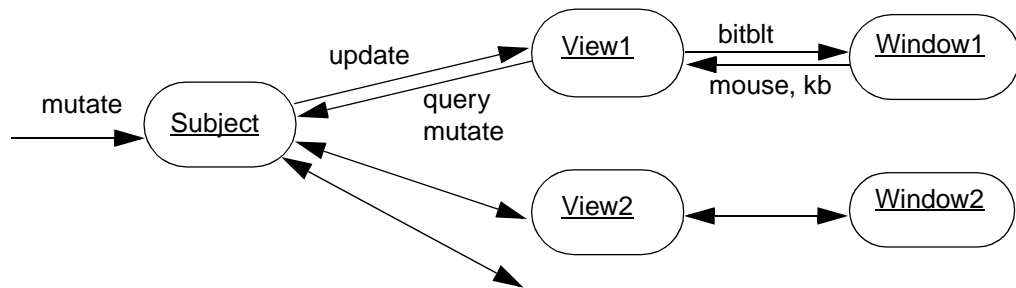
‘*’ signifies there may be many Viewers per Subject. Each Viewer displays a Subject on the screen (or provides a ‘view’ in database terms, to other interactive devices or other parts of the software), and may possibly be used to invoke modifications of it. When a modification occurs, all extant views should simultaneously reflect the change: that is, the `projects` relation should be maintained.

Because we still wish to be able to add new classes (for example, new kinds of view) after the initial design of the framework, it is essential to be able to define the interfaces between them. [MP91] suggests a rule-based programming approach, and [HHG90] demonstrates a ‘white-box’ notation for this purpose, in which skeleton sequences of operation calls are documented. Fresco uses types to achieve a black-box approach.

Suppose that Core is a type of object representing some problem-specific information. There is some stored Data with an invariant, and a typical pair of modification and query operations:

Core	
op mod \in	(Key, Item)
fn get \in	(Key) Item
{ \vdash post-mod(d) } mod	
{ \vdash post-get(\uparrow , d) } get (...)	
var d \in	Data
d.consistent	

Now if the requirement is to display Core-members on the screen, then extensions will be required. This object-diagram illustrates the general scheme:



This is an object diagram showing a typical snapshot of the connections in a system. The arrows show *protocols*, named groups of messages: for example, the mutate protocol just contains mod in this example. There may be any number of View subsystems each displaying part of the current state of the Subject. Each sends queries to the Subject to find its current state, and displays part of it on the screen using appropriate bitmap operations (‘bitblt’). Part of the view’s function is to translate the user’s mouse and keyboard actions into mutating operations on the Subject. Each view is required always to show the current state of the Subject, and so there is an ‘update’ protocol whereby the Subject can notify all the views of any alterations as soon as they happen.

The Subject object must implement the Core type, and in addition must be able to acquire views and have them display the relevant pictures:

VisibleCore ::+ Core
op addView \in (Viewer) op resetViews
var vv \in IdSet(Viewer) fn wf-Subject = $(\forall v \in \text{View} \cdot v \in \text{vv} \Rightarrow v.\text{projects}(\text{self}))$ fn bkwd-ptr = $(\forall v \in \text{View} \cdot v \in \text{vv} \Rightarrow v.s == \text{self})$ bkwd-ptr // invariant $v \cdot \{ \vdash \text{vv} = \overline{\text{vv}} \cup \text{IdSet}(\text{Viewer}).\text{mk}(v) \} \text{ addView}$ $\{ \vdash \text{wf-Subject} \} \text{ resetViews}$

(An IdSet(T) is a set all of whose members are the identities of members of T – see §10-2.1 — p.170.)

This assumes the existence of the Viewer type, which must provide for the update protocol, which in turn must use Core's get function.

Viewer
op update $\{ \vdash \text{self.projects}(s) \} \text{ update}$
var s \in VisibleCore fn projects \in VisibleCore \rightarrow Boolean self \in s.vv

There may be several different kinds of Viewer, with the projects relation specified differently for each one.

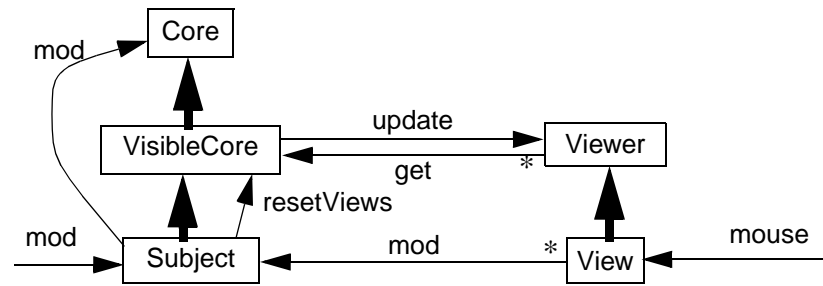
From the point of view of clients external to this subsystem, the Subject must invariably be accurately displayed on the Views, so we have a further type. (For the purpose of illustration, let's conjecture that it has some operation of its own.)

Subject ::+ VisibleCore
op bringForwardAllViews
wf-Subject // from VisibleCore

From the point of view of the display, we need to provide for user input, and again it is convenient to express this as yet another type. The projection constraint should always be true when input operations are dealt with, and the standard way of expressing this is to insist that s must be a Subject:

View ::+ Viewer
op mouseOp \in (MouseStuff) ...
var s \in Subject

Finally, to summarise this complex net of types:



(Thick arrows show subtyping, thin arrows are message paths; ‘*’ denotes the possibility of multiple instances at one end of the connection.)

Each type represents a protocol. An object dealing with several protocols is a member of several types.

8-1.2 Callback and invariants

It was observed in an earlier section that an object either belongs throughout its life to some type, or it doesn’t, since type membership is a predicate over an object’s history. This is certainly true if we look upon each operation as an atomic transition. But whilst `mod` is executing, the projection invariant may be invalid, and so the object is temporarily not a `Subject`, in the practical sense that it would be improper to call `bringForwardAllViews`, whose implementation might rely on the views being up to date.

The type to which an object is temporarily ‘downgraded’ whilst an operation is active will be called a ‘transaction type’.

For completely encapsulated objects, this is not an issue, since it would be impossible to call an operation on an object that is already in the process of executing one. But `VisibleCore` and `Viewer` stipulate a loop of pointers between their members so that a call to `update` can ‘call back’ the originating `Core` object. For this reason, `Viewer::s ∈ VisibleCore`, rather than `Subject`: this allows `update` to get the information it needs to restore the contractual constraint.

As an aside, it is worth noting that such callbacks are prohibited in POOL [America], to avoid this complication. The same strategy could be followed in other languages, setting a flag in an object during the activity of any public method, and rebuffing any attempt to use public methods while it is set. It might be thought that the problem in POOL is connected with its concurrency — that this measure is equivalent to the protection by a semaphore of a critical segment of code. But encapsulation makes the problem almost as bad in a single-process system: any message sent to another object is liable to be implemented by code outside our ken — and so we have no control or knowledge about what else the receiver may send messages to, and whether control may not ultimately loop back to us. The only intrinsic advantage single processing has over concurrent processing, in OOP, is the guarantee that there will be no interruptions between the evaluations of one expression and the next, and in the invocation and return from messages to self.

This is an instance where the distinction between $x \in T$ and $x:T$ is a useful one. (Recall that $x \in T$ means that the invariants are interpreted as permanent assertions about every state, while $x:T$ conjoins the ‘invariants’ with every pre- and postcondition — §6-1.2, p.83.) Of an instance s of some class which implements `Subject`, it is always

true that $s : \text{Subject} \wedge s : \text{VisibleCore} \wedge s : \text{Core}$; but $s \in \text{Subject}$, for example, is true only while no method of Subject is in progress.

Consider an implementation of a method for mod in Subject, shown here with the skeleton of a proof:

```
Subject::mod (k,v) =
  {self ∈ Subject :- self ∈ Subject ∧ post-mod }           //opspec & invariants
  (
    {self ∈ VisibleCore :- self ∈ VisibleCore ∧ post-mod(k,v) }
    super.mod(k, v) ;                                       // call VisibleCore::mod
    {self ∈ VisibleCore :- self ∈ Subject ∧ d=̄d },
    self.resetViews;
  )
```

Strong type-membership has been used here as a shorthand for conformance to the various invariants. The method has been implemented by calling a more primitive version of mod, inherited from a superclass, which does the required business on the data; it seems reasonable to expect that this may affect the correctness of the views, so that $\text{self} \in \text{Subject}$ is no longer true: and so the next thing is to call resetViews. (We need to know that the data d are not affected by this. Once again, discussion of such ‘frame’ inferences is left until a later section.)

The code of VisibleCore::mod is inherited from Core, where it will be shown to conform to $\{ \text{self} \in \text{Core} :- \text{self} \in \text{Core} \wedge \text{post-mod}(k,v) \}$. For VisibleCore, it will therefore additionally have to be shown to conform to the invariant $\{ \text{self} \in \text{VisibleCore} \}$ or $\{ \text{bkwd-ptr} \}$.

VisibleCore::resetViews is implemented as a call to update in each view:

```
VisibleCore::resetViews =
  { self ∈ VisibleCore :- self ∈ VisibleCore ∧ wf-Subject }
  (
    vv do: [:v |
      {self ∈ VisibleCore :- v.projects(self) ∧ self ∈ VisibleCore ∧
        ∧ vv=̄vv ∧ ∀ vx · vx ∈ vv ∧ vx|v ⇒ vx=̄vx }
      v.update] )
```

8-1.2.1 Propagation of transaction status

Whilst the truth of $\text{self} : T$ is fixed for any object, $\text{self} \in T$ may be assumed (in methods attached to T or its subtypes) only on entry to a method, and should be restored by the end. Thus Subject::mod could not, half way through, call bringForwardAllViews, while the implicit precondition $\text{self} \in \text{Subject}$ is false. Nor should mouse messages be sent to any of the associated views, since $v \in \text{View}$ depends on $v.s \in \text{Subject}$.

8-1.3 Constraints & contracts summary

Constraints operating between classes can be implemented as frameworks of classes co-operating according to a set of contracts. Using the weaker Fresco definition of type membership, the contracts have been characterised using transaction types. This contrasts with the “white box” approach to contracts, in which specific sequences of operations are stipulated.

8-2 Aliasing example

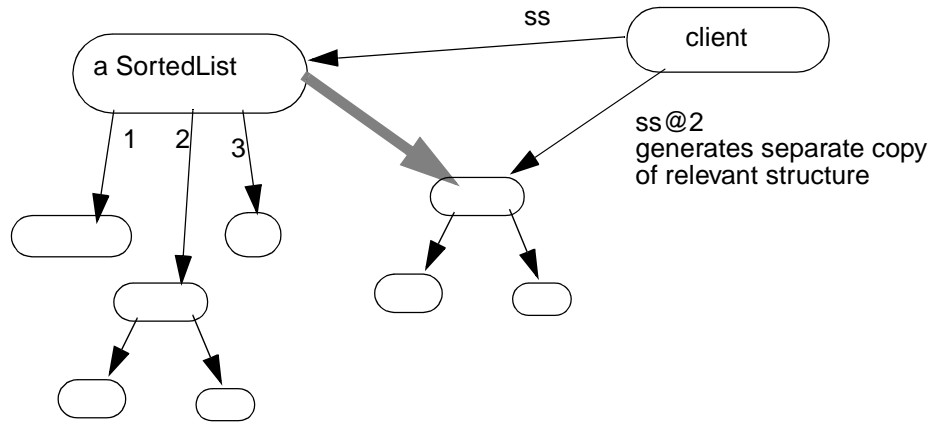
This section considers the problems which arise from the possibility of multiple access paths to an object.

8-2.1 SortedList and SortedIdList

SortedList(T) is a generic type of objects which accept members of type T and give them back in sorted order; T has to have a relation \leq which is a total ordering.

SortedList(T)
op add \in (T) fn $_@_ \in$ (Nat) $\rightarrow T$ $x \cdot \{x \in T :- x = \bar{x} \wedge \text{sl.elems} = \bar{\text{sl}}.\text{elems} \cup \text{IdSet.mk}(x)\} \text{ add}(x)$ $\{i \in \text{Nat} \wedge i \in 1..\text{sl.length} :- \uparrow = \text{sl}@i\} \text{ self}@i$
var sl \in IdList(T) $\forall i, j \in 1..\text{sl.length} \cdot i < j \Rightarrow \text{sl}@i < \text{sl}@j$

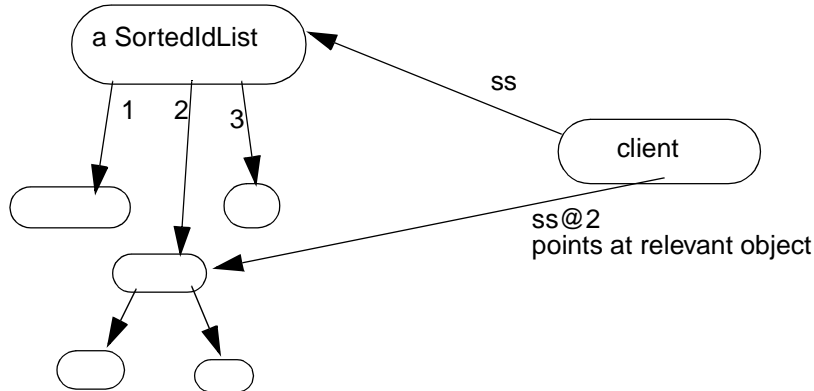
SortedList guarantees only to give back objects which are equal to the inputs, but



there is a variant SortedIdList(T) which gives back the identical objects that were presented to it.

SortedIdList(T)
op add \in (T) fn $_@_ \in$ (Nat) $\rightarrow T$ $x \cdot \{x \in T :- x = \bar{x} \wedge \text{sl.elems} = \bar{\text{sl}}.\text{elems} \cup \text{IdSet.mk}(x)\} \text{ add}(x)$ $\{i \in \text{Nat} \wedge i \in 1..\text{sl.length} :- \uparrow == \text{sl}@i\} \text{ self}@i$
var sl \in IdList(T)

(`IdList` guarantees to preserve the identity of the objects, rather than just their states. I have chosen to use `IdList` in both models, to minimise differences between them; the difference is in the use of `==` or `=` in the postconditions.)



8-2.2 Method implementations

Now consider a pair of simple implementations having the same structure — that is, each object has one component which is an `IdList`. The `add` methods are:

```

SortedIdList(T) :: add(x) =
  { x ∈ T ∧ self ∈ SortedIdList(T)                                //i.e. conforms to invariants
    :- x =  $\bar{x}$  ∧ ∃ before, new, after ∈ IdList(T) ·
      new.length = 1 ∧ before ++ after =  $\bar{sl}$  ∧
      before ++ new ++ after = sl ∧ new@1 == x
    ∧ self ∈ SortedIdList(T)
  }
  (
    var n;
    { :- (n > 1 ⇒ sl@n-1 ≤ x) ∧ (n < sl.length ⇒ x ≤ sl@n) }
    n := findPlace(sl, x);                                           // local fn
    { :- ∃ before, new, after ∈ IdList(T) · before.length = n ∧
      new.length = 1 ∧ before ++ after =  $\bar{sl}$  ∧
      before ++ new ++ after = sl ∧ new@1 == x }
    sl.insert(x, n);
  )

```

and — with even more of the skeleton proof omitted —

```

SortedList :: add(x) =
  (
    var n;
    n := findPlace(sl, x);
    { :- ∃ before, new, after ∈ IdList(T) · before.length = n ∧
      new.length = 1 ∧ before ++ after =  $\bar{sl}$  ∧
      before ++ new ++ after = sl ∧ new@1 == x }
    sl.insert(x.copy, n);
  )

```

The access methods are

```

SortedIdList(T) :: @( $\bar{x}$ ) = ( $\uparrow$ sl@i);
SortedList(T) :: @( $\bar{x}$ ) = ( $\uparrow$ (sl@i).copy);

```

8-2.3 Using the lists

The crucial difference between the two classes is that while `SortedIdList` accepts and yields the identity of the items to be sorted, `SortedList` always makes copies, both when accepting a new item, and when providing access to the list. `SortedIdList` is

open to interference with its invariant: having added mutable items to a SortedIdList, a client could alter individual items, causing the invariant to become invalid:

```

sidl := SortedIdList(T).empty;           // make new list
sidl.add(item1);
sidl.add(item2);
item1.addOn(50000);                       // mutate list contents
smaller := sidel@1;                       //not necessarily!

```

The problem does not arise with SortedList, because item1 is not the object stored in the list. Clearly, clients of SortedIdList have some extra proof obligation to fulfill, if they are to use it properly. Let us look at that code fragment augmented with the principal code-specs of an attempted proof:

```

{ item1 ∈ T ∧ item2 ∈ T :- ??? } (
  { item1 ∈ T ∧ item2 ∈ T :-
    :- item1 ∈ T ∧ item2 ∈ T ∧ sidel ∈ SortedIdList(T) ∧ sidel.length = 0 }
    sidel := SortedIdList(T).empty;
  { sidel ∈ SortedIdList(T) ∧ item1 ∈ T ∧ item2 ∈ T
    :- item1 ∈ T ∧ item2 ∈ T ∧ sidel ∈ SortedIdList ∧ sidel.elems = IdSet.mk(item1) }
    { sidel ∈ SortedIdList(T) ∧ item1 ∈ T
    :- sidel ∈ SortedIdList(T) ∧ item1 = item1
      ∧ sidel.sl.elems = sidel.sl.elems ∪ IdSet.mk(item1) }
    sidel.add(item1);
  { sidel ∈ SortedIdList ∧ sidel.elems = IdSet.mk(item1) ∧ item2 ∈ T ∧
    :- sidel ∈ SortedIdList ∧ sidel.elems = IdSet.mk(item1, item2) } //similarly
    sidel.add(item2);
  { sidel ∈ SortedIdList ∧ sidel.elems = IdSet.mk(item1, item2) item1 ∈ T
    :- item1 ∈ T ∧ item1 = item1 + 50000 }
    item1.addOn(50000);
  ...)

```

The proof really comes adrift at this point because $\text{sidel} \in \text{SortedIdList}$ is not preserved by $\text{item1.addOn}(50000)$. But how do we know that? Now consider a similar use of SortedList:

```

{ item1 ∈ T ∧ item2 ∈ T
:- (smallest = item1 ∨ smallest = item2) ∧ smallest ≤ item1 ∧ smallest ≤ item2 } (
  sidel := SortedList(T).empty;           // make new list
  sidel.add(item1);                       //put a copy of each item in the list
  sidel.add(item2);
  { sidel ∈ SortedList }                  // how do we know???
  item1.addOn(50000);                     // mutate original – list unaffected
  { sidel ∈ SortedList :- ... } smaller := sidel@1; // guaranteed smaller of originals
)

```

The key thing is that $\text{sidel} \in \text{SortedList}$ is not disturbed in this case by changes to item1. The same immunity can be got even with SortedIdList, if the client does the copying:

```

{item1 ∈ T ∧ item2 ∈ T
:- (smallest = item1 ∨ smallest = item2) ∧ smallest ≤ item1 ∧ smallest ≤ item2 } (
    sidl := SortedIdList(T).empty;                                // make new list
    sidl.add(item1.copy);                                           // put a copy of item1 in the list
    sidl.add(item2.copy);
    {sidl ∈ SortedIdList}                                          // how do we know???
    item1.addOn(50000);                                             // mutate original
    {sidl ∈ SortedIdList :- ... } smallest := sidl@1;              // guaranteed smallest
)

```

Or conversely, how do we know that, for example, $\text{item1} \in T$ is preserved by the statements in whose invariants it is not directly implicated? Consider the above fragment with the code-specs pared down to focus interest on item1 :

```

{item1 ∈ T :- item1 ∈ T ∧ item1 = item1 + 50000 } (
    {item1 ∈ T} sidl := SortedIdList(T).empty;                    // how do we know???
    {item1 ∈ T} sidl.add(item1);                                    // spec of SortedIdList::add
    {item1 ∈ T} sidl.add(item2);                                    // how do we know ???
    {item1 ∈ T :- item1 ∈ T ∧ item1 = item1 + 50000} item1.addOn(50000); // T::addOn
    {item1 ∈ T ∧ item1 = item10 + 50000} smallest := sidl@1;      // how do we know???
)

```

In some cases, the relevant opspec is explicit about the effects on item1 ; in other cases, the independence (or otherwise) of invariant and code is entirely implicit.

8-2.4 Aliasing problem summary

The general problem of aliasing is to prove $\{inv\}$ code in those cases where the explicit specifications of the operations involved in code don't mention inv . The problem of type-invariant-breaking also falls within this scope if the weak definition of typing is used (in which $x : T$ is static and $x \in T$ varies). In OOP, the particular obstacle is that we wish to specify and design one type without knowing about the other subsystems which may interfere with it (as distinct from designing a whole program at once, and having invariants that apply to the whole thing).

The analysis of the problem in terms of invariants on types has, for the author at least, made the nature of the problem clearer than an intuitive feeling that aliasing is difficult.

8-3 Effects calculus

This section outlines a method of deciding whether a piece of code will leave an assertion invariant. It is similar to a system described in [Johnson 91], which was developed simultaneously.

Each expression or statement has a *writing frame*, which is the set of variables to which it may write. Every expression or statement has a *reading frame*, the set of variables whose values affect its behaviour and outcome. If the reading frame of some pure predicate M is disjoint from the writing frame of S , then $\{M\} S$ (i.e. M is an invariant of S).

The complication is that since there may be aliasing, other variables may be affected, besides those immediately apparent. A variable is said to be *separate* from the frame of a statement S , if v is unaffected by the execution of S . A variable x is

said to be separate from another y , written $x \text{ sep } y$, if for every operation S whose frame includes y but not x , x is unaffected by S .

To determine whether $\{M\}S$, it is therefore necessary to determine:

- the reading frame ρ of M ;
- the writing frame ω of S ;
- whether $\rho \text{ sep } \omega$

A relevant set of rules follows.

8-3.1 Definition of concepts

8-3.1.1 Fields

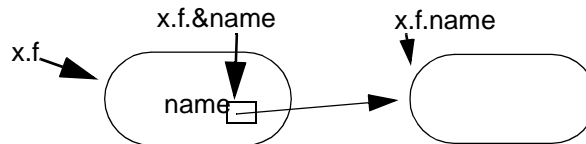
A *field* is a name to which an assignment of an object identity may be made within some context. This includes:

- local variables, within operations and statement-sequences
- global variables
- the instance-variables of objects (marked *var* in TCDs)

A field may be referred to

- by name for local and global variables: $\&\text{name}$
- by containing object and name for instance variables; the object is identified by some expression yielding its identity: $x.f.\&\text{name}$
(In a TCD, $\&\text{name}$ is an abbreviation for $\text{self}.\&\text{name}$)

All the fields of an object may be referred to as a set by omitting a specific field-name: $x.f$, self , etc. Notice that $x.f.\&\text{name}$ refers to a specific field within the object identified by $x.f$, whilst $x.f.\text{name}$ refers to the whole of the object identified by that field.



There may be more than one expression which refers to a single object.

8-3.1.2 Frames

Every expression or statement E has a reading frame and a writing frame. The writing-frame is the set of fields which could possibly be altered by executing E ; the reading-frame is the set of fields, altering which could possibly affect the outcome of executing E . We define relations ∇ and Δ :

$\nabla \text{ dr} \cdot E$ states that the reading frame of $E \subseteq \text{dr}$

$\Delta \text{ dw} \cdot E$ states that the writing frame of $E \subseteq \text{dw}$

Notice that the domain of these relations is the syntactic structure of pieces of code, not the result-values they return: an expression appearing to the right of the “.” cannot be substituted by another which has an equal evaluation.

If the reading and writing frames of an expression E are disjoint,

$$\exists \text{ dw}, \text{ dr} \cdot E \Delta \text{ dw} \wedge E \nabla \text{ dr} \wedge \text{dw} \cap \text{dr} = \emptyset$$

then e is said to be *pure*, which means it can be repeatedly evaluated without any change of outcome (which doesn't necessarily mean it won't affect the outcome of any other operations).

If the reading frame of an expression e and the writing frame of a statement S are disjoint, S will not have any effect on e . If e is a pure predicate,

$$\text{fx-indep:} \quad \rho, \omega, e, S \cdot \frac{\begin{array}{c} \Delta \omega \cdot S \\ \nabla \rho \cdot e \\ \rho \cap \omega = \emptyset \end{array}}{\{ e \} S}$$

More generally, if the reading frame of any expression or statement $S2$ is disjoint from the writing frame of $S1$, preceding $S2$ with $S1$ will make no difference to any effects attributable to $S2$:

$$\text{fx-indep-stmt:} \quad \rho, \omega, S1, S2 \cdot \Delta \omega \cdot S1, \nabla \rho \cdot S2, \rho \text{ sep } \omega \vdash \{ P:-R \} (S1; \{ P:-R \} S2)$$

(where $\rho \text{ sep } \omega \equiv \rho \cap \omega = \emptyset$).

A frame is an outer bound on the arena of action of an operation. If there are two frame-specifications for one operation, then they must intersect. (Many of the following rules are the same for both Δ and ∇ , and are given with \diamond standing for either.)

$$\text{fx-conjoin:} \quad d1, d2, S \cdot \frac{\begin{array}{c} \diamond d1 \cdot S \\ \diamond d2 \cdot S \end{array}}{\diamond d1 \cap d2 \cdot S}$$

If we know S deals with a frame sd which is wholly contained in a frame d , then we may also state that S deals with the whole d .

$$\text{fx-expand :} \quad sd, d, S \cdot \frac{\begin{array}{c} sd \subseteq d \\ \diamond sd \cdot S \end{array}}{\diamond d \cdot S}$$

8-3.1.3 Demesnes¹

The details of the frames of a statement will usually depend on the encapsulated detail of the definitions of the objects being dealt with. For example, if $dd \in \text{SortedList(Shape)}$, then the writing-frame of $(dd@3).\text{move}(v)$ depends on what kind of $\text{Shape } dd@3$ is. We therefore define the *demesne* of an object x , written $x.\delta$, to be that set of fields which are involved in representing its current state. Each class has its own definition of δ , and frames can be defined without knowing the details of the frames. For example:

$$\Delta (dd@3).\delta \cdot ((dd@3).\text{move}(v))$$

-
1. A demesne in feudal times was the land owned by the lord of the manor, held in feu by the villeins of his village. Except that of course his land was in turn really owned by the local baron; and so on through dukes and princes etc. up to the King. 'Demesne' has common derivation with 'domain', and is pronounced more or less the same.

for which sample demesne functions might be:

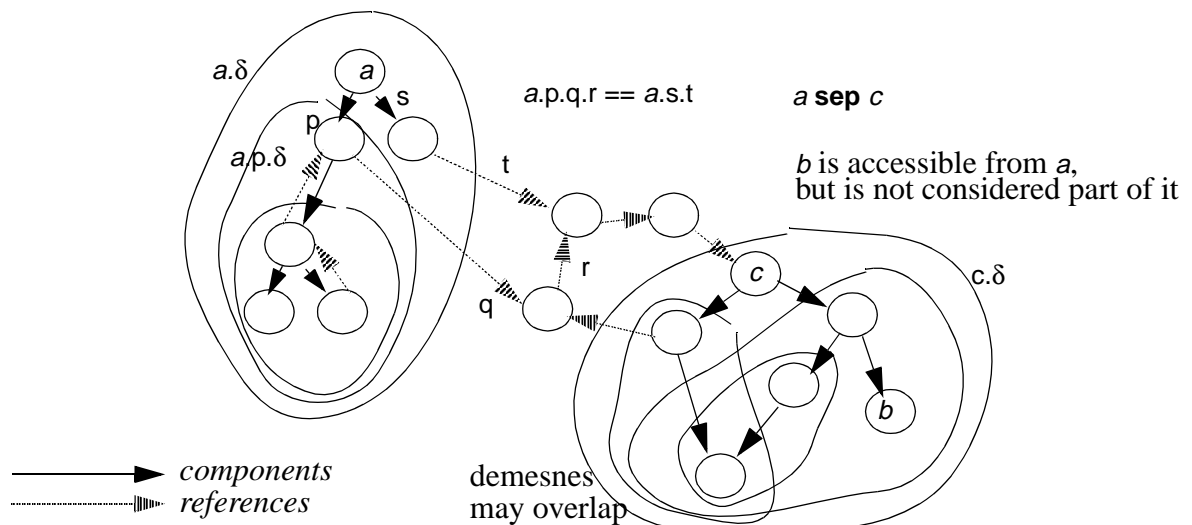
Triangle	MultiLine
...	...
var v1, v2, v3 ∈ Point fn δ = {self, v1.δ, v2.δ, v3.δ}	var path ∈ List(Point) fn δ = {&path, pathδ}

Demesnes are written in conventional set notation, and for convenience are always interpreted after flattening any nested set structure: only the unified set of fields is important.

The designer must define a demesne function for every TCD. It should have no parameters, and should include all fields which are taken into account in an equality comparison. Each field can be categorised into the following groups, according to the role of the object to which it points:

- *Component* — the state of the object to which it points is regarded as part of the state of self: e.g. each vertex of Triangle; or the salary field of an Employee; or the boundaries and subwindows of a window. Both the field and the demesne of the component object should be included in the demesne of self.
- *Reference* — it is the identity of the pointed-to object which is important, rather than its current state. For example, the boss field in Employee should point to another Employee; reassigning self.boss would usefully be regarded as a significant change to the state of self, while a change in self.boss.salary would not be regarded as a change to self. The model displayed in a window is a reference, since not all changes in the model's state would be regarded as a change to the window.
A reference field should be included in the demesne (&boss), but not the demesne of the referred-to object.
- *Redundant* — caches and “upward” pointers fall into this category: they have no functionally visible effect or are always completely determined by other parts of the state.

Fig. 12. Demesnes define boundaries around subsystems of objects



8-3.1.4 Frames in signatures

SortedList(T)		
op add ∈	(x:T)	$\Delta \{sl.\delta\} \quad \nabla \{x.\delta, sl.\delta, \{sli.\delta \mid sli \cdot sli \in sl.elems\}\}$
fn _@_ ∈	(Nat) → T	$\Delta \emptyset \quad \nabla \{sl.\delta, \{sli.\delta \mid sli \cdot sli \in sl.elems\}\}$
...		
var sl ∈	IdList(T)	
...		

8-3.1.5 Separation

$$\rho \text{ sep } \omega \equiv \rho \cap \omega = \emptyset$$
$$x \text{ sep } y \equiv x.\delta \cap y.\delta = \emptyset$$
$$\text{sep-comm : } z1, z2 . \frac{z1 \text{ sep } z2}{z2 \text{ sep } z1}$$
$$\text{sep-preserve: } z1, z2, S, \rho, \omega \cdot \frac{\begin{array}{c} \forall \rho \cdot S \\ \Delta \omega \cdot S \\ P \vdash \omega \text{ **sep** } z1 \vee \rho \text{ **sep** } z2 \\ P \vdash \rho \text{ **sep** } z1 \vee \omega \text{ **sep** } z2 \end{array}}{\{ P \wedge z1 \text{ **sep** } z2 :- z1 \text{ **sep** } z2 \} S}$$

The property of *isolation* means that an item is separate from everything else accessible to any client:

$$\text{isol-defn:} \quad z.\text{isolated} \equiv \forall x \cdot \neg(\uparrow == x) \Rightarrow z \text{ **sep** } x$$

All uninitialised newly created items are isolated:

$$\text{create-sep :} \quad \{:- \uparrow.\text{isolated}\} \text{ AClass.basicNew}$$

where AClass is any class. This is a property of the built-in creation function **basicNew**, but the designer-built creation functions for any particular class may create non-isolated items, ready-linked into an existing structure.

If a statement has an empty reading and writing demesne, then it can only yield an isolated item:

$$\text{indep-sep :} \quad \Delta \emptyset \nabla \emptyset \cdot S \vdash \{:- \uparrow.\text{isolated}\} S$$

8-3.2 Aliasing example revisited

8-3.2.1 Type specification with framing

The opspecs have been augmented to provide the relevant guarantees, and some of the theorems are now labelled:

SortedList(T)			
op add ∈	(x:T)	Δ {sl}	∇{x.δ, sl.δ, {sli.δ sli · sli ∈ sl.elems}}
fn @_ ∈	(Nat) → T	Δ ∅	∇{sl.δ, {sli.δ sli · sli ∈ sl.elems}}
add-1: x · {x ∈ T :- x = x̄			∧ (∀ sli · sli ∈ sl.elems ⇔ (sli ∈ s̄l.elems ∨
sli = x ∧ sli.isolated))			∧ (∀ sli · sli ∈ sl.elems ⇒ x sep sli)
{ add(x)			
add-2: x · { x sep self } add(x)			
{ i ∈ Nat ∧ i ∈ 1..sl.length :- ↑ = sl@i ∧ ∀ sli · sli ∈ sl.elems ⇒ ↑ sep sli } self@i			
var sl ∈			
IdList(T)			
∀ i, j ∈ 1..sl.length · i < j ⇒ sl@i ≤ sl@j			

and the creation function:

$$\Delta \emptyset \nabla \emptyset \cdot \{ \uparrow \in \text{SortedList}(T) \wedge \uparrow.\text{length} = 0 \} \text{SortedList}(T).\text{empty}$$

8-3.2.2 Sketch proof

An outline proof of the example using SortedList (which keeps copies of its arguments)

```

{item1 ∈ T ∧ item2 ∈ T
:- (smaller =  $\overline{\text{item1}}$  ∨ smaller =  $\overline{\text{item2}}$ ) ∧ smaller ≤  $\overline{\text{item1}}$  ∧ smaller ≤  $\overline{\text{item2}}$  } (
    sidl := SortedList(T).empty; // make new list
    sidl.add(item1); // put a copy of each item in the list
    sidl.add(item2);
    {sidl ∈ SortedList ∧ item1 sep sidl} // crucial invariant
    item1.addOn(50000); // mutate original – list unaffected
    {sidl ∈ SortedList} // crucial precondition
    :- (smaller =  $\overline{\text{item1}}$  ∨ smaller =  $\overline{\text{item2}}$ ) ∧ smaller ≤  $\overline{\text{item1}}$  ∧ smaller ≤  $\overline{\text{item2}}$ 
    {smaller := sidl@1; // guaranteed smaller of originals
)

```

The last line can be shown easily from the postcondition of @ and the invariant of SortedList, provided the invariant has not been interfered with — as would happen if SortedIdList had been used. It is essential for the preservation of $\text{sidl} \in \text{SortedList}$, that $\text{item1} \mathbf{sep} \text{sidl}$ when $\text{item1.addOn}(50000)$ executes. To see that this is the case on entry to that operation, consider another sketch highlighting the continued separation of the various protagonists:

```

{item1 ∈ T ∧ item1 sep item2 :- ... } (
    fn all = item1 ∈ T ∧ item1 sep item2 ∧ item1 sep sidl ∧ item2 sep sidl
        ∧ ∀ sli · sli ∈ sidl.sl.elems ⇒ item1 sep sli; // convenient auxiliary
    {item1 ∈ T ∧ item1 sep item2 :- all }
    sidl := // instantiate x in:
        {:- ∀ x · ¬(↑ == x) ⇒ ↑ sep x} SortedList(T).empty; // spec of empty
    {all } (sidl.add(item1)); // spec of SortedList::add etc
    {all } (sidl.add(item2)); // (see below)
    {all :- all ∧ item1 ∈ T ∧ item =  $\overline{\text{item1}}$ +50000 } item1.addOn(50000); // T::addOn
    {item1 ∈ T ∧ item1 = item10+50000 } smallest := Δ ∅ ∇ {sidl} · (sidl@1); // spec of @
)

```

8-3.2.3 An invariance proof

Elaborating one of these lines in detail will suffice to demonstrate the principles (and horribleness) involved:

```

{item1 ∈ T ∧ item1 sep item2 ∧ item1 sep sidl ∧ item2 sep sidl
  ∧ ∀ sli · sli ∈ sidl.sl.elems ⇒ item1 sep sli } (sidl.add(item2))

```

(Notice the invariant deals with item1 while the item being added is item2 .)

The proof begins with a restatement of the frame information in the signature of add:

```

1    sidl ∈ SortedList ⊢ Δ {sidl.sl.δ} · sidl.add(item2) from sig SortedList::add
2    sidl ∈ SortedList ⊢ ∇ {item2.δ, sidl.sl.δ, {sli.δ | sli · sli ∈ sl.elems}} · sidl.add(item2)
                                     from sig SortedList::add
3    sidl ∈ SortedList ⊢ Δ {sidl.δ} · sidl.add(item2)
                                     from 1, defn SortedList:δ by fx-expand

```

The requirement is to prove $\{all :- all\} \text{sidl.add}(\text{item2})$; the various conjuncts of the postcondition will be dealt with separately. The preservation of item1 's type can be derived from the separation of item1 from the writing-frame of the method:

```

4    all ⊢ item1 sep sidl by ∧-elim
5    ∇ item1.δ · item1 ∈ T by expr-∇ // see below
6    {all :- item1 ∈ T } (sidl.add(item2)) by implicit-invar from 3, 4, 5

```

The two items' separation is preserved because the operation writes to neither:

7 $\text{all} \vdash \text{item2} \text{ sep } \text{sidl}$ **by** \wedge -elim
8 $\{ \text{all} :- \text{item1} \text{ sep } \text{item2} \} (\text{sidl.add}(\text{item2}))$ **by** sep-preserve **from** 3, 4, 7

The separation of item1 and the list is preserved because item1 is separate from each element in the reading-frame of the operation:

9.h all
9.1 $\text{item1} \text{ sep } \text{item2}$ **by** \wedge -elim **from** 9h
9.2 $\text{item1} \text{ sep } \text{sidl.sl}$ **by** sep-specialise **from** 4, SortedList::dem
9.3.1 $\forall \text{sli} \cdot \text{sli} \in \text{sidl.sl.elems} \Rightarrow \text{item1} \text{ sep } \text{sli}$ **by** \wedge -elim **from** 9.h
9.3 $\text{item1} \text{ sep } \text{sidl.sl.elems}$ **by** sep-conjoin **from** 9.3.1
9.4 $\text{item1} \text{ sep } \{ \text{item2}, \text{sidl.sl}, \text{sidl.sl.elems} \}$ **by** sep-conjoin **from** 9.1, 9.2, 9.3
9 $\{ \text{all} :- \text{item1} \text{ sep } \text{sidl} \} (\text{sidl.add}(\text{item2}))$ **by** sep-preserve **from** 3, 4, 2, 9

The term involving the elements of the list is dealt with in a subproof about each individual element. The old members and the new one have to be dealt with separately; the isolation of the new one is explicitly guaranteed by add:

10 con sli //ranges over new elements
10.h1 $\text{sli} \in \text{sidl.sl.elems} \Leftrightarrow (\text{sli} \in \overline{\text{sl.elems}} \vee \text{sli} = x \wedge \text{sli.isolated})$ // postcond of add
10.h2 $\overline{\text{all}}$ //precond of add
10.h3 $\text{sli} \in \text{sidl.sl.elems}$
10.1 $\text{sli} \in \overline{\text{sl.elems}} \vdash \text{item1} \text{ sep } \text{sli}$ **from** 10.h2 **by** \wedge -elim, \forall -elim
10.2 $\text{sli} = x \wedge \text{sli.isolated} \vdash \text{item1} \text{ sep } \text{sli}$ **by** def isolated
10.3 $\forall \text{sli} \cdot \text{sli} \in \text{sidl.sl.elems} \Rightarrow \text{item1} \text{ sep } \text{sli}$ **from** 10.1, 10.2, 10.h3 **by** \forall -intro, \wedge -intro
12 $\{ \text{all} :- \forall \text{sli} \cdot \text{sli} \in \text{sidl.sl.elems} \Rightarrow \text{item1} \text{ sep } \text{sli} \} (\text{sidl.add}(\text{item2}))$
from 10, add-1 **by** strengthen

Finally, the separation of the operands is implied by the specification itself. Notice that according to this specification, clients are not allowed to add the list (or anything containing it) to itself:

13 $\{ \text{all} :- \text{item2} \text{ sep } \text{sidl} \} (\text{sidl.add}(\text{item2}))$ **by** SortedList::add-2
14 $\{ \text{all} \} (\text{sidl.add}(\text{item2}))$ **by** post-conjoin **from** 6, 8, 9, 12, 13

8-3.2.4 SortedIdList used with copied arguments

SortedIdList, which keeps a list of references to the original items, allows more flexibility for those cases where copying would be slow; but the client has to beware breaking the invariant by altering any items while they belong to the list.

SortedIdList(T)		
op add ∈	(x:T)	$\Delta \{sl.\delta\} \quad \nabla \{x.\delta, sl.\delta, \{sli.\delta \mid sli \cdot sli \in sl.elems\}\}$
fn _@_ ∈	(Nat) → T	$\Delta \emptyset \quad \nabla \{sl.\delta, \{sli.\delta \mid sli \cdot sli \in sl.elems\}\}$
add-1: $x \cdot \{x \in T \text{ :- } x = \overline{x} \wedge \forall sli \cdot sli \in sl.elems \Leftrightarrow (sli \in \overline{sl.elems} \vee sli = x)\} \text{ add}(x)$		
add-2: $x \cdot \{x \text{ sep self } \} \text{ add}(x)$		
$\{i \in \text{Nat} \wedge i \in 1..sl.length \text{ :- } \uparrow = sl@i \} \text{ self}@i$		
var sl ∈ IdList(T)		
$\forall i, j \in 1..sl.length \cdot i < j \Rightarrow sl@i \leq sl@j$		
fn $\delta = \{sl.\delta\}$		

The guarantees about separation are missing. However, this client chooses to do its own copying for every argument, so that the effect is just like using SortedList:

```

{item1 ∈ T ∧ item2 ∈ T
:- (smallest = item1 ∨ smallest = item2) ∧ smallest ≤ item1 ∧ smallest ≤ item2 } (
    sidl := SortedIdList(T).empty;                                // make new list
    {∀ sli · sli ∈ sidl.sl.elems ⇒ item1 sep sli}
    sidl.add({:- ↑.isolated }item1.deepCopy);                    //put a copy of item1 in the list
    {∀ sli · sli ∈ sidl.sl.elems ⇒ item1 sep sli}
    sidl.add({:- ↑.isolated }item2.deepCopy);                    //convention of deepCopy
    {sidl ∈ SortedIdList ∧ item1 sep sidl.elems }
    item1.addOn(50000);                                           // mutate original
    {sidl ∈ SortedIdList :- ... } smallest := sidl@1;           // guaranteed smallest
)

```

The isolation of each new member is guaranteed by deepCopy, while the separation of the existing ones can be shown using sep-preserve.

8-3.2.5 Commentary

Apart from demonstrating once again the prolixity and tedium of raw proof, the example suggests that a great deal of effort may be spent in verifying the continued separation of the entities involved, mostly so as to demonstrate the invariance of type membership (to guard against the possibility of aliasing): §8-3.2.3 shows a sub-proof of the sketch in §8-3.2.2, which is chiefly about $\text{item} \in T$. The preservation of these invariants is in turn subsidiary to the real matter of the proof (§8-2.3 — p.131).

Notice that the responsibility of proving that all invariants are not broken is effectively placed entirely on the client. A client using several servers will have to keep track of the integrity of each variable used, each with a similar cost to that above.

Fortunately, many parts of this kind of proof seem straightforward, with good hope of mechanisation by reasonably intelligent tactics.

(The toll could possibly be alleviated somewhat by the invention of a shortcut method for saying "SortedList looks after its own invariants and those of its parameters — no further proof necessary". This is not pursued here.)

However, the heavy load of proof upon the client is not due to any breach of encapsulation: the proof of the client's code shown above involves nothing from the implementation of the class. Rather, the conditions under which the type may be used have been spelled out, and the client has to prove compliance.

8-3.3 Inference of frame of an expression

The proof above included the lines:

```

1      sidl ∈ SortedList ⊢ Δ {sidl.sl.δ} · sidl.add(item2)          from sig SortedList::add
5      ∇ item1.δ · item1 ∈ T                                         by expr-∇

```

This section discusses the rules for determining the frames of a statement or expression. It is important to realise that a framing assertion (beginning with Δ or ∇) is a statement about the syntax of an expression, and the rules apply to the expression's syntax, not the result it yields. So for example, in a context in which $\text{item1} \in T$, the expression $\text{item1} \in T$ could elsewhere be substituted by true; but while the reading-frame of $\text{item1} \in T$ is $\nabla \text{item1}.\delta$, the reading-frame of true is \emptyset . Substitutivity of equals does not apply to the expression following "·", although it may be applied to the framing expression itself.

In many cases, the rules for read- and write-frames are the same, so the symbol \diamond stands for either ∇ or Δ .

8-3.3.1 Constants

A constant — type names, metavariables, and members of types with no mutating operations (such as the numbers) — can be ignored, since frames are about possible interference through unexpected mutations:

$$\text{fx-const:} \quad \diamond \emptyset \cdot (c) \quad \text{— } c \text{ a constant}$$

8-3.3.2 Variables

Using a variable reads the reference in that variable (whether the resultant object is read depends on the expression which uses it):

$$\text{fx-var-r:} \quad \nabla \&v \cdot (v)$$

and writes to nothing:

$$\text{fx-var-w:} \quad \Delta \emptyset \cdot (v)$$

8-3.3.3 Operations

A messages's frames are advertised with the type description. The frame (reading or writing) of an invocation is the union $\bigcup_i d_i$ of the frames d_i of the arguments E_i , and the frame of the message itself. The latter is usually expressed in terms of the parameters ($d_{op}[p_i]$), and must be instantiated with the yields of the arguments.

$$\text{fx-op:} \quad \frac{\begin{array}{c} \diamond d_{op}[p_i] \cdot (p_0 \text{ op } (p_{i>0})) \\ \diamond d_i \cdot e_i \end{array}}{\diamond d_{op}[e_i] \cup \bigcup_i d_i \cdot e_0 \text{ op } (e_{i>0})} \quad \text{— } e_i \text{ pure}$$

A rule allowing for e_i with side-effects would be too complicated: such a program can be rewritten as a series of assignments.

Any composition of pure expressions is pure; this applies to operations declared with **fn**:

$$\text{pure-form:} \quad \frac{\begin{array}{c} \Delta \emptyset \cdot E \\ \Delta \emptyset \cdot R[v] \end{array}}{\Delta \emptyset \cdot R[E]}$$

A *transparent* operation is one which reads only its arguments:

$$\text{transp:} \quad \frac{\begin{array}{c} f \text{ transparent} \\ \nabla \rho_i \cdot E_i \end{array}}{\nabla E_i \cdot \delta \cup \bigcup_i \rho_i \cdot E_0.f(E_i)}$$

Most of the operations on primitive types are pure and transparent — for example,

h	$v \in \text{Int}$	
1	$\nabla \&v \cdot v$	by fx-var-r
2	$\nabla \emptyset \cdot 5$	by fx-const
3	$x, y \cdot x \in \text{Int} \vdash x+y \text{ pure \& transparent}$	//documentation of Int::+
4	$\nabla \{\&v, v.\delta\} \cdot v+5$	from 3, h, 1, 2 by fx-op
5.1	$v.\delta = \emptyset$	from h by Int:: δ //all ints are immutable
5	$\nabla \{\&v\} \cdot v+5$	from 5.1, 4 by subs=

The built-in identity comparison operation function == depends only on the identities of the operands:

$$\text{fx-id:} \quad \frac{\begin{array}{c} \diamond d1 \cdot E1 \\ \diamond d2 \cdot E2 \end{array}}{\diamond d1, d2 \cdot (E1 == E2)}$$

(An '=' method would usually read the demesnes of the objects as well.)

An opspec governing a pure expression can be recast as an invariant:

$$\text{pure-opspec:} \quad \frac{\begin{array}{c} E \Delta \emptyset \\ \{ P :- R[\uparrow] \} E \end{array}}{P \vdash R[E]}$$

8-3.3.4 Compound statements

A sequence of statements E_i reads and writes the union $\bigcup_i d_i$ of everything its components do:

$$\text{fx-seq:} \quad \frac{\diamond d_i \cdot E_i}{\diamond \bigcup_i d_i \cdot (E_1; E_2; \dots)}$$

Conditionals and loops are pure and transparent:

$$\text{fx-cond:} \quad \diamond b \cdot B, \diamond t \cdot T, \diamond f \cdot F \vdash \diamond b \cup t \cup f \cdot (B \text{ ifTrue: } [T] \text{ ifFalse: } [F])$$

$$\text{fx-loop:} \quad \diamond b \cdot B, \diamond t \cdot T \vdash \diamond b \cup t \cdot ([B] \text{ whileTrue: } [T]) \quad \text{--- } t \text{ invariant}$$

Throughout this discussion, the frames considered are the outer bounds for any possible execution: thus the frame of a conditional statement is the union of the frames of its branches; and the frame of a loop can be determined only from a frame-spec which can be shown to be true of its body for every execution. It would be possible to make a finer analysis in which frame-specs are a form of postcondition, which may differ depending on the starting conditions.

In inline form, these rules can be written:

$$\text{fx-seq:} \quad \diamond \bigcup_i d_i \cdot (\diamond d_1 \cdot E_1; \diamond d_2 \cdot E_2; \dots)$$

$$\text{fx-cond:} \quad \diamond b \cup t \cup f \cdot (\diamond b \cdot B \text{ ifTrue: } [\diamond t \cdot T] \text{ ifFalse: } [\diamond f \cdot F])$$

$$\text{fx-loop:} \quad \diamond b \cup t \cdot ([\diamond b \cdot B] \text{ whileTrue: } [\diamond t \cdot T]) \quad \text{--- } t \text{ invariant}$$

8-3.3.5 Assignment

Assignment reads nothing, but care must be taken if any existing frame was expressed in terms of that variable:

$$\text{fx-ass-r:} \quad \frac{\nabla \rho \cdot E}{\nabla \rho[v\bar{v}] \cdot (v:=E)}$$

An assignment to a variable writes to that variable, and there are the same precautions:

$$\text{fx-ass-w:} \quad \frac{\Delta \omega \cdot E}{\Delta d[v\bar{v}] \cup \{\&v\} \cdot (v:=E)}$$

The use of a local variable v can be forgotten when we leave its scope, but the rule only works if the frame expression involves v only as a reference:

$$\text{fx-var-elim:} \quad \frac{\Diamond \{d, \&v\} \cdot E}{\Diamond d \cdot (\text{var } v \cdot E)} \quad \text{--- } v \text{ does not occur free in } d$$

In this example, a variable is used to hold a reference to a value temporarily ($c1$ is supposed to be a list of references to complex objects of some kind):

```

 $\nabla c1.\delta \Delta \&c2 \{ c1 \in \text{IdList}(\text{NatCell}) :- c2 == c1@3 \}$ 
(
  var v;
   $\nabla c1.\delta, \&v \Delta \&v, \&c2 \cdot ($ 
     $\nabla c1.\delta \Delta \&v \cdot v := c1+3;$ 
     $\nabla \&v \Delta \&c2 \cdot c2 := v$ 
  )
)
```

The same rule applies to bindings of all kinds, such as

$$\text{fx-var-elim-}\forall: \quad \nabla \{d, \&v\} \cdot E \vdash \nabla d \cdot (\forall v \cdot E) \quad \text{--- } v \text{ does not occur free in } d$$

If the content of a variable can be shown to be isolated (that is, separate from every other frame: §8-3.1.5 — p.137) then leaving the scope of a variable throws the information away:

$$\text{fx-var-elim-iso:} \quad \frac{\{ :- v \text{ isolated} \} E \quad \Diamond \{d, \&v, v\} \cdot E}{\Diamond d \cdot (\text{var } v \cdot E)}$$

8-3.3.6 Substitution

Where these elimination rules are not applicable, it is only necessary to recast the frame-expression in a form not involving v . If two expressions lead to the identical object, then each may be substituted for the other in a frame specification:

$$\text{fx-subs-==:} \quad \frac{\Diamond E1 \cdot S \quad E1 == E2}{\Diamond E2 \cdot S}$$

The overall effect of the following fragment should be that the list $c1$ is read, but we do the reading through the temporary variable v . The side-condition on fx-var-elim

$$\begin{array}{ll} \Delta \&c2 \nabla c1.\delta, \&c1 \cdot (& \text{var } v; \\ \quad \nabla \&c1, c1.\delta \Delta \&c2, \&v \cdot & // \text{fx-var-elim} \\ \quad \quad \nabla \&c1, v.\delta \Delta \&c2 \&v \cdot (& // \text{fx-subs} \\ & \quad \nabla \&c1 \Delta \&v \cdot \{ \text{:- } v == c1 \} \\ & \quad \quad v := c1; \\ \Delta \&c2 \nabla v.\delta \cdot & c2 := v@3 \\)) & \end{array}$$

Any operation writing to an isolated demesne can have no effect on any other; therefore the effect can be ignored.

$$\frac{\text{fx-copy-nop:} \quad \Delta \text{ d} \cdot \text{E} \quad \Delta \text{ x} \cdot \delta \cdot \text{x}, \text{p}_i \cdot (\text{x op})}{\Delta \text{ d} \cdot \text{E} \cdot \text{deepCopy.op}}$$

With the exception of the rule for variable elimination, it should be possible to apply these rules automatically. The syntactic mode of application does not, in any case, fall within the normal rule-matching method of the theorem prover. A special built-in “oracle” would be invoked, inserting a single justification in the proof, called `expr-Δ` above.

This section discusses an incompletely resolved difficulty with frames.

Consider as an example Shape, now with framing information:

Shape		
fn contains \in	$(p \in \text{Point}) \text{ Bool}$	$\nabla p.\delta, \text{self}.\delta$
op move \in	$(v \in \text{Vector})$	$\Delta \text{self}.\delta \nabla v.\delta, \text{self}.\delta$
<u>mv-def</u> : $v \cdot \{ v \in \text{Vector} :- \forall p \cdot p \in \text{Point} \Rightarrow$ $(\text{self.contains}(p) \Leftrightarrow \text{self.contains}(p+v)) \} \text{move}(v)$		

The signature states that `move` writes only to the `Shape`'s own components; but since there are no variables in this very abstract TCD, there is no point in defining δ : that must be left to the subtypes. E.g.:

FourSides ::+ Shape			
op setp1 ∈	(Point)	Δ self.δ	∇ self.δ, p.δ
op setp2 ∈	(Point)	Δ self.δ	∇ self.δ, p.δ
op setp3 ∈	(Point)	Δ self.δ	∇ self.δ, p.δ
op setp4 ∈	(Point)	Δ self.δ	∇ self.δ, p.δ
axfsc:	p · { :- ↑ = p.withinLoop(⟨p1, p2, p3, p4⟩) } contains(p)		
axfs1:	np · { nonIntersectingLoop(⟨np, p2, p3, p4⟩) :- p1=np } setp1(np)		
...			
var p1, p2, p3, p4 ∈ Point			

`FourSides` is still abstract, and can be refined with different constraints. The new Δ in `Quadrilateral` ensures that each vertex-setting operation leaves the others unmoved:

Quadrilateral ::+ FourSides			
op setp1 \in	(Point)	Δ p1. δ	// other points are fixed
...			

The new Δ is clearly OK, as it stipulates a subframe of the inherited one; the same restriction could be described with an extra postcondition.

However, this type adds a new redundant variable — a valid move in refinement — which the existing operations must clearly be allowed to alter:

QuadWithArea ::+ Quadrilateral	
var area \in	Area
area = areaWithin(p1,p2,p3,p4)	
fn $\delta = \{p1.\delta, p2.\delta, p3.\delta, p4.\delta, \text{area}.\delta\}$	

In this case, the extension of δ seems intuitively unlikely to cause any problem; but in general, arbitrary expansions of δ could permit interference unanticipated by clients, and would make separation proofs impossible except where you know the precise class you are using. The invariance proof of §8-3.2.3 — p.139 worked by expanding the definition of δ ; but if δ is allowed to be redefined in subtypes, that will not work. We need to be able to make statements about the separation of `self. δ` from arbitrary demesnes of concern to clients, while still allowing subtypes to choose their own implementations.

This suggests rules governing expansions of δ :

- Frames defined in operation signatures may only be reduced in subtypes; multiply-inherited frame-specifications intersect.
- Demesnes (i.e. δ -functions) may be reduced in subtypes; multiply-inherited demesnes intersect.
- A demesne may be expanded to a superset of its ancestors only if the extension is guaranteed to remain separate from every frame accessible to any

client.

To allow for such extensions, the rules about restriction of frames are applied *before* elaboration of δ -functions.

(A more formal version of these rules is a topic for future work.)

8-3.5 Type Isolation

But this raises the question of how universal separation from clients can be guaranteed. It is merely a matter of applying the well-known rule, observed by `SortedList` but not by `SortedList`, that clients and servers should never be passed pointers to your private demesne; nor should items to which pointers are passed to you by clients or servers ever be incorporated into your own demesne. (This does not preclude keeping references themselves, as such — `&p` is OK to keep, `p` is not.) For classes observing this regime, isolation can be proven by proving that it is maintained by each operation.

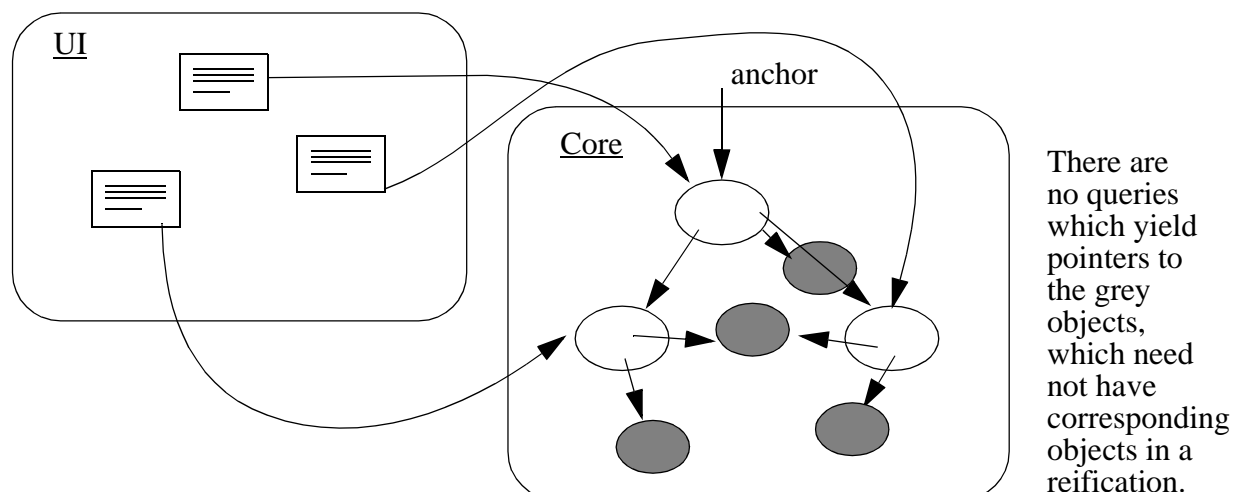
Isolation, characterised by an invariant `self.isolated`, should, like other properties, be observed by all classes claiming to be subtypes of this type. It can be used by clients (of `SortedList` and its subtypes, for example) as a short-cut to demonstrating separation from all other objects (using `isol-defn` — page 137).

8-3.6 Reification of subsystems

A system may be designed by splitting it into subsystems to which responsibilities are assigned, and then defining protocols of communication between them. The same process is reiterated on the subsystems, until individual objects are attained. (E.g. [WWC90].) The design of a typical interactive system will begin with the partitioning into the User Interface and Core subsystems, with the latter holding all the domain-specific information, and the former providing the means to browse and massage it. The UI will begin a session by presenting the end-user with an overall picture of the state of the Core — let's suppose it's a programming support system, and the initial view is of a list of classes.

At this stage, the UI has a pointer of some sort to an anchor object in the Core: that is, a fixed, globally accessible object, which provides methods for querying the

Fig. 13. Pointer access to parts of a subsystem



overall state — getting a list of classes. The user selects a class or other detail of the current display, and asks to see more information about that: the UI gets a pointer to that class from the anchor, and asks it for sufficient information to display details. The user then asks for more detail or some associated information, and again the UI gets another pointer to a core object. In other words, the user's requests lead to a navigation around the structure of pointers inside the core.

The earlier remarks on reification (§6-5.5 — p.102) assert that a class can be restructured internally whilst still looking the same from the outside. This is clearly not applicable in the present example: the query operations provided by the anchor object pass out pointers to its own components; since these can be looked at directly, they must remain after any reification. It would be permissible for a reification to try to simulate the old components by constructing them on the fly, but because part of the expected behaviour is that changes to these objects actually change the components of the anchor, the constructions would have to be two-way views, translating not only queries, but also editing operations.

But intuition suggests that there are some components of the core which really can be altered: those to which no direct access is given to clients external to the subsystem. How is this distinction to be made?

A query whose purpose is to yield the identity of a component should be documented with a postcondition that makes this explicit:

$$\{:- \uparrow == \text{aComponent} \} \text{getComponentPointer}$$

from which it may be inferred that $x.\text{getComponentPointer} \in x.\delta$; any reification must observe this constraint.

So it follows that, during design, those objects not directly accessible to the external clients may be reified away since they may be considered part of a hypothetical model. But queries yielding identities of components must continue to do so, and those objects must remain. This is a distinction not made in the current texts on OO design, which generally consider design to be an expansion of the existing model.

8-3.7 Effects calculus summary

The flavour of this section is very much more oriented towards actual objects and pointers than the preceding abstract stuff about models. This is in line with the Fresco objective of specifying pieces of interchangeable software — not just high-level OO-style specifications of whole programs. At the interface to such a module, it is clearly necessary to specify the interconnections (or lack of them) of the objects: $x \text{ sep } y$ is often a vital precondition for the parameters of an operation.

This section has introduced the notion of frame-specifications, which are typically written in terms of demesnes. This replaces the simpler framing clauses such as “**ext**” in VDM and “ Δ ” in Z. These do not provide the user with much help where there is much potential aliasing, and where fields may be altered which are not among the simply-named variables; nor do they cope with type extension.

The inference of frame of an expression is done with rules whose domain is the syntax of an expression rather than its value; nevertheless, framing assertions seem to fit into the general Fresco scheme of type theories. The quest for monotonicity has shown up a difficulty with the extension of the frame of an operation in a subtype — without carefully-chosen safeguards, aliasing could be introduced in a subtype which was not there in the supertype.

Although no claim can be made that the effects system presented here is well-trying or complete, it is claimed that it could only be designed within the context of a type system such as Fresco's. Effects systems have been used elsewhere: for example in [Johnson91], to assist with the optimisation of a compiler; but in that case, there are no type-specifications, and so encapsulation is poor — the effect-inferencer often has to re-analyse clients when a change is made. In Fresco, a type defines what behaviour all of its subtypes will conform to, and the Fresco effects system seeks to extend this to aliasing or its absence.

8-4 Naming previous states

8-4.1 Barred expressions

The essential thing about a barred assertion is that it remains unchanged by any operation:

$$\text{bar-invar:} \quad \text{pre}, S \cdot \{ \text{pre} :- \overline{\text{pre}} \} S$$

Many theorems involve barred expressions, as in:

$$\{ \langle P1 :- R1 \rangle S, \overline{P} \wedge R1 \vdash R, P \vdash P1 \} \vdash \{ P :- R \} S$$

An expression such as $\overline{x+5} > y$ can be rewritten as $\overline{x} + 5 > \overline{y}$, using these rules, which may be applied automatically:

$$\begin{array}{ll} \text{unbar-const:} & \overline{c} \equiv c \quad \text{--- iff } c \text{ is a constant or metavariable} \\ \text{unbar-transp:} & \overline{E0.f(Ei)} \equiv \overline{E0}.f(\overline{Ei}) \quad \text{--- iff } f \text{ is pure \& transparent} \\ \text{unbar-binding:} & \overline{\forall x \cdot E[x]} \equiv \forall x \cdot \overline{E[x]} \quad \text{--- \& same for other binders} \end{array}$$

No rules are provided for impure operations, as these should be avoided in assertions. Field-selectors (instance variable names) may be regarded as functions; a field is transparent if its demesne occurs in the demesne of the receiver.

The situation with untransparent functions is more complex. A function may be untransparent because it reads a global variable, or because it reads a field which is not part of the receiver's or parameters' demesnes, but nevertheless accessible through them (via a reference field). Consider an object b of type B :

B	
op	$\text{op} \in \quad () \Delta c.\delta, \text{self}.\delta$
$\{ :- b.c = \overline{b}.c + 1 \wedge \overline{b}.c = \overline{b}.c + b.\overline{c} \} \text{op}$	
var	$c \in \quad \text{IntCell}$
var	$x \in \quad X$

c is a mutable container for an integer. c 's demesne is not part of b 's. op alters the C-member to which c refers, as well as altering the contents of both the old and the new c . These are distinct:

- $b.c$ — the new value of the C-member now pointed to by field c
- $\overline{b.c}$ — the old value of the C-member now pointed to by field c
- $\overline{b}.c$ — the old value of the C-member $b.c$ used to point to
- $\overline{\overline{b}.c}$ — the new value of the C-member $b.c$ used to point to

Reference fields and other untransparent functions should clearly be treated with caution.

8-4.2 Metavariables and code

8-4.2.1 Potential inconsistencies in interpretation of metavariables

Recall that for a sequence of two statements, the sequence rule is:

$$\text{sequence: } P, M, R, S1, S2 \cdot \frac{\begin{array}{c} \{ P :- M \} S1 \\ x_0. \{ M[\bar{x}\backslash x_0] :- R[\bar{x}\backslash x_0] \} S2 \end{array}}{\{ P :- R \} (S1; S2)}$$

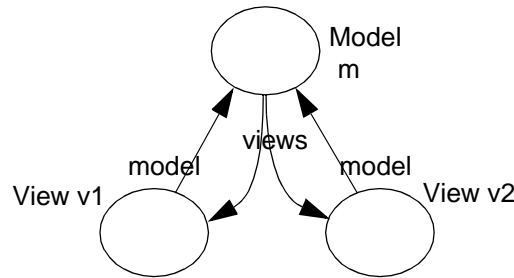
(§5-2.1.3 — p.74) a sample application being

```

{ x ∈ Int :- x =  $\bar{x}$ +7 } (                                     //sequence
  con x0;                                                    //kinder on reader to declare it, as in [Morgan]
  { :- x =  $\bar{x}$ +5 }                                             x := x+5 ;
  { x=x0+5 :- x = x0+7 }                                     // subs=, arith
    { x=x0+5 :-  $\bar{x}$ =x0+5 ∧ x= $\bar{x}$ +2 }                         // unbar
      { x=x0+5 :-  $\bar{x}$ =x0+5 ∧ x= $\bar{x}$ +2 }                       // { P :-  $\bar{P}$  }
        { :- x =  $\bar{x}$ +2 } x := x+2
    )
)
```

The purpose of the metavariable x_0 is to be the name of any arbitrary value satisfying M and which will remain unaltered by any operation in the code (and can therefore be unbarred). To be clear about the meaning of x_0 , we can imagine an infinite store containing all possible patterns of objects (in addition to the ‘real’ ones we’re interested in and can get at through the real variables); x_0 is a name for any of those which satisfy the assertions and are outside the frame of all our operations. However, it is by no means clear that such a thing can always exist.

For suppose we have a model-view framework, with instances thus:



Every model has a variable $\text{views} \in \text{IdSet}(\text{View})$, and the invariant observed by every View is $\text{self} \in \text{model.views}$.

Within the reasoning about some code modifying $v1$, I wish to refer to its original state, and so introduce vw_0 , a hypothetical View which satisfies the same predicate (matching M in sequence) as $v1$. It may not be necessary that $v1 = vw_0$, or even that there be an equality operator defined for Views: only that they behave the same in respect of the predicate chosen to match M . It is often sufficient to postulate that the constant represents some hypothetical other object which could exist in the system — this works for arithmetic examples and many others; but not in every case.

For example, according to the invariant, $vw_0 \in vw_0.\text{model.views}$. Now $vw_0.\text{model} == m$ — if it is required by our M — should evaluate to true iff $v1.\text{model} == m$ does, and by

transitivity of $==$, $vw_0.model == v1.model$. But since $v1.model.views$ contains only the “real” views, with no apparent mention of vw_0 , then either the invariant is contravened or $vw_0 == v1$. How can $vw_0 == v1$ while vw_0 is nevertheless separate from $v1$?

Two strategies are considered here. One is to reduce our ambitions about constancy of metavariables, and the other is to extend our model of the system.

8-4.2.2 Metavariables are constant pointers only

In this solution, a metavariable is a new variable which contains a pointer to some object, which may be any postulated *possible* object. Metavariables are never assigned or used in code, so the pointer itself is constant; but the object it points to, and hence the value it represents, is not immune from alteration.

In the case of primitive and other immutable objects, this makes no difference from standard practice. In the case of mutable objects, the designer must stipulate explicitly any separation the object is to have. So when dealing with Points,

$$\{ p \in \text{Point} :- p = \bar{p} + \text{Point.xy}(7,7) \} ($$

$$\quad \text{con } p_0;$$

$$\quad \{ :- p = \bar{p} + \text{Point.xy}(5,5) \} p.\text{move}(5,5);$$

$$\quad \{ p = p_0 + \text{Point.xy}(5,5) \wedge p_0 \text{ sep } p :- p = p_0 + \text{Point.xy}(7,7) \wedge p_0 \text{ sep } p \}$$

$$\quad \quad \{ p = p_0 + \text{Point.xy}(5,5) \wedge p_0 \text{ sep } p$$

$$\quad \quad :- p = p_0 + \text{Point.xy}(5,5) \wedge p = \bar{p} + \text{Point.xy}(2,2) \wedge p_0 \text{ sep } p \}$$

$$\quad \quad \{ p_0 \text{ sep } p :- p_0 \text{ sep } p \wedge \text{Point.xy}(2,2) \} p.\text{move}(2,2)$$

$$))$$

(To reduce clutter, $p_0 \in \text{Point}$ has been omitted from every pre & postcondition.)

The unbaring of p_0 is permitted here only because it is not in the frame of $p.\text{move}(2,2)$, (because $p_0 \text{ sep } p$); unbaring of $\text{Point.xy}(5,5)$ is permitted because it is specified as generating an isolated value.

If the model-view problem is to be dealt with under this scheme, the designer must first invent a special type of view which does not necessarily conform to the invariant, and define equality etc to proper Views. Alternatively, a separate constant may be used to hold each component of the View.

8-4.2.3 Metavariables refer to ghost copies

In this scheme, any object-address may be occupied by one real object and many ghost objects. A real variable always refers to a real object, and a metavariable points to a ghost object, possibly at the same address as a real one. The contents of the ghost need not be the same as the other objects at the same address, and are set by the constraints where the metavariable is used. Each instance variable of a ghost object point at another ghost, if the field is within the object’s demesne; and at a real object otherwise. The application of mutating operations to ghosts is undefined.

The use of a metavariable therefore implies a sort of transitive copy (“deepCopy”) from real to ghost objects, which have the advantage that they can occupy the same addresses as real objects, thus permitting $vw_0 == v1$ whilst allowing $v1$ to change without affecting vw_0 . The extent of the copying is entirely determined by the user’s definition of δ for the relevant classes.

8-4.3 Bars and metavariables summary

In traditional program proof, ghost constants are used freely under the assumption that an isolated constant of any value can be postulated. In OOP, where values are represented by groups of interlinked objects, such an assumption is invalid. This section has looked at the alternatives.

Whilst the “ghost copies” model is a more convenient scheme for the user, it currently lacks formalisation, and the first alternative appears more clearly-defined and therefore safer. However, the duty of explicitly stipulating and proving the maintenance of separation is onerous. Proofs elsewhere in this thesis must be regarded as only semi-formal because of this (deliberate) omission.

Dealing with potential aliasing is clearly a potential drawback in this as in other areas.

8-5 Projection to supertype, and equality

8-5.1 Equality and subtypes

It seems obvious that a definition of equality in this style is meaningful:

Point::def-eq: $a \in \text{Point}, b \in \text{Point} \vdash (a = b \vdash a.x = b.x \wedge a.y = b.y)$

but we have to be careful about the meaning of ‘=’. Membership of **Point** does not preclude membership of any of its subtypes such as **ColouredPoint**, which contain more information: if **a** and **b** were to turn out to be members of **ColouredPoint**, then we would want an equality test to check their colours as well.

So (a useful definition of) equality is not monotonic. Designers can follow two strategies. One is to define a separate equality for each type in the hierarchy:

$$a \in \text{Point}, b \in \text{Point} \vdash (a =_{\text{Point}} b \Leftrightarrow a.x = b.x \wedge a.y = b.y)$$

$$a \in \text{ColouredPoint}, b \in \text{ColouredPoint} \vdash$$

$$(a =_{\text{ColouredPoint}} b \Leftrightarrow a.x = b.x \wedge a.y = b.y \wedge a.\text{colour} = b.\text{colour})$$

A less tedious notation involves the projection of the objects into the bottom element of a type.

8-5.2 Projection operator

|·-defn: $x \in T, y \in T \vdash (x|T \in T \wedge x|T = y|T \equiv \bigwedge f \cdot x.f = y.f)$
for all functions *f* defined for *T*.

$x|T$ (read “*x* as *T*”) is the projection of *x* into the bottom element of *T*: there is no other type *T*’ such that $T' \subseteq T \wedge T' \neq T \wedge x|T \in T'$.

Designers can now make useful assertions about equality; for example:

$$a \in \text{Point}, b \in \text{Point} \vdash (a|_{\text{Point}} = b|_{\text{Point}} \equiv a.x = b.x \wedge a.y = b.y)$$

which remains true for all subtypes of **Point**. Of course, since we have decided that equality is always relative to a given type, we should be careful here about what ‘*a* *x* = *b* *x*’ means. For primitive values, ‘=’ is the same as ‘==’, so in those cases there is no problem. More generally, the equality of the components will be tested according to the types that we expect them to have, and extra details they might have that

we don't know about are irrelevant, because they can't make any difference to our design. For example, if I define a finite straight line by its end-points, then so far as I am concerned, two lines are equal if the respective end-points have equal co-ordinates; now suppose some client then decides to set up a couple of my lines using ColouredPoints: that's OK, but the colours are irrelevant to any property of the lines.

8-5.2.1 Definition of equality for a type

In general, if a designer wishes to state that T has a view (i.e. a determining set of variables or queries) whose components are $x_i \in T_i$ then a definition of equality may be written according to the scheme:

$$\text{eq-}T: \quad a, b \cdot a \in T, b \in T \vdash a|T = b|T \Leftrightarrow \bigwedge_i a x_i | T_i = b x_i | T_i$$

and we can write

$$a =_T b \text{ to abbreviate } \bigwedge_i a x_i | T_i = b x_i | T_i$$

Now what happens if we try to compare two objects of different types? No rule of the form $\text{eq-}T$ from either of their types will apply; but such rules defined for their common supertypes will apply:

$$a \in A, b \in B, A \subseteq C, B \subseteq C \vdash a|C = b|C \Leftrightarrow a =_C b$$

Notice that there's no assertion here that *the* equality of a and b depends on what their common supertype is: the absolute notion of equality has been discarded. What is asserted here is that anyone who can't tell an A from a B should be satisfied with C 's definition of equality.

If there is definitely no way that any object can be invented that is both an A and a B , then any member of A is not equal to any member of B either in terms of A , or B :

$$\text{neq-}T: \quad a, b \cdot a \in A, b \in B, A \cap B = \emptyset \vdash a|A \neq b|A \wedge a|B \neq b|B$$

8-5.3 Substitution

The familiar substitution rule

$$P[a], a = b \vdash P[b]$$

looks a bit less useful than before subtyping, since in most cases, we are not be able to say more than $a|T = b|T$. However, consider the situations in which it is applied: in nearly all cases, predicates matching P will deal only in those features understood in the context in which we are working. Interesting inferences we make about Points, for example, will rely only on the structure we have defined at that juncture, and not on any future added information which descendants might contain. In such cases, we can therefore employ a modified version of the above rule:

$$\text{subs-eq} : \quad a, b, P, T \cdot a \in T, b \in T, P[a|T], a|T = b|T \vdash P[b|T]$$

If P only knows/cares about the T -information in these objects, then equality of their T -projections will be sufficient for it.

However, we can still write the old rule in terms of object identity:

$$\text{subs-id} : \quad a, b, P \cdot P[a], a == b \vdash P[b] \quad \text{from subs-eq, eq-}\bot$$

8-5.4 Application of rules

Matching a rule to an assertion often requires a judgement of equality: in

$$x \in \text{Scalar}, y \in \text{Scalar}, z \in \text{Scalar} \vdash x < y \wedge y < z \Rightarrow x < z$$

y stands for a subexpression which must be matched identically in each occurrence. Conventional theorem-proving relies heavily on the frequent application of the local version of **subs-eq**, but Fresco's version is weaker than usual. So the above rule will be much more widely applicable if we can make it

$$x \in \text{Scalar}, y \in \text{Scalar}, z \in \text{Scalar} \vdash x < (y|\text{Scalar}) \wedge (y|\text{Scalar}) < z \Rightarrow x < z$$

as it will frequently not be possible to do more than obtain the *typed* equality of two occurrences.

8-5.5 Comparison with conventional LPF

This conventional rule is omitted in Fresco:

$$\text{=subs:} \quad E, s1, s2 \cdot \quad E[s1], s1 = s2 \vdash E[s2]$$

Now in some sense, it is up to the designer of a type to decide what equality means for members of that type. However, because equality is central to rule application, we impose certain constraints. The above and the following are inherited by every type, and so any definition which attempts to contradict them will be unimplementable:

$$\begin{array}{lll} \text{=subs:} & E, T, s1, s2 \cdot & E[s1|T], s1=s2 \vdash E[s2|T] \\ \text{id=}: & a, b, T \cdot & a == b, a \in T \vdash a|T = b|T \\ \text{\(\neq\)\(\sim\)\(\sim\)}: & a, b, T \cdot & a|A \neq b|B \vdash \neg(a==b) \\ \text{antimono-}: & a, b, T, TT \cdot & a \in T, b \in T, T \subseteq TT, a|T = b|T \vdash a|TT = b|TT \\ \text{mono-\(\neq\)}: & a, b, S, T \cdot & a \in T, b \in T, S \subseteq T, a|T \neq b|T \vdash a|S \neq b|S \\ \text{uneq-types-l:} & a, b, A, B \cdot & a \in A, b \in B, A \cap B = \emptyset \vdash a|A \neq b|A \end{array}$$

The Logic of Partial Functions as described in [CJ90] uses a delta-function (unrelated to Fresco's demesne δ) to denote whether a term signifies a meaningful value: so $x \in \text{Integer} \vdash \delta(x/0) = \text{false}$; in Fresco, only meaningful terms are members of types (other than \perp), so this would be written $x \in \text{Integer} \vdash ((x/0) \in \text{Integer}) = \text{false}$. Clearly only the possession of a definite type gives access to any theories: it is a rare theorem of any method or function which does not require the operands to have definite types. Little can therefore be proven about untyped terms, and terms formed from them are therefore also untyped.

8-5.6 Summary of typed equality

This section has investigated some consequences of the un-inheritability of comparison operators, and in particular equality. The usual substitution of equals rule should not be used: you must always say which equality you mean. This arises entirely from the desire to use a type as a specification of a server for a polymorphic client, which may be seen as the especial pursuit of this thesis.

8-6 Summary

In attempting to clarify and resolve those particular difficulties associated with reasoning about object-oriented programs, this chapter has demonstrated the utility, and some of the limitations, of the overall Fresco approach of expressing types as theories:

- The intricate contracts in complex frameworks can be characterised as types within the weak typing interpretation.
- Type theories have provided a framework within which to apply a framing calculus. The strictly monotonic approach imposed by types-as-theories has exposed the difficulty of making frame specifications monotonic.
- Reification of types which pass pointers to components has been clarified.
- In the presence of much potential aliasing, the standard ghost-variable approach to verification of code sequences has turned out to be fraught either with difficulties of interpretation, or with separation proofs.
- The Freco view of subtyping has exposed a limitation in the conventional view of equality.

Problems with the framing calculus:

- Separation proofs seem even huger and more laborious than most.
- The rules are incomplete and little-tried as yet.

Object-oriented programs are not yet the stuff of which nuclear power stations ought to be built. However, the application of a framing calculus within the context of proofs seems likely to be a step towards the alleviation of the aliasing problem, since it is frequently the attempt to do proofs, either formally or informally, that points the way to an improved style of programming. And while none is presented here, it is reasonable to expect that effective elisions of separation proofs will be found in future work.

9 Conclusion

9-1 Overview

The objective of this work has been to set up a framework in which object-oriented software can be specified, verified and exchanged in usefully re-usable units. Polymorphic and incremental design are supported. The overall aims and the envisaged mode of use of Fresco were described in Chapter 2.

The key notions, described in Chapter 6, are the formulation of types as theories of object-histories, and subtypes as sub-theories. These theories are defined as sets of opspecs, the basic theorems of object behaviour, described in Chapter 5. Chapter 4 described theories and proofs.

Using these ideas, Chapter 7 showed how “capsules” of specified software are generated and incorporated into systems in such a way as to ensure the correct operation of each successfully incorporated capsule. Because of the constraints to correct subtyping, new versions of servers always continue to work with old clients.

Finally Chapter 8 used the previously-described concepts as a language in which to investigate a number of difficult issues in the formalisation of OO programming.

9-2 Assessment

The utility of the key concepts has been demonstrated in three ways:

- Construction of a number of examples throughout the text, showing the way in which types, implementations, and proofs will be constructed by Fresco-users.
- As the basis of the guarantees of correct integration of capsules.
- In the formulation of the discussions of Chapter 8, and the light shed on those issues. Although firm conclusions were not reached on all of these topics, their exposure in itself demonstrates the value of the Fresco approach to types and proofs.

The Fresco style of specification is, I believe, readable and not difficult for good programmers to learn. Type hierarcchies provide good structures around which to hang formal specification, which otherwise tends to be daunting. The style can be integrated with diagrammatic OO analysis and design methods such as [Coad], [Rumbaugh]: it provides, for example, for multiple appearances of a type in a document.

Fresco is much more oriented towards programming than, for example, Object Z, dealing as it does with concerns like aliasing.

Unlike many OO design texts and OOP languages, Fresco permits and promotes the use of data-reification, in which the concrete model used for implementation is not necessarily an extension of the abstract model used for specification.

Proofs are, as usual, enormous and tedious: it seems as unlikely as ever that real programmers will do them much without significantly more mechanical help. Although some progress has been made on tackling aliasing, taking such considerations into account serves to expand *every* proof — not just those where aliasing is suspected.

9-3 Future work

Several directions can be perceived:

- The work described here lacks the concrete touchstone of a testbed. A prototype Fresco should be built. To date, an experimental framework of the capsules mechanism has been designed, which provides some help in structuring published software, but without the extra security provided by formal specification.
- The system should then be used to specify, design and build a number of practical capsules, with semi-formal proofs.
- The semantics of barred variables needs further clarification.
- The rules of the effects calculus need larger trials and further refinement and formalisation.
- The user interface to the stepwise development and browsing of inline proofs is clearly crucial to the usefulness of the tool.
- It should be easy to mix ordinary stepwise refinement of code with the invocation of code-transformational rules, chosen and applied in the same way as proof rules and tactics. This would be an interesting investigation of coding technique.
- Methods of verification in respect of aliasing need to be improved. Mechanical assistance ('tactics') for the straightforward parts of these proofs should be developed.
- A number of tools supporting the OOA/D methods are appearing; none of them supports formal methods, and it would be interesting to make such an extension.

Bibliography

- AG91 AJ Alencar, JA Goguen: OOZE: an object-oriented Z environment. In [ECOOP91] pp 180–199
- America 87 Pierre America: Inheritance and subtyping in a parallel object-oriented language. in [ECOOP 87] pp 234–242.
Argues for separation of inheritance and subtyping.
- America 90 Pierre America & Frank de Boer: A Proof system for process creation. in Workshop on Foundations of Object-Oriented Languages, Noordwijkerhout, the Netherlands, May 1990
- BCJ84 H. Barringer, J.H. Cheng, and C.B. Jones.
A logic covering undefinedness in program proofs.
Acta Informatica, 21:251–269, 1984.
- Berard Ed Berard. Article in comp.software-eng Usenet newsgroup 22 Jun 1989.
OOD needs a ‘front end’. Bitter experience shows that should be object-oriented (like OORA [Coad]).
- Black 86 A Black, N Hutchinson, E Jul, H Levy. Emerald. in [OOPSLA 86].
Strong optionally static typing. Type conformity solely on basis of messages understood. No explicit classes: individual creation procedures; an abstract type is just a signature of operations — no model.
- Booch Grady Booch *Object oriented design*
- Borning 87 Alan Borning, Tim O’Shea: Deltatalk: An empirically and aesthetically motivated simplification of the Smalltalk-80 language. in [ECOOP 87] pp 1–10
Some small improvements, some of which are incorporated in the latest release. “Modules” are recommended as partitions of the global name-space, after the one-level selective-export style of Modula.
- Cardelli 85 Luca Cardelli, Peter Wegner: On understanding types, data abstractions, and polymorphism. *ACM Computing Surveys*, Vol.17, No. 4, December 1985, pp. 471–522.
Types are sets of values. Any one language may be able to describe a subset of the universe of types; a polymorphic language is one which can describe intersecting types. Quantifiers in types: universal for type parameters in function signatures; existential for modularisation with hiding, via the encapsulation of groups of functions within records.
- CDDKRS90 DA Carrington, D Duke, R Duke, P King, GA Rose, G Smith: Object-Z: an object-oriented extension to Z. In S Vuong (ed): *Formal Description Techniques FORTE’89* pp401–420, North Holland, 1990.
- CHC 90 William R Cook, Walter L Hill, Peter S Canning: Inheritance is not Subtyping. *ACM Trans. on Programming Languages and Systems* 1990
- CIP 87 *Munich Project CIP: CIP-L Language.* (LNCS 183) and
Munich Project CIP: CIP-S System. (LNCS 292). Springer-Verlag 1987.
Wide-spectrum language and transformational program development system. The syntactical class of expressions includes: predicate calculus; a recursion abstraction; typed lambda-definitions with preconditions and higher-order argument and result types; descriptive expressions (that $x:T \cdot \dots$; some $x:T \cdot \dots$), with which postconditions may be written; non-deterministic guarded conditionals

(if B_1 then E_1 [] B_2 then ...). Procedural sequences and loops are defined formally in terms of function composition and recursion expressions. ADTs may be defined algebraically, with 1st-order predicates over [in]equalities; or with a computational model and individually-defined functions.

- CJ 90 Jen Cheng, Cliff Jones: On the usability of logics which handle partial functions. In C.Morgan, J.Woodcock (eds) *Proc. 3rd Refinement Workshop* Springer 90. Contrasts various logics for dealing with partial functions and undefinedness, including LPF (Cheng & Jones), McCarthy's conditional operators, Weak Logic (Owe, part of Abel project), PFL (Plotkin), LCF (Milner et al). LPF wins.
- CL90a Marshall Cline and Doug Lea: Using Annotated C++ *Proceedings, C++ at Work* Meadowlands NJ, Sept 90
- CL90b Marshall Cline and Doug Lea: The Behaviour of C++ Classes. *Proceedings, Symp OOP Emphasizing Practical Applications* Marist Coll., Sept 90
- Coad Peter Coad *Object oriented requirements analysis* [Prentice-Hall] 2nd edn 1991 ISBN 0-13-629981-4
- Cox 86 Brad Cox: *Object-oriented programming: an evolutionary approach*. A-W 86
- Cusack90 Elspeth Cusack: OO specification in LOTOS and Z, or, my cat really is object-oriented! in [FOOL90]
- Danforth Scott Danforth and Chris Tomlinson: Type theories and object-oriented programming. *ACM Computing Surveys* 20(1) pp29–72.
- DD90 David Duke, Roger Duke: Towards a semantics for Object-Z. In [VDM90] pp244–261
- DLO86 Ole-Johan Dahl, Dag F. Langmyhr, Olaf Owe: *Report on the Specification and Programming Language ABEL*. Uni. Oslo, Inst. Informatics. RR106 Dec 1986. ISBN 82-7368-006-1
- DT92 M Dodani, C-S Tsai: ACTS: A type system for object-oriented programming based on abstract and concrete classes. [ECOOP92]
- DK91 Eugene Dürr, Jan van Katwijk: VDM++: a formal specification language for object-oriented designs. *TOOLS 92*
- ECOOP 87 J.Bézivin, J-M.Hullot, P.Cointe, H.Lieberman (eds.) *ECOOP '87: European Conference on Object-Oriented Programming(1987: Paris, France) Proceedings. (LNCS 276)* Springer-Verlag ISBN 0-387-18353-1 / ISBN 3-540-... QA76.6.E973
- ECOOP 90 → [OOPSLA90]
- ECOOP 91 P America (ed) *ECOOP '91: European Conference on Object-Oriented Programming (Geneva, Switzerland, July 1991) proceedings (LNCS 512)* Springer-Verlag ISBN 0-387-54262-0
- ECOOP92 Ole Lehrman Madsen (ed) *ECOOP '92: European Conference on Object-Oriented Programming (Utrecht, The Netherlands, July 92)* Springer LNCS 615
- Eiffel → [Meyer 88].
- Emerald → [Black 86].
- Fitzgerald89 John Fitzgerald, VDM theory store population review. IPSE Document 060/jsf03/0.1, Sept 89, University of Manchester
- Fitzgerald90 John Fitzgerald, Ph.D. thesis, University of Manchester, 1991.

- FOOL90 JW de Bakker, WP de Roever, G Rozenberg (eds): *Foundations of OO Languages, Netherlands 1990* Springer LNCS 489
- GGH80 J.Guttag, J.J.Horning: *Formal specification as a design tool*. PARC-CSL-80--1
- GHW85 J.V.Guttag, J.J.Horning, J.M.Wing *Larch in Five Easy Pieces* July 1985, Digital Systems Research Center, Palo Alto CA 94301
- Goldberg 83 Adele Goldberg, David Robson: *Smalltalk-80: the language and its implementation*. Addison-Wesley, Reading, MA. 1983.
- GMW79 M.Gordon, R.Milner, C.Wadsworth *Edinburgh LCF* Springer LNCS78, 1979
- Good 82 Don Good, *Proof of a distributed system in Gypsy* TR30, Dept CS, University of Texas at Austin, Sept 82
- Green Cordell Green, *Refine*
- HHG 90 A. Richard Helm, Ian M. Holland, Dipayan Gangopadhyay: Contracts: Specifying behavioural compositions in object-oriented systems. [OOPSLA90]
- HO 87 Daniel C.Halbert, Patrick D.O'Brien (DEC OO, Hudson MA; Trellis): Using types and inheritance in object-oriented languages. in [ECOOP 87] pp 20–31. Guidelines for programmers. Subtyping for specialisation (e.g. window to VDU-window), reification (satisfaction of behaviour defined in supertype), combination of characteristics by multiple inheritance; generalisation (e.g. circle to ellipse), variance. In the last two cases, the conceptual and implementation hierarchies mismatch.
- Hoare90 CAR Hoare, foreword to [VDM'90]
- Hogg91 John Hogg: 'Islands: Aliasing protection in object-oriented languages.' *OOPSLA'91*.
- HW90 Trevor Hopkins, Brian Warboys: Asset management and object-oriented technology *Hotline on Object-Oriented Technology* 1(11) Sept 90 pp12–13
- Ipse "The Ipse2.5 Project" in [Mural91] pp8–9
- Jacobson Ivar Jacobson: *Object-oriented software engineering* Addison-Wesley, 1992
- JF 88 Ralph E Johnson, Brian Foote: Designing reusable classes. *Journal of Object-Oriented Programming* 1(2) June/July 88. pp22–35.
Guidelines. How to make maintainable evolvable software. A 'framework' is a module of partially abstract classes, designed to be specialised by inheritance or run-time parameterisation ('pluggable' classes).
OOP → faster programming; use the time to generalise for re-use.
- Johnson 86 Ralph E Johnson: Type checking in Smalltalk. [OOPSLA 86]. pp315–322.
- JGZ 88 R.E.Johnson, Graver, Zurawski: Typed Smalltalk: an optimising compiler [OOPSLA 88].
A type here is a set of parameterised classes. Part aim is optimisation of code: only possible knowing types at compile time.
- Johnson91 Carl McConnell, Ralph Johnson: Compile-time garbage collection in Typed Smalltalk. Dept CS, U Illinois, 1304 W.Springfield, Urbana, IL 61801
How Typed Smalltalk can be augmented with an *effect system* that will allow the TS compiler to determine when objects may be allocated on the stack rather than the heap, thus reducing and in many cases eliminating the need for run-time garbage collection.

- Jones 86a Cliff Jones: *Systematic software development using VDM*. PHI, 1986. ISBN 0-13-880725-6 QA76.76.D47 J66
- Jones 86b Cliff Jones: ‘Proof Obligations for Data Reification’ in OG.Folberth, C.Hockl (eds) *Der Informationsbegriff in Technik und Wissenschaft* [Oldenbourg, München, 1986] (retr-reln)
- Jones 87 Cliff Jones: VDM Proof obligations and their justification. In [VDM 87] pp260–286.
- Jones92 C.B.Jones: $\pi\alpha\beta\lambda$. In TOOLS workshop 92
Excite your local librarian: ask them to look up this paper.
- Jonkers 88 H.B.M.Jonkers: *Introduction to COLD-K* Dept of Philosophy, University of Utrecht 1988
- Hoare69 C.A.R. Hoare: An axiomatic basis for computer programming’, *Comm. ACM* 12(10), 576–590, 583 (Oct 1969)
- Hogg91 John Hogg. Islands: aliasing protection in OO languages. Tech. note, Bell-Northern Research, Ottawa, Ont.
- Leavens 90 Gary T. Leavens: *Modular verification of object-oriented programs with subtypes* TR #90-09 July 90, Dept of CS, Iowa State University, Ames, Iowa
- Leavens 91 Gary T. Leavens: Modular specification and verification of object-oriented programs. *IEEE Software* July 1991.
- LH92 Kevin Lano, Howard Haughton: Reasoning and refinement in object-oriented specification languages. In [ECOOP 92].
- Lieberherr 88 Lieberherr: Object-oriented programming: an objective sense of style. [OOPSLA 88] pp323–334.
The Law of Demeter — good for modularity and avoiding cross-modular invariant breakage: $\forall c:\text{Classes } C, m:\text{Method} \cdot$ all objects to which m sends a message must be instances of [subclasses of]: argument classes of m ; or instance variable classes of c . (Objects created by m or its calls, and global objects, are allowed too.) As stated above, it can be checked at compile time; but the real ideal is stricter: m should only send messages to: **self** or arguments; instance variable contents; objects created by m or functions it calls; globals.
- Lindsay87 Peter Lindsay: Logical frames for interactive theorem proving. TR: UMCS 87-12-7, University of Manchester Dept of Computer Science, 1987
- Lindsay88 Peter Lindsay: A survey of mechanical support for formal reasoning. *Software Engineering Journal*, 3(1), Jan 1988
- Lindsay91 Natural Deduction for Interactive Theorem Proving: the *mural* proof model
- Meyer 88 Bertrand Meyer: *Object-oriented software construction* PHI. ISBN 0-13-629049-3.
Eiffel exposition. In Eiffel, classes are held to implement ADTs, and the class hierarchy should be the same as the ADT hierarchy; but there is no mechanical checking of constraints on the hierarchy, or of correct reification.
- Minkowitz C.Minkowitz, P.Henderson: A formal description of object-oriented programming using VDM. In [VDM 87], pp 237–259.
- Morgan90 Carroll Morgan: *Programming from specifications* PHI 1990
ISBN 0-13-72633-7
- MP91 C.B.Medeiros & P.Pfeffer ‘Object integrity using rules’ *ECOOP’91*

- Mural91 C.B.Jones, K.D.Jones, P.A.Lindsay, R.C.Moore: *mural: a formal development support system* Springer-Verlag, 1991.
- Nipkow 85 T.Nipkow: ‘Nondeterministic data types: models and implementations’ *Acta Inf*, 1985
- O’Brien 87 P.D.O’Brien, D.C.Halbert, M.F.Kilian. The Trellis programming environment. [*OOPSLA 87*] pp.91–103
- Oliver 88 Huw Oliver: *Formal specification methods for re-usable components* Ph.D. Thesis, University College of Wales, Aberystwyth, 1988.
- OOPSLA 86 *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. Portland, Oregon, 1986. SIGPLAN 21(11) Nov. 86.*
- OOPSLA 87 *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. Orlando, Florida, 1987. SIGPLAN 22(12) Dec. 87.*
- OOPSLA 88 *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. 1988.*
- OOPSLA90 *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. 1990. in SIGPLAN Notices*
- Prawitz 71 D.Prawitz. *Natural Deduction*. Proc. 2nd Scandinavian Logic Symp. 1971.
- Refine → [Green 68]
A medium-spectrum applicative language and transformational system. Quantifiers are limited to finite ranges; all programs are executable — some more efficient than others. ML-like type inference. System finds matches for transformations, offers choice for user selection.
- RTD83 Reps, Teitelbaum, Demers. Incremental context-dependent analysis for language-based editors. *ToPLaS* April 83.
Structure-editors with attributed grammars and action routines, were, with active databases, and ADTs, part of the intellectual input to the development of object-oriented ideas. The principles of constraint maintenance developed for them are adaptable to OOP.
- Rumbaugh James Rumbaugh et al, *Object-Oriented Modeling and Design* [PHI 91] ISBN 0-13-630054-5
- Schaffert 86 Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, Carrie Wilpolt. An introduction to Trellis/Owl. [*OOPSLA 86*] pp. 9–16.
- Smalltalk → [Goldberg 83], [Borning 87].
- Spivey J.M.Spivey *The Z Specification Language* PHI
- Stepney 91 Susan Stepney, ed., *Comparison of Object-Oriented Specification Methods*, Logica, Cambridge, UK, 1991
- Snyder 86 Alan Snyder: Encapsulation and inheritance in object-oriented programming languages. [*OOPSLA 86*] pp. 38–45.
Encapsulation and inheritance are antagonists. Even within a class, don’t access instance variables directly (or your language should have interchangeable ‘features’, like Eiffel and CLU). The visible/hidden op-set should be separately defined for clients and inheritors, as in Trellis/Owl. Subtyping shouldn’t be related to inheritance: if it is, you can’t divorce the implementation of a child

from that of its parent; so don't use subclasses for specialisation, but just for re-use of code. Use of inheritance by an implementor should be an alterable, private decision.

- Sørensen 91 I. Sørensen, *The B Method VDM 91* Noodwijkerhout, The Netherlands, 1991.
- SP91 Jen Palsberg, Michael Schwartzbach: What is type-safe code reuse? -ECOOP 91
- Szyperski92 Clemens Szyperski: Import is not inheritance — why we need both: modules and classes [ECOOP'92]
- Trellis → [Schaffert 86], [O'Brien 87]
Trellis is the environment, Owl is the language. DEC. Static multiple inheritance; proper behavioural type hierarchy. Parameterised types. Working towards a multi-user system.
- UR91a Mark Utting, Ken Robinson: Towards an object-oriented refinement calculus. In *Proc. 14th Australian Comp Sc Conf, 1991*.
- UR91b Mark Utting, Ken Robinson: The object-oriented lollipop: an example of subtyping. In *Proc. 6th Australian Software Engineering Conf. 1991*, pp355–368.
- UR92 Mark Utting, Ken Robinson: Modular reasoning in an object-oriented refinement calculus. TR, Software Verification Research Centre, U. Queensland 1992
- Wegner 87 P Wegner, A Brown: Dimensions of object-oriented design. [OOPSLA 87] p168–182.
Essentials of OOD: objects, classes, inheritance.
Dimensions (more or less orthogonal): objects; types (sorts of expressions); delegation; abstraction (interface specs); concurrency; persistence.
Defines a lot of terms; relates actual OOP features to these dimensions.
- Wegner 90 P Wegner, *Concepts and Paradigms of Object-Oriented Programming OOPS Messenger* 1(1) Aug.90 pp7–87
- Wills 91a Alan Wills, Capsules and Types in Fresco: Smalltalk meets VDM [ECOOP 91] pp59–76
- Wills 91b Alan Wills, *Object Oriented Software Engineering* participants' notes for a course. Aug.91.
Unconnected with Darrel Ince's and Ivar Jacobson's books of the same name. Covers analysis and design using graphical notation with a formal component.
- Wills92a Alan Wills: 'Specification in Fresco' in Susan Stepney (ed): *Comparison of Object-oriented Specification Techniques*, Addison-Wesley 1992
- Wills92b Alan Wills: 'Refinement in Fresco' in Kevin Lano (ed): *Case Studies in Object-oriented Refinement* PHI 1992
- Wolczko 87 Mario Wolczko: Semantics of Smalltalk-80. [ECOOP 87] pp 108–120
- WW90 Alan Wills, Mario Wolczko (administrators): goodies-lib@cs.man.ac.uk.
A mail-server distributing public-domain Smalltalk software, with contributions from all over the planet.
- WWC90 R. Wirfs-Brock et al *Designing Object-Oriented Software* 1990
- VDM 87 *VDM — a formal method at work*. Ed. D.Bjørner, C.B.Jones, M.Mac an Airchin-nigh, E.J.Neuhold. LNCS 252, Springer-Verlag. ISBN 0-387-17654-3 / ISBN 3-540-17654-3
- VDM90 D Bjørner, CAR Hoare, H Langmaack (eds) *VDM 90: VDM and Z — Formal*

methods in software development LNCS 428, Springer 1990

VDM91 S.Prehn & WJ Tötenel *VDM'91 — Formal Software Development Methods. Proc 4th Int Symp of VDM Europe, Noordwijkerhout, The Netherlands, Oct 91. Vol 1: Conference Contributions.* [Springer-Verlag 1991] LNCS 551

Appendices

10-1 Fresco development language FST

The syntax of the executable parts has been somewhat modified from that of Smalltalk, to accommodate the extra specification constructions. The ability to refer to the vector of parameters to an operation is useful, and so traditional argument-list syntax has been introduced.

10-1.1 Expressions

Expr ::= MonadicPrefix | DyadicExpr | OpExpr | KeywdExpr
 | Variable | Constant | '(' Expr ')' | Block | Assignment
 | SetExprn | BoundExprn | Metavariable | SpecStmt

Expressions differ from Smalltalk in providing: unary operators; a syntactic precedence for operators; parenthesised argument lists; and specialised syntaxes for quantified predicates and members of the type **Set**. All of these are syntactic sugar, and can be translated to conventional Smalltalk syntax. In addition, there is provision for metavariables which are used within theorems to stand for subexpressions.

10-1.1.1 Conventional expressions

MonadicPrefix ::= UnaryOp Expr
 DyadicExpr ::= Expr BinOp Expr
 OpExpr ::= Expr OpName ['(Expr [' , ...] ')']

The names of binary and unary operators are constructed from nonalphanumeric symbols; OpNames are constructed from alphanumeric symbols. There are predefined binary and unary operators, and users may define new ones.

The predefined binary operators have a conventional syntactic binding precedence, and bind left-to-right.

| × / ∩
 + − ∪
 ∈ ⊆ < > ≤ ≥ = ~ = == ~~
 ^
 /
 ⇒ ⇔

Unary operators bind tighter than binary; binary bind tighter than OpExprs, which bind left-to-right: $a\ b(c)\ d(e) \equiv (a\ b(c))\ d(e)$. OpExprs need no parentheses if there are no arguments beyond the first, ‘receiver’, argument.

(Once defined, the unary and binary operators have a fixed precedence within a capsule and its importers; but the operations they represent may vary from class to class. For example, I could define a new type **T** with operators **+** and **<**, and they

would have new meanings relative to T , but their syntactic precedence cannot be changed. However, the same symbol can be defined as an operator in different capsules, with different precedence: the parsing depends on the capsule you're working in. Where there is a conflict between the imports of a new capsule, the designer must redefine the precedences in the new capsule.)

10-1.1.2 Smalltalk-style expressions

$\text{KeywdExpr} ::= \text{Expr sel1: Expr [sel2: Expr [sel3: Expr [...]]]$

This is an invocation of one parameterised operation called sel1:sel2:sel3: .

KeywdExprs bind less tightly than OpExprs .

There is no real difference between operations declared and used in the KeywdExpr style and in the OpExpr style: the former is standard Smalltalk, and works well with program-constructing expressions (ifTrue:ifFalse: etc), and the latter is more convenient where the operation name is a metavariable in a theorem.

10-1.1.3 Assignment

$\text{Assignment} ::= \text{Variable} \text{'=' Expr}$

The variable now refers to the object yielded by the expression.

10-1.1.4 Creation functions

$\text{ClassName a:...b:...}$

10-1.1.5 Special notations

$x == y$	The expressions X and Y refer to the identical object.
$x = y$	The expressions refer to objects which represent equal values, where equality is determined by their type(s).
$\{x T, y T, \dots\}$	A member of Set containing objects equal (wrt type T) to x , or y , or ..., and not containing objects not equal to any of these.
$\{x, y, \dots\}$	A member of Set containing the objects referred to by x, y, \dots
$x \in S$	If S is a member of Set , the object referred to by x belongs to S .
$x T \in S$	If S is a member of Set , then $\exists y \cdot y T = x T \wedge y \in S$
$\{x \in T \cdot P[x]\}$	This is a member of Set containing a representative object of every equality-class whose members satisfy $P[x]$: $\forall y \cdot P[y] \Rightarrow y \in \{x \in T \cdot P[x]\}$
$x \in T$	If T is a type, then x conforms to the axioms defining T .
$\forall x \in T \cdot P[x]$	P holds for all states of all possible objects in all possible systems satisfying the assumptions of the current context.
$\exists x \in T \cdot P[x]$	P holds for some state of some possible object in <i>all</i> possible systems <u>satisfying the assumptions of the current context</u> .

10-1.2 Specifications

$\text{Spec} ::= \text{'{' Expr ':' Expr '}' | '}' \text{ inv '}' | ('Spec-stmt '[' , ...]')}$

A specification can prefixing a piece of code asserts that the code conforms to the specification:

SpecStmt ::= Spec ['/' Justification '/'] Statement

In addition, a **SpecStmt** is executable, being equivalent to its **Statement** or **Expr** for that purpose. The optional **Justification** is the label of a [justified] theorem which proves the conformance.

10-1.2.1 Pre/post

In

$\{ \text{pre} :- \text{post} \} S$

if **pre** is true before executing **S**, then **S** is a **Statement** or **Expr** that will

- terminate and leave the system in a state conforming to relevant invariants (for example, type invariants if this statement forms part of a type-description);
- leave the system in a state such that **post** would evaluate to true.

post may contain barred expressions, and the special variable ' \uparrow '; **pre** may not.

10-1.2.2 Code-Invariant

A code-invariant stipulates that *if* the assertion is true beforehand, it will be true after:

$\{ \text{inv} \} S \equiv \{ \text{pre} :- \text{post} \} S$

10-1.2.3 Composition

More than one **Spec-stmt** may be applied to any statement or expression:

$(\{ \text{pre1} :- \text{post1} \} , \{ \text{pre2} :- \text{post2} \} , \dots) S$

in which case all of them apply individually to **S**.

10-1.2.4 Inline justification

A theorem or proof may be included in with the code:

JustifiedSpecStmt ::= **SpecStmt**
 | Spec '(' JustifiedTheorem ['...',...] Statement ')'

Most of the useful decomposition proofs can be written this way.

10-1.3 Code

A method definition associates a piece of executable code with a particular type and operation name. If an object **x** **class** = **T**, and if **T::op** = **S[self, p_i]**, then the evaluation of **x op (E_i)** will be the evaluation of **S(x, E_i)**. **T::op** is the code explicitly prescribed if there is any, or the same as **TT::op** if **TT** is a *superclass* of **T**.

A method definition may be written:

MethodDefn ::=
 metavars · Type '::' OpName '(' params ')' ':= ' '(' Sequence ')'

A sequence is a group of statements, possibly with local variables:

Sequence ::= ['var' localVar [...]' '] Statement [';' ...]
 Statement ::= '(' Sequence ')' | Expr | JustifiedSpecStmt

A block is a value representing a sequence of statements, possibly with parameters and local variables:

Block ::= '[' [params ' '] Sequence ']'

Executing a Block yields an object representing the code in the block; executing a sequence executes the statements in the sequence. To execute the sequence inside a Block, an appropriate operation must be used.

10-1.3.1 Control expressions

Blocks form the basis of control structures. Used in a limited fashion, they present no particular problems of validation. We use these basic Smalltalk control structures, where C_i are expressions yielding members of Boolean, E_i are expressions, and S_i are Statements:

C ifTrue: [S1] ifFalse: [S2]
 [S1 ; C] whileTrue: [S2]

10-1.4 Metavariables

These are used within theorems to stand for variables, expressions, operation-names, lists of variables or expressions, or statements. Metavariables are distinguished from other variables by their declaration in the bindings of theorems or proofs.

E	(capital initial letter) stands for an expression or statement
v	(small letters) stands for an op-name, variable or parameter
E_i, v_i	(unbound subscript) stands for a list of items, of which any particular one may subsequently be referred to as v_1 etc (literal or bound subscript)

When a theorem is applied, operation-name metavariables match with ‘conventional’ operations, binary or unary operators, or Smalltalk-style keyword expressions: for example,

a, op, b · a op(b)

matches all of:

x f (y), x add: y, 2+x.

whilst

a, op, b_i · a op(b_i)

matches all of

x f(y), x ff(y, z), x floor, -x, x+y

Theorems about variables can be applied to expressions, provided the expressions are pure (relative, at least, to the expressions of the theorem) (see §8):

pure-thm: $A, B \cdot (a_i \cdot A(a_i) \vdash B(a_i)) \vdash (E_i \cdot (i \cdot E_i \Delta \emptyset), A(E_i) \vdash B(E_i))$

10-1.4.1 Qualified and modified metavariables

Expression-metavariables may be qualified:

$M[x]$

The qualifier may be any expression, including another metavariable. A theorem containing metavariables matches with an expression only if there is a consistent assignment of subexpressions to metavariables.

For example, in an induction rule:

$P \cdot P[0], (i \cdot i \in \text{Nat}, P[i] \vdash P[i+1]) \vdash (j \cdot j \in \text{Nat} \vdash P[j])$

P matches any proposition; i and j match variables or pure expressions; 0 , Nat , $+$ and \in are constants and operation-names defined in the context.

Expression-metavariables may be modified:

$M[x \setminus y]$

If M matches some expression containing occurrences of x , then $M[x \setminus y]$ is the same expression with each occurrence of x replaced with y .

10-1.5 Theorems

Assertion ::= Exprn | Spec-Stmt | Theorem

Theorem ::= [label ':'] [metavar ['...','] ':']
[Theorem ['...',']
'⊢' Assertion ['...',']

The initial set of **metavar** names binds those names to the scope of the theorem.

Syntactic precedence rules: any expression-construction $>$, $'>$ $>$ $'\vdash'$; and the latter associates left-to-right:

$a, b \vdash c \vdash d, e \equiv ((a, b) \vdash c) \vdash (d, e)$

Multiple conclusions just mean that each of them can be inferred separately:

$A_i \vdash C_i \equiv A_i \vdash C_1, A_i \vdash C_2, \dots$

A convenient alternative syntax substitutes horizontal lines for \vdash :

pure-opspec:
$$P, R, E \cdot \frac{e \Delta \emptyset \quad \{ P :- R[\uparrow] \} E}{P \vdash R[E]}$$

10-1.5.1 Proofs

A proof is a theorem documented with intermediate theorems from which the final assertion follows:

```

Proof ::=
    [ label ':' ] [ metavar ['...'] ':' ]
    [ Assertion ['...']
    [JustifiedTheorem ['...']]
    '⊢' JustifiedAssertion ['...']

```

An intermediate theorem may be the nested proof of a theorem which can be proven within the context of the containing hypotheses; or it may be an assertion justified as a match to the conclusion of some rule:

```
JustifiedTheorem ::= Proof | JustifiedAssertion
```

A justified assertion is documented with the label of the rule whose conclusion it matches, together with the antecedents which match the rule's hypotheses. (There are also other styles of justification.)

```
JustifiedAssertion ::= Theorem 'by' Justification 'from' label ['...']
```

10-1.6 Types and classes

```
TypeDef ::=
```

Name ['(' TypeExprn ')'] ['::+' Type [, ...] [
'op' name '∈' OpSignautre [...] opspec [...]½ [
'var' 'const' name '∈' TypeExprn [...] invarriant]]

Methods may be attached to a TypeDef:

```
MethodDef ::= TypeName '::' OpName '=' SpecStatment
```

10-2 Fresco kernel types

10-2.1 Sets

The conventional set operators have their usual meanings, and are defined as pure functions, so that they may be used in assertions or code:

$\cap \cup -$

$(s1 \cap s2) \cup (s1 - s2) = s1$

s card The size of set s.

Set of: T A set whose members must be subtypes of T

Set of: T		
\dot{y}	: Set of: T	/* creation func */
(Set of:T): (T)	Set of: T	/* creation func */
$_ \gg _$: (Set of: T) Set of: T	
$_ \ll _$: (Set of: T) Set of: T	
$_ \subseteq _$: (^, _) Bool	
card	: Integer	
$\emptyset \in$	(Set of: T)	
$\forall x \cdot$	$\neg x \in \emptyset$	
$\forall x:T, y:T \cdot$	$x \in (\text{Set of: T})(x) \wedge (x \neq y \Rightarrow y \notin (\text{Set of: T})(x))$	
$\forall x \cdot$	$x \in S1 \vee x \in S2 \Leftrightarrow x \in S1 \cup S2$	
$\forall x \cdot$	$x \in S1 \wedge x \in S2 \Leftrightarrow x \in S1 \cap S2$	
\emptyset	card = 0	
(Set of: T)	(x) = 1	
$\forall s1, s2 \cdot$	$s1 \text{ card} + s2 \text{ card} = (s1 \cup s2) \text{ card}$	

10-2.2 Lists

An IdList keeps references to objects:

IdList(T)		
op $_ ++ _$	\in (IdList(T))	$\Delta \{\text{self}.\delta\}$
fn $_ @ _$	\in (Nat) \rightarrow T	$\Delta \emptyset$
var length	\in Int	
var @	\in Nat \rightarrow T	
$\{ \text{:- } \uparrow.\text{length} = \text{self}.\text{length} + x.\text{length} \wedge (\forall i \in 1..\text{self}.\text{length} \Rightarrow \uparrow @ i == \text{self} @ i) \wedge$ $\quad \forall i \in \text{length} + 1..\uparrow.\text{length} \cdot \uparrow @ i == x @ (i + \text{length}) \} \text{self} ++ t$		
$\{ i \in 1..\text{length} \text{ :- } \uparrow == @ (i) \} \text{self} @ i$		
fn dem = {length, {@(i) i · i ∈ 1..length}}		

$\{ x \in T \text{ :- } \uparrow \in \text{Idlist}(T) \wedge \uparrow.\text{length} = 1 \wedge \uparrow @ 1 == x \} \text{IdList}(T).\text{mk}(x:T)$

10-2.3 Maps

Map from: S to: T	
\emptyset	Map from: S to: T
$_ \rightarrow _$	(S, T) Map from: S to: T
$_ @ _$	(S) T
dom	Set of: S
rng	Set of: T
$_ \uparrow _$	(Map from: S to: T) Map from: S to: T
=	(Map from: S to: T) Boolean
$s \in \text{self dom} \vdash \text{self}@s \in \text{self rng}$	
$(s \rightarrow t) \text{ dom} = \text{Set}(s)$	
$(s \rightarrow t) @ s = t$	
$m \in (\text{Map from: S to: T}) \vdash \text{self} \uparrow m \text{ dom} = \text{self dom} \cup m \text{ dom}$	
$\wedge (\forall s \cdot s \in m \text{ dom} \Rightarrow (\text{self} \uparrow m) @ s = m @ s)$	
$\wedge (\forall s \cdot s \notin m \text{ dom} \Rightarrow (\text{self} \uparrow m) @ s = \text{self} @ s)$	
$xx = \text{self} \Leftrightarrow xx \text{ dom} = \text{self dom} \wedge$	
$\wedge (\forall s \cdot s \in \text{dom} \Rightarrow xx @ s = \text{self} @ s)$	

10-3 Kernel proof rules

$a \cdot a \in A, (x \cdot x \in A \vdash P[x]) \vdash P[a]$

10-3.1 Comparison with conventional LPF

The Mural standard complement of theories defines these fundamental theories:

- *Propositional LPF*, declares symbols **true**, \neg , \vee , and defines in terms of them \Leftrightarrow , \Rightarrow , **false**, and \wedge . LPF is designed to deal with the possibility of the falsity of the conventional axiom $e1 \vee \neg e2$.

The other axioms follow conventional logic:

contradiction:	$e1, e2 \cdot$	$e1, \neg e1 \vdash e2$
true-intro:		true
$\neg \vee$ -elim-l:	$e1, e2 \cdot$	$\neg(e1 \vee e2) \vdash \neg e2$
$\neg \vee$ -elim-r:	$e1, e2 \cdot$	$\neg(e1 \vee e2) \vdash \neg e1$
$\neg \vee$ -intro:	$e1, e2 \cdot$	$\neg e1, \neg e2 \vdash \neg(e1 \vee e2)$
$\neg \neg$ -elim:	$e \cdot$	$\neg \neg e \vdash e$
$\neg \neg$ -intro:	$e \cdot$	$e \vdash \neg \neg e$
\vee -elim:	$e1, e2, e \cdot$	$e1 \vee e2, (e1 \vdash e), (e2 \vdash e) \vdash e$
\vee -intro-l:	$e1, e2 \cdot$	$e2 \vdash e1 \vee e2$
\vee -intro-r:	$e1, e2 \cdot$	$e1 \vdash e1 \vee e2$

- *Equality and typing*, inherits from propositional LPF, and declares symbols $=$ and \in ; \neq is defined in terms of \neg and $=$. These axioms are defined:

=-comm:	$s1, s2 \cdot$	$s1 = s2 \vdash s2 = s1$
---------	----------------	--------------------------

$\neq\text{-comm}$:	$s1, s2 \cdot$	$s1 \neq s2 \vdash s2 \neq s1$
$=\text{-contr}$:	$s, e \cdot$	$s \neq s \vdash e$
$=\text{-subs}$:	$E, T, s1, s2 \cdot$	$E[s1 T], s1=s2 \vdash E[s2 T]$
$\text{id}=\text{:}$:	$a, b, T \cdot$	$a == b, a \in T \vdash a T = b T$ — neither barred
$\neq\text{---}$:	$a, b, T \cdot$	$a A \neq b B \vdash \neg(a==b)$
$\text{antimono}=\text{:}$:	$a, b, T, TT \cdot$	$a \in T, b \in T, T \subseteq TT, a T = b T \vdash a TT = b TT$
$\text{mono}\neq\text{:}$:	$a, b, S, T \cdot$	$a \in T, b \in T, S \subseteq T, a T \neq b T \vdash a S \neq b S$
uneq-types-l :	$a, b, A, B \cdot$	$a \in A, b \in B, A \cap B = \emptyset \vdash a A \neq b A$

10-3.2 Projection to a type

$|- \text{defn:}$ $x \in T, y \in T \vdash (x|T \in T \wedge x|T = y|T \equiv \bigwedge f \cdot x f = y f)$
for all functions f defined for T .

10-3.3 Opspecs

code-inv-defn :	$\text{inv}, \text{Code} \cdot$	$\{ \text{inv} :- \text{inv} \} \text{Code} \equiv \{ \text{inv} \} \text{Code}$
stren :	$P, P1, R1, R, S \cdot$	$(P \vdash P1), (\bar{P}, R1 \vdash R) \vdash \{ P :- R \} \{ P1 :- R1 \} S$
stren :	$P, R, P1, R1, S \cdot$	$\{ P :- R \} \{ (P \vdash P1), (\bar{P}, R1 \vdash R), \{ P1 :- R1 \} S \}$
seq :	$P, M_i, x_i, S_j \cdot$	$\{ P :- M_n[x_j] \} x_{0j} \cdot \{ P :- M_i[x_j] \} S_1; \{ M_{i-1}[x_{0j}] :- M_i[x_{0j}] \} S_i; \dots$
if :	$P, R, C, S1, S2 \cdot$	$\{ P :- R \} (C \text{ ifTrue: } \{ P \wedge C :- R \} S1 \text{ ifFalse: } \{ P \wedge \neg C :- R \} S2)$
loop-se :	$\text{inv}, C, v, S \cdot$	$\{ \text{inv}[x] :- R[\bar{x}] \}$ [$\{ \text{inv}[x_0] \wedge \neg PT :- R[x_0] \wedge \uparrow = \text{false} \},$ $\{ \text{inv}[x_0] \wedge PT :- M[\bar{v}, x_0] \wedge \uparrow = \text{true} \}) C]$ $\text{whileTrue: } \{ \{ v \in \text{Int} \wedge M[v_0, x_0] :- \text{inv}[x_0] \wedge 0 \leq v \wedge v < v_0 \} S \}$
assignment :		$\{ P :- \exists x_0 \cdot R[x, x_0] \wedge \uparrow == x \} x := \{ P :- R[\uparrow, x] \} E$
assignment' :		$\{ P :- R \wedge \uparrow == x \} x := \{ P :- R \} E[x]$
yield :	$P, Q, x, \text{op} \cdot$	$\{ P :- Q[\uparrow] \} x \text{ op } \vdash Q[x \text{ op}]$
var-exprn :	$x, y \cdot$	$\{ :- x = \bar{x} \wedge \uparrow == x \wedge y = \bar{y} \} x$

$$\text{use-pure: } G, F, R, a, S \cdot \frac{\begin{array}{c} F[a] \Delta \emptyset \\ a \cdot \{ P[a] :- R[\uparrow, a] \} F[a] \\ \{ \Delta s \cdot PS :- P[a] \wedge G[F[a]] \} S \end{array}}{\{ \Delta s \cdot PS :- \exists n \cdot G[n] \wedge R[n, \bar{a}] \} S}$$

— must be applied separately for each occurrence of $F[a]$ in G .

$$\text{promote: } P, R, S \cdot \frac{v_i \cdot \{ P[v_i] :- R[v_i, v_i] \} S[v_i]}{x_i \cdot \llbracket S[x_i] \rrbracket \Rightarrow (P[x_i] \Rightarrow R[\bar{x}_i, x_i])}$$

conjoin :	$\{ PA \vee PB :- (PA \Rightarrow RA) \wedge (PB \Rightarrow RB) \} (SA \mid \wedge \mid SB)$
disjoin :	$\{ PA \wedge PB :- RA \vee RB \} (SA \mid \vee \mid SB)$
intersect :	$\{ PA \wedge PB :- RA \wedge RB \} (SA \mid * \mid SB)$
fallback :	$\{ PB :- (RA \neq RB) \wedge (\neg PA \Rightarrow RB) \} (SA \mid / \mid SB)$
seq :	$\{ PA :- \exists s' \cdot RA(s, s') \wedge RB(s', s) \} (SA ; SB)$

10-3.4 Types

- The *theory* of a Fresco type T is a set of theorems $A_{T,i}$ over a set of message-selectors.
- An object x is a member of type T , written $x \in T$, iff all the theorems of the theory of T are valid when x is substituted for **self** (and after making **self** explicit as a prefix to attribute names — **self.x** rather than just x):

$$\frac{\begin{array}{c} A_{T,1}[\text{self} \backslash x] \\ \dots \\ A_{T,n}[\text{self} \backslash x] \end{array}}{x \in T}$$

- A type S is a *subtype* of a type T , written $S \subseteq T$, iff every member x of S is also a member of T .

$$\frac{x \in S \vdash x \in T}{S \subseteq T}$$

- $x \in T_1 \cap T_2 \Leftrightarrow x \in T_1 \wedge x \in T_2$

T-impl:

$$x \text{ class} = T \vdash x \in T$$

basicNew:

$$C \cdot C \text{ basicNew class} = C$$

10-3.5 Effects

fx-indep: $r, w, e, S \cdot \nabla r \cdot e, \Delta w \cdot S, r \text{ sep } w \vdash \{e\} S$

fx-indep-stmt: $r, w, e, S_1, S_2 \cdot \Delta w \cdot S_1, \nabla r \cdot S_2, r \text{ sep } w \vdash \{P:-R\} (S_1; \{P:-R\} S_2)$

fx-conjoin: $d_1, d_2, S \cdot \Diamond d_1 \cdot S, \Diamond d_2 \cdot S \vdash \Diamond d_1 \cap d_2 \cdot S$

fx-expand: $sd, d, S \cdot sd \subseteq d, \Diamond sd \cdot S \vdash \Diamond d \cdot S$

sep-comm: $z_1, z_2 \cdot z_1 \text{ sep } z_2$

sep-preserve: $z_1, z_2, S, r, w \cdot \nabla r \cdot S, \Delta w \cdot S, (P \vdash w \text{ sep } z_1 \vee r \text{ sep } z_2), (P \vdash r \text{ sep } z_1 \vee w \text{ sep } z_2) \vdash \{P \wedge z_1 \text{ sep } z_2 :- z_1 \text{ sep } z_2\} S$

isol-defn: $z.\text{isolated} \equiv \forall x \cdot \neg(\uparrow == x) \Rightarrow z \text{ sep } x$

create-sep: $\{ :- \uparrow.\text{isolated} \} \text{ AnyClass basicNew}$

indep-sep: $\Delta \emptyset \nabla \emptyset \cdot S \vdash \{ :- \uparrow.\text{isolated} \} S$

10-3.6 Frames

fx-const: $\Diamond \emptyset \cdot (c)$ — c a constant

fx-var-r: $\nabla \&v \cdot (v)$

fx-var-w: $\Delta \emptyset \cdot (v)$

fx-op:	$\frac{\begin{array}{c} \Diamond d_{op}[p_i] \cdot (p_0 \text{ op } (p_{i>0})) \\ \Diamond d_i \cdot e_i \end{array}}{\Diamond d_{op}[e_i] \cup \bigcup_i d_i \cdot e_0 \text{ op } (e_{i>0})} \quad \text{--- } e_i \text{ pure}$
pure-form:	$\frac{\begin{array}{c} \Delta \emptyset \cdot E \\ \Delta \emptyset \cdot R[v] \end{array}}{\Delta \emptyset \cdot R[E]}$
transp:	$\frac{\begin{array}{c} f \text{ transparent} \\ \nabla \rho_i \cdot E_i \end{array}}{\nabla E_i \cdot \delta \cup \bigcup_i \rho_i \cdot E_0.f(E_i)}$
fx-id:	$\frac{\begin{array}{c} \Diamond d_1 \cdot E_1 \\ \Diamond d_2 \cdot E_2 \end{array}}{\Diamond d_1, d_2 \cdot (E_1 == E_2)}$
pure-opspec:	$\frac{\begin{array}{c} E \Delta \emptyset \\ \{ P :- R[\uparrow] \} E \end{array}}{P \vdash R[E]}$
fx-seq:	$\frac{\Diamond d_i \cdot E_i}{\Diamond \bigcup_i d_i \cdot (E_1; E_2; \dots)}$
fx-ass-r:	$\frac{\nabla \rho \cdot E}{\nabla \rho[v\bar{v}] \cdot (v := E)}$
fx-ass-w:	$\frac{\Delta \omega \cdot E}{\Delta d[v\bar{v}] \cup \{ \&v \} \cdot (v := E)}$
fx-var-elim:	$\frac{\Diamond \{d, \&v\} \cdot E}{\Diamond d \cdot (\text{var } v \cdot E)} \quad \text{--- } v \text{ does not occur free in } d$
fx-var-elim-iso:	$\frac{\begin{array}{c} \{ :- v \text{ isolated} \} E \\ \Diamond \{d, \&v, v\} \cdot E \end{array}}{\Diamond d \cdot (\text{var } v \cdot E)}$

$$\begin{array}{c}
\text{fx-subs-==:} \qquad \qquad \qquad \frac{\begin{array}{c} \Diamond E1 \cdot S \\ E1 == E2 \end{array}}{\Diamond E2 \cdot S} \\
\\
\text{fx-copy-nop:} \qquad \qquad \qquad \frac{\begin{array}{c} \Delta d \cdot E \\ \Delta x.\delta \cdot x, p_i \cdot (x \text{ op}) \end{array}}{\Delta d \cdot E.\text{deepCopy.op}}
\end{array}$$

10-3.7 Barred variables

$$\begin{array}{ll}
\text{bar-invar:} & \text{pre, } S \cdot \{ \text{pre} : - \overline{\text{pre}} \} S \\
\text{unbar-const:} & \overline{c} \equiv c \qquad \text{--- iff } c \text{ is a constant or metavariable} \\
\text{unbar-transp:} & \overline{E0.f(Ei)} \equiv \overline{E0}.f(\overline{Ei}) \qquad \text{--- iff } f \text{ is pure \& transparent} \\
\text{unbar-binding:} & \overline{\forall x \cdot E[x]} \equiv \forall x \cdot \overline{E[x]} \qquad \text{--- \& same for other binders}
\end{array}$$

10-3.8 Projection

$$\begin{array}{ll}
|- \text{defn:} & x \in T, y \in T \vdash (x|T \in T \wedge x|T = y|T \equiv \bigwedge f \cdot x f = y f) \\
& \text{for all functions } f \text{ defined for } T. \\
\text{eq-T:} & a, b \cdot a \in T, b \in T \vdash a|T = b|T \Leftrightarrow \bigwedge_i a x_i | T_i = b x_i | T_i \\
& a =_T b \text{ to abbreviate } \bigwedge_i a x_i | T_i = b x_i | T_i \\
\text{neq-T:} & a, b \cdot a \in A, b \in B, A \cap B = \emptyset \vdash a|A \neq b|A \wedge a|B \neq b|B \\
\text{subs-eq:} & a, b, P, T \cdot a \in T, b \in T, P[a|T], a|T = b|T \vdash P[b|T] \\
\text{subs-id:} & a, b, P \cdot P[a], a == b \vdash P[b] \qquad \text{from subs-eq, eq-}\bot
\end{array}$$

+ 22

A

a 37

Aliasing 43, 130, 138

arity 59

Assertion 169

Assertions 52

Assignment 77

attributes 37

B

Barred expression 149

binding 60

Block 168

block 60

Box 91

C

Callback 128

capsule 30, 110

capsules 16, 123

Certification 118

certification 116

Class 37

class 28

classes 106, 170

client 38

Code 70, 167

Code invariant 72

Code spec 72

Component 136

Composition 167

Concurrency 43

Conflict 118

conformance 116

Constraint 125

context 67, 88

Creation 106, 122

D

Decomposition 73

decomposition 29

deficits 118

Demesnes 135

E

Effects 133, 148, 174

Eiffel 42

Encapsulation 44

encapsulation 38

Equality 152

- expression 59
- Expressions 165
- extension 22
- F
- Fields 134
- frame 133, 141
- Frames 134, 174
- framework 126, 150
- Functions 90
- G
- Generic 25, 94
- Global 120
- H
- holopraxis 29
- I
- identity 43
- Implementability 100
- incorporation 116
- Inheritance 37, 47
- Inline 76
- Invariant 167
- invariant 139
- invariants 89
- invocation 78
- Isolation 147
- isolation 138
- J
- justification 63
- K
- Kernel 172
- kernel 170
- L
- Lists 171
- Loop 75
- LPF 37, 154
- M
- Maps 172
- Matching 64
- message 37
- Metavariables 150, 168
- metavariables 62
- methods 37
- model 22, 89
- Monotonicity 47, 145
- monotonicity 85
- Mural 12, 68
- N
- Natural Deduction 35

O
Object 37
Object Histories 84
Object-Z 40
OOZE 40
opspec 20, 72
Opspecs 173
Oracles 66
P
Polymorphism 39
POOL 42, 128
Preconditions 23
product 86
Projection 152, 176
Promotion 80
proof 63
Proof expectations 116
Property 90
protocol 126
Q
Qualified 169
R
Re 10
Reference 136
refinement 25
Reification 102, 147
Renaming 119
S
Semantics 55
Separation 137
Sequence 74
Sets 170
Signatures 91
signatures 137
Specifications 166
Statements 70
Strengthening 73
subclass 37
Subranges 48
Substitution 62, 144, 153
subsystems 147
Subtypes 85
Subtyping 98
Symbol 58
System 110
T
tactic 36
TCD 19, 107
Theorem 169

theorem 61
theory 57
transaction type 128
Transformational 35
type 20, 38
Type Box 91
Type theories 83
Types 170, 174
V
VDM 34
Z
Z 34