An Object-oriented Formal Notation: Executable Specifications in *Clay*

(Una notación formal orientada a objetos: especificaciones ejecutables con *Clay*)

Versión preliminar para el depósito de la tesis en el plan “Lenguajes y sistemas informáticos e ingeniería de software”

**Autor:** Ángel Herranz Nieva  
**Director:** Julio Mariño Carballo

25 de noviembre de 2010
Contents

Prefacio (Preface in Spanish) vii

Resumen (Summary in Spanish) xi

Preface xxiii

Summary xxix

I Introduction 1

1 Introduction 3

1.1 Software Engineering .................................................. 3

1.2 Formal Methods in Software Engineering .......................... 5

1.2.1 Formal Methods and Open Problems ............................ 6

1.3 Lightweight Formal Methods ......................................... 9

1.4 Executable Prototypes and Logic Programming .................. 10

1.5 Our Proposal ............................................................ 11

1.6 Structure of the Book .................................................. 12

2 Clay 13

2.1 Classes and Objects ................................................... 13

2.2 Inheritance ............................................................. 15

2.3 Case Classes .......................................................... 17
Modelling Design Patterns in Clay

9.1 Introduction

9.2 Background

9.2.1 Modelling object oriented specifications

9.2.2 Other formalizations of design patterns

9.3 Design patterns as class operations

9.3.1 Composite pattern

9.3.2 Decorator pattern

9.3.3 Different modelling possibilities

9.3.4 Design patterns composition

9.3.5 Application: reasoning with design patterns

9.3.6 Application: design patterns in a development environment

9.4 Future Trends

9.5 Conclusion

9.6 Formalisation of DP in SLAM-SL

9.6.1 Abstract Factory (Figure 9.5)

9.6.2 Bridge (Figure 9.6)

9.6.3 Strategy (Figure 9.7)

9.6.4 Adapter (Figure 9.8)

9.6.5 Observer (Figure 9.9)

9.6.6 Template Method (Figure 9.10)

9.6.7 Decorator (Figure 9.11)

9.6.8 State (Figure 9.12)

9.6.9 Builder (Figure 9.13)

V Conclusion

10 Conclusions

10.1 The Design of a Formal Object Oriented Notation

10.2 Non-structural Type System Definition
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.3 First-order Formal Semantics</td>
<td>172</td>
</tr>
<tr>
<td>10.4 Executable Prototype Generator</td>
<td>173</td>
</tr>
<tr>
<td>10.5 Bridging the Gap Between Formal Methods and Software Engineering</td>
<td>173</td>
</tr>
<tr>
<td>10.6 The Clay Compiler</td>
<td>174</td>
</tr>
<tr>
<td><strong>11 Future Work</strong></td>
<td>175</td>
</tr>
<tr>
<td><strong>VI Appendices</strong></td>
<td>177</td>
</tr>
<tr>
<td>A Clay Notation Reference</td>
<td>179</td>
</tr>
<tr>
<td>A.1 Lexical Issues</td>
<td>179</td>
</tr>
<tr>
<td>A.2 Namespaces</td>
<td>179</td>
</tr>
<tr>
<td>A.3 Grammar</td>
<td>179</td>
</tr>
<tr>
<td>A.4 Precedence and Associativity</td>
<td>179</td>
</tr>
<tr>
<td>A.5 Semantic Basis</td>
<td>179</td>
</tr>
<tr>
<td>A.6 Types and Overloading</td>
<td>179</td>
</tr>
<tr>
<td>A.7 Language Features</td>
<td>179</td>
</tr>
<tr>
<td><strong>Bibliography</strong></td>
<td>180</td>
</tr>
</tbody>
</table>
Resumen

La tesis presenta el lenguaje Clay, una notación formal orientada a objetos que busca acercar los métodos formales a los lenguajes de programación y procesos de desarrollo de software más en uso hoy en día. Junto con la definición formal del lenguaje, se proporcionan herramientas y aplicaciones que demuestran la viabilidad del proyecto.

Motivación

El punto de partida de esta tesis es una reflexión sobre la ingeniería de software y el papel que las matemáticas pueden o deben jugar en su práctica, tomando como referencia las otras ingenierías que nos llevan unos siglos de ventaja y el estado actual del uso de métodos formales para el desarrollo de software.

Mucho se ha escrito sobre por qué los métodos formales no encuentran un uso en el desarrollo de software acorde con el grado de sofisticación técnico alcanzado por las diferentes propuestas académicas. Parte del capítulo de Introducción se dedica precisamente a recapitular parte de las opiniones más cualificadas sobre este particular, tratando de extraer algunas conclusiones.

Entre ellas, podemos destacar:

- La falta de profesionales cualificados en técnicas formales,
- la distancia de lenguaje entre los formalismos de especificación y los de programación,
- la escasez de herramientas adaptadas a los procesos de desarrollo más habituales,
- la percepción de que los métodos formales sirven para aumentar la fiabilidad, si bien a costa de reducir la productividad e incrementar los costes de desarrollo.
Todos estos problemas están, de una u otra manera, interrelacionados. Así, la falta de profesionales cualificados es fundamentalmente debida a no haberse establecido una serie de estándares de facto, lo cual es imposible sin una cierta masa crítica de casos de éxito, lo que desemboca en el consiguiente círculo vicioso.

El cuarto punto está también provocado en parte por la falta de engarce entre las técnicas propuestas y los modelos de desarrollo más arraigados. Por ello, los métodos formales son percibidos como obstáculos que añaden nuevas fases al desarrollo sin ayudar realmente en las “de toda la vida”. Se piensa que el uso de formalidad en la especificación de requisitos retrasará la aparición del primer código en funcionamiento, produciendo una sensación de retraso tanto a clientes como a los gestores del proyecto —de nada vale que, a la postre, el tiempo total de desarrollo se reduzca.

Algo similar ocurre con las especificaciones formales y las fases de testing: aun asumiendo que las técnicas formales proporcionasen fiabilidad total del código producido respecto a la especificación, nada protege contra la posibilidad de errores en la propia formalización de los requisitos, por lo que no nos ahorramos el testing tradicional.

Resumiendo: las herramientas formales deben crear productos percibidos como útiles en todas las fases de desarrollo: código, tests, documentación, etc.

Todo esto ha hecho que los métodos formales (al menos en su acepción más clásica) sigan restringidos a nichos donde las ventajas de su uso son claramente superiores a los inconvenientes percibidos. Estamos hablando de software crítico, de un tamaño limitado y que admite una especificación formal totalmente exhaustiva de los requisitos. El uso de las técnicas formales para el desarrollo de software de propósito general sigue siendo escaso.

En tiempos recientes se observa una tendencia a intentar romper con estas inercias por la vía de sacrificar algunos de los dogmas del desarrollo verificado (supuestamente conducente a un software correcto al 100%) en aras de obtener beneficios inmediatos, tangibles y que mejoren la calidad del software producido, al menos en promedio. Así, asistimos a la aparición de los llamados métodos formales ligeros que resultan mucho más fáciles de asimilar por los desarrolladores y que producen resultados inmediatos en forma de errores en la captura de requisitos, generación de casos de prueba, etc. Algunos de estos sistemas, de relativa simplicidad, están consiguiendo una popularidad difícil de imaginar hace unos años.

Teniendo muy presente estas experiencias recientes, nos planteamos la posibilidad de poner al día nuestro trabajo en notaciones formales orientadas a objetos, que iba encaminado fundamentalmente a reducir el desnivel lingüístico apuntado en el segundo de los problemas arriba mencionados, con la posibilidad de generar prototipos ejecutables.
Una de las ventajas que se pueden conseguir con este enfoque es que estos prototipos pueden ser integrados en el desarrollo de software tanto para reducir el tiempo de generación de código tangible como para ayudar en la validación temprana de los requisitos. De hecho, un prototipo ejecutable tiene una capacidad de validación de requisitos superior a las técnicas (generalmente basadas en model-checking) que implementan los métodos ligeros existentes en la actualidad.

La cuestión que resta es, pues, la de cómo generar esos prototipos ejecutables. Una de las áreas tradicionales de nuestro grupo de investigación es la programación lógica, donde un subconjunto de la lógica de primer orden admite un procedimiento sistemático de deducción que permite a un programador experimentado expresar una especificación lógica como un algoritmo de una eficiencia más que razonable.

A pesar de las expectativas despertadas por la programación lógica desde los años 70 del s. XX, de ser una especie de lenguaje de especificación universal, su aparente distancia lingüística con los lenguajes tradicionales de programación siempre ha hecho que los desarrolladores le diesen la espalda.

Decidimos estudiar la posibilidad de combinar una notación de alto nivel incorporando los conceptos de la programación orientada a objetos con la capacidad de deducción automática que ofrece la programación lógica.

**Clay**

Nuestra propuesta consiste en desarrollar una notación formal orientada a objetos que sea, por un lado, lo suficientemente sencilla y cercana a la forma de pensar de los desarrolladores y, a la vez, lo suficientemente formal (y sencilla) como para permitir la síntesis de programas lógicos a partir de las especificaciones.

El lenguaje propuesto, Clay, se ha diseñado, pues, a partir de dos ideas básicas. En primer lugar, debe poseer un núcleo capaz de dar cabida a las principales construcciones propias de la orientación a objetos. En segundo lugar, las especificaciones deben admitir al menos una traducción canónica a prototipos ejecutables que permita a los desarrolladores la validación temprana de los mismos requisitos.

Podemos considerar Clay una evolución *ligera* de SLAM-SL, una notación formal en la que trabajamos durante los primeros años de mi investigación predoctoral. En dicha etapa se mostró que podía diseñarse un lenguaje de especificación formal con construcciones sintácticas que facilitaban la integración de técnicas formales en procesos de desarrollo actuales. Como contrapartida, SLAM-SL resultaba ser un lenguaje extraordinariamente grande lo que dificultaba la descripción de su semántica.
La búsqueda de los dos objetivos expresados más arriba llevó a una deconstrucción de SLAM-SL que permitió completar el esfuerzo de formalización.

Desde el punto de vista de la orientación a objetos, Clay es una notación formal orientada a objetos sin concepto de estado, basada en clases y con un sistema de tipos nominal. Las clases dan lugar a tipos algebraicos mediante el uso de casos: subclases disjuntas y completas de la clase contenedora. Las clases se extienden por subclasificación (herencia). La especificación de los métodos es mediante pre- y post-condiciones, fórmulas de primer orden que establecen la relación entre el objeto receptor (self) los parámetros y el resultado. Los predicados predefinidos fundamentales que permiten la definición de otros (incluyendo pres y posts) son la igualdad (=) y la pertenencia a una clase (:).

El requisito de pretender una semántica asequible nos lleva a considerar en primer lugar una lógica subyacente clásica y de primer orden. Uno de los objetivos que se pretende conseguir con esto es sacar partido del avanzado estado de la tecnología de demostración automática para teorías de primer orden, con lo que esto puede representar tanto para depurar la metateoría de Clay como para razonar sobre las mismas especificaciones.

En cuanto a la generación de prototipos ejecutables y su uso como una herramienta de validación temprana de requisitos, ya hemos comentado que el objetivo era sacar partido de toda la tecnología existente en programación lógica, tanto a nivel de implementación como de transformación de programas. El camino ideado es el refinamiento de la codificación a lógica de primer orden para generar teorías en cláusulas de Horn ejecutables en Prolog y posteriormente optimizarlas con las técnicas antedichas.

Estructura del libro

Parte I: Introducción

Capítulo 1: Introducción Aquí se motiva la tesis, se describe en detalle el estado del arte de los métodos formales en relación con el desarrollo de software y se presentan los puntos fundamentales de la propuesta Clay.

Capítulo 2: Clay Este capítulo es una presentación informal del lenguaje de especificación. Todas las características de Clay son introducidas mediante ejemplos, y aquellas decisiones menos evidentes justificadas con argumentos prácticos y metodológicos.
Parte II: Semántica

Capítulo 3: Semántica estática de Clay Este capítulo presenta la sintaxis abstracta y un sistema de tipos para Clay basado en nombres.

Capítulo 4: Semántica de primer orden de Clay Aquí se proporciona una semántica dinámica a Clay mediante la traducción de las especificaciones a teorías de primer orden.

Parte III: El sistema Clay

Capítulo 5: Síntesis de programas lógicos Se presenta un refinamiento de la semántica dinámica del capítulo anterior que permite generar código Prolog a partir de las especificaciones Clay.

Capítulo 6: El compilador de Clay El compilador de Clay materializa las funciones presentadas en los capítulos anteriores: análisis sintáctico, comprobación de tipos, generación de teorías de primer orden y conexión con demonstradores y generación de código Prolog.

Parte IV: Aplicaciones

Capítulo 7: Agilidad formal en Clay El capítulo trata sobre la integración de métodos formales en procesos de desarrollo ágiles, programación extrema en particular.

Capítulo 8: Especificando de manera escalable Se tratan ideas para la generación de código imperativo a partir de patrones sintácticos en las especificaciones Clay. La idea es facilitar la incorporación de técnicas formales en el proceso de prototipado rápido e iterativo.

Capítulo 9: Modelado de patrones de diseño en Clay Se presenta la formalización de patrones de diseño como un ejercicio de especificación en Clay.

Parte V: Conclusión

Capítulo 10: Conclusiones

Capítulo 11: Trabajo futuro
Apéndices

Apéndice A: Referencia del lenguaje Clay  Incluye sintaxis concreta y descripción detallada de todas las construcciones.

Contribuciones de esta tesis

Resumimos a continuación las principales contribuciones de la presente tesis.

Diseño de una notación formal orientada a objetos

Una de las motivaciones de este trabajo era el estudio y la integración de conceptos de orientación a objetos, tanto los más establecidos como algunos de los más novedosos, en una notación formal que ayudase a la implantación de metodologías de desarrollo riguroso de software. Para ello, en un principio se desarrolló el lenguaje de especificación SLAM-SL. Clay es una evolución ligera de SLAM-SL que permite un mejor tratamiento formal y mecanizable de sus características esenciales.

Los aspectos más reseñables de Clay son:

- Clay es una notación formal orientada a objetos sin concepto de estado, un lenguaje basado en clases con un sistema de tipos nominal.

- Las clases se definen por casos, dando lugar a tipos de datos algebraicos.

- Las clases se extienden por subclasificación.

- Se permite la especificación implícita de los métodos mediante pre- y post-condiciones, fórmulas de primer orden que relacionan al sujeto del método (self), los parámetros y el resultado (result). Entre los predicados predefinidos esenciales tenemos la pertenencia a una clase (:), y una igualdad paramétrica por clase (=).

- La interpretación contextual de la igualdad no puede ser invalidada por subclasificación.

- La redefinición de métodos (overriding) soporta la semántica escandinava y ligado dinámico.

- Esquema de sobrecarga muy permisivo.
Definición de un sistema de tipos nominal para orientación a objetos

Tal como ya fue puesto de manifiesto en su día por Abadi y Cardelli, el subtipado basado en nombres de tipos, en lugar de en la estructura de éstos, es difícil de definir con precisión. Una de las principales desventajas del tipado estructural es que dos tipos pueden quedar accidentalmente relacionados cuando el desarrollador podría considerarlos ajenos, algo que evitado por nuestra contribución de un sistema de tipos nominal para Clay. Éste es usado tanto para descartar especificaciones ilegales como para guiar los procesos de traducción usados en la definición de la semántica de primer orden y en la generación de código Prolog a partir de las especificaciones.

Semántica formal de primer orden para Clay

Otra de las contribuciones principales es una semántica de primer orden para las especificaciones Clay entre cuyas virtudes podemos mencionar:

- Una interpretación en lógica de las principales construcciones de la orientación a objetos: definición de clases por casos, herencia, sobrecarga permissiva, ligado dinámico e igualdad estática.

- Mediante el uso de la sintaxis concreta de los demostradores Prover9/Mace4 se ha avanzado en la mecanización tanto de la metateoría de Clay como de las propias especificaciones. Por ejemplo, algunos de los teoremas de Clay han sido probados de manera automática.

Un generador de prototipos ejecutables

Hemos presentado un esquema de compilación de especificaciones Clay a programas lógicos. La principal conclusión que podemos extraer es que la generación de prototipos ejecutables a partir de una notación formal orientada a objetos es algo factible, lo que abre la posibilidad de aplicar métodos de desarrollo de software basados en la orientación a objeto en la especificación formal de requisitos y su validación ágil mediante prototipos tempranos.

Algunas de las características reseñables de este generador son:

- Generación de código a partir de especificaciones implícitas, incluso en presencia de definiciones recursivas, algo inusual en otras herramientas.
• Nuestra implementación hace uso de diferentes técnicas de programación para conseguir una eficiencia aceptable: transformación Lloyd-Topor, negación constructiva, búsqueda en anchura con profundización incremental, etc.

El compilador de Clay

Habiendo identificado la carencia de herramientas formales como una de las principales causas de la escasa penetración de los métodos formales en el desarrollo de software de propósito general, no podíamos quedarnos en una simple definición matemática de la traducción, sino que queríamos construir una herramienta real que de una manera tangible incorporase los desarrollos arriba mencionados. El compilador de Clay permite:

• el análisis sintáctico de especificaciones Clay estructuradas en diferentes módulos,

• la comprobación de tipos y la anotación de especificaciones Clay,

• la traducción de especificaciones Clay a teorías de primer orden en la sintaxis concreta aceptada por Prover9, y

• la síntesis de prototipos ejecutables Prolog.

Implementado en Haskell, el compilador de Clay ha sido desarrollado aplicando métodos y técnicas de programación funcional de última generación.

Acercando los métodos formales a las prácticas habituales en ingeniería de software

Dentro del objetivo general de contribuir al uso de las técnicas formales en el común de la ingeniería de software, se han incluido varios capítulos dedicados a ilustrar diferentes modos de aplicación de estas tecnologías. Dichos capítulos recogen nuestro trabajo previo en el diseño de SLAM-SL que como hemos dicho debe considerarse el lenguaje precursor de Clay.

• Mostramos cómo es posible integrar los métodos formales y metodologías ágiles como la programación extrema (XP). En particular, hemos estudiado las pruebas de unidad, la refactorización y, de manera especial, el desarrollo incremental desde el prisma de los métodos formales.
• Hemos definido una caracterización sintáctica de una clase de especificaciones que permite sintetizar código eficiente (Java, C++) haciendo así que proceso iterativo de prototipado rápido y los métodos formales se integren de una manera rentable.

• Finalmente, hemos mostrado cómo formalizar patrones de diseño tratándolos como operadores entre clases. La idea, en sí, estaba en el *folclore* de la comunidad de patrones de diseño desde hace tiempo, pero la hemos desarrollado completamente por vez primera.

**Publicaciones a las que ha dado origen esta tesis**


Cómo leer este libro

El lector más interesado en los aspectos formales relacionados con la descripción sintáctica y semántica del lenguaje Clay y con la generación de prototipos ejecutables pueden comenzar leyendo el capítulo 2 en el que se ofrece una descripción informal del lenguaje para continuar con la formalización del mismo en los capítulos 3 y 4 y terminar con los aspectos esenciales de la generación de programas
lógicos en el capítulo 5. Los capítulos 7, 8 y 9 son de especial interés para comprender el origen de algunas de las construcciones de Clay y sus implicaciones a nivel ingenieril.

El lector más interesado en nuestras aportaciones al área de la ingeniería del software puede saltar directamente por los capítulos 7 y 8 en los que analizamos la compatibilidad de las técnicas formales en algunas prácticas actuales de la industria. Puede entonces continuar con el capítulo 9 en el que se muestra la formalización de varios patrones de diseño utilizando nuestro propio lenguaje. Los aspectos formales en los capítulos 3 y 4 podrán leerse tras el capítulo 2 en el que se presenta informalmente Clay.
Summary

This thesis introduces Clay, an object-oriented formal notation that aims at bridging the gap between formal methods and actual programming languages in use today. Along with a formal account of the notation, tools and applications are proposed to demonstrate the feasibility of this approach.

Motivation

The starting point of this thesis is a personal reflexion on the nature of software engineering and the role that mathematics could (or should) play in its practise, taking as reference older branches of engineering and the actual usage of formal methods in software development.

Much has been written on why formal methods have not found their place in software development according to the sophistication of the existing academic proposals. Part of the Introduction is devoted to summarise some of the more qualified positions on this issue, trying to draw some useful conclusions, like:

- Shortage of practitioners trained in formal techniques,
- linguistic distance between specification formalisms and actual programming languages,
- lack of tools that can be integrated in commonly used development processes,
- the perception that formal methods can increase reliability, but only at the cost of decreasing productivity and increasing overall development costs.

All of these problems are somehow interconnected. For instance, the shortage of qualified professionals is mainly due to the lack of de facto standards which, in turn, can be a consequence of not having enough success stories, etc.
The fourth point in the list above is also partly due to a mismatch between the formal methodologies and the actual software models in use. Thus, formal methods are often perceived as obstacles that add more phases to the development process without being a real help with the “unavoidable” ones. For instance, its a common concern that being more formal in the requirements stage will delay the production of a first working prototype, thus giving customers and managers the impression that the project is being delayed – no matter how much time will be saved at the end.

A similar situation arises regarding testing and formal specification. Even assuming that formal development techniques would provide a totally reliable code from its specification, nothing protects against errors in the specification itself, which is a flaw in the argument that states that formal development techniques reduce overall development costs by saving in the testing stage, as traditional testing is still needed for dealing with holes in the original specifications.

Summarising: in order to be useful, formal tools must generate products perceived as useful in all the stages of software development: coding, testing, documentation, requirement acquisition, etc.

As a result of these shortcomings, formal methods in software remain largely confined to niches were the advantages for their use clearly win the perceived disadvantages. We are talking about critical software, often of a limited size and which allows for an exhaustive formalisation of requirements. By the contrary, adoption of formal techniques for most of the general purpose software remains scarce.

Recently, there has been a trend towards breaking these vicious circles by sacrificing part of the dogma of verified software development – allegedly aiming at error-free software – in order to obtain less ambitious, but more tangible, immediate results that can, at least, improve the average quality of software. This is the rationale behind so-called lightweight formal methods, easier to assimilate by developers and which produce immediate results like errors in requirements, test-case generation, etc. Some of these tools, in spite of their relative simplicity and apparently humble goals, are achieving an acceptance unimaginable years ago.

Taking these recent trends into account, we considered the possibility of casting our previous work on object oriented formal notations in this new context. More specifically, we were thinking of improving the capability of generating executable prototypes from specifications.

One of the strong points of this approach is that these prototypes can be integrated in the software development process in order to reduce both the time to deliver a first executable, but also to help in early requirements validation, thus attacking two of the weaknesses stated above. In fact, an executable prototype is superior, for requirements validation, to existing techniques present in existing lightweight
methods, often based on model-checking techniques.

The remaining question was, then, how to obtain those executable prototypes. Our research group has a long tradition in logic programming, were a subset of first-order logic admits a systematic deduction procedure that allows programmers to express logical specifications as algorithms of a reasonable efficiency.

In spite of the high expectations raised by logic programming since the 1970s, apparently, its linguistic distance with traditional programming languages deterred it from being widely used by developers.

So, we decided to study the possibility of combining a high-level notation – closer to object oriented programming languages – with the automated deduction capabilities of logic programming.

Clay

Our proposal consists in an object-oriented formal notation that can be, on one hand, simple enough and close to the developers way of thinking and, at the same time, formal enough to allow the synthesis of logic programs from the specifications.

The proposed language, Clay, has been designed starting from two basic premises. First, it had to be equipped with a core capable of expressing the main constructs of object oriented languages. Second, specifications had to admit one canonical translation into executable prototypes so as to allow for early requirement validation.

Clay is a lightweight evolution of SLAM-SL, a formal notation we developed during the first years of my predoctoral research. Then, we could prove that the integration of formal notations with some of the current trends in software development was possible. On the down side, SLAM-SL was too large which resulted in a complex semantics that was never formalised.

The search for the two goals described above led us to a deconstruction of SLAM-SL which finally resulted in the current formalisation.

From the standpoint of object-orientation, Clay is stateless, class-based and has a nominal type system. Classes can define algebraic data types by means of case definitions: disjoint and complete subclasses of the container superclass. Classes can be extended by means of subclassing (inheritance). Methods are specified by pre- and post-conditions, first-order formulae that relate the method subject (self), the parameters and the result (result). Distinguished predefined predicates include class membership (:), and class-indexed equality (=).
The requirement to have an accessible semantics leads to consider first-order logic as the main option. An extra benefit of this is to take advantage of the current state of FOL automatic theorem provers. This can be used to help in debugging – logically speaking – Clay's meta-theory, and to help in reasoning about individual Clay specifications.

Regarding executable prototypes and their use as a tool for early requirement validation, we have already mentioned that the goal was to get advantage of existing logic programming technology, both in the compiler side and also program transformation techniques. The approach is to refine the first-order encoding in order to generate Horn-clause theories that can be executed – possibly with the help of some of the aforementioned optimisations – in a Prolog system.

**Book Organisation**

**Part I: Introduction**

**Chapter 1: Introduction** Motivates the thesis, describing the state of the art of formal methods for software development and presents the main points of the Clay proposal.

**Chapter 2: Clay** This chapter is a quick tour of the specification language. The main characteristics of Clay are introduced by means of examples, and the less-trivial design decisions motivated on practical and methodological basis.

**Part II: Semantics**

**Chapter 3: Static Semantics of Clay** This chapter introduces the abstract syntax of the notation and a nominal type system.

**Chapter 4: First Order Semantics for Clay** Here a dynamic semantics for Clay is provided, by means of a translation of Clay specifications into first-order logic.

**Part III: The Clay System**

**Chapter 5: Synthesis of Logic Programs** A refinement of the dynamic semantics introduced in the previous chapter allows to generate Prolog code from the Clay specifications.
Chapter 6: The Clay Compiler  The architecture of the Clay compiler that integrates all the functionalities presented in previous chapters: syntactic analysis, type checking, first-order theory generation, connection with automated provers and synthesis of executable Prolog prototypes.

Part IV: Applications

Chapter 7: Formal Agility in Clay  The chapter deals with the issue of integrating formal methods in agile software development processes and, more specifically, extreme programming.

Chapter 8: Specifying in the Large  The possibility of generating imperative code from Clay specifications is studied. The goal, that follows naturally from the ideas in the previous chapter, is to introduce formal techniques in iterative rapid prototyping process, and the method proposed relies on a study of those specification patterns that occur more often.

Chapter 9: Modelling Design Patterns in Clay  The formalisation of design patterns is considered a case study for Clay specifications.

Part V: Conclusion

Chapter 10: Conclusion  Conclusions are presented in this chapter.

Chapter 11: Future Work  Lines of future work are presented in this chapter.

Appendices

Appendix A: Clay Language Reference  Includes concrete syntax and a detailed description of all the constructs.

Contributions of the thesis

A summary of the main contributions of this thesis follows.

Design of an object-oriented formal notation

One of the main motivations of this work was to study and integrate object orientation concepts – both traditional and innovative ones – into a formal notation.
that helped implement rigorous software development methodologies. This is why SLAM-SL was originally conceived. Clay is a lightweight evolution of SLAM-SL which allows a better formal and mechanised treatment of its essential features.

The key features of Clay are:

- Clay is a stateless object-oriented formal notation with a nominal type system.
- Case-based class definitions are allowed, thus supporting algebraic data types.
- Safe inheritance as subclassing.
- Implicit method specification is allowed via pre- and post-conditions, first-order formula that relate the method subject (self), the parameters and the result (result). Distinguished predefined predicates include class membership (:) and class-indexed equality (=).
- Equality is safe, i.e. it cannot be invalidated by subclassing.
- Method overriding supports the Scandinavian semantics and dynamic binding.
- Permissive overloading.

A nominal type system for object orientation

It is well known – since Abadi and Cardelli – that nominal subtyping can be trickier than structural subtyping. However, the latter has the disadvantage that two different (from the designer’s point of view) types can be accidentally identified. This is precluded in our nominal proposal.

This is used to reject illegal specifications, and also to help in the generation of executable prototypes from them.

A first-order formal semantics for Clay

Another key contribution is a first-order semantics for Clay specifications. Some of their features are listed below:

- A logic interpretation of the main features of object oriented languages: inheritance, defining classes by cases, permissive overloading, dynamic binding and static equality.
• The use of the Prover9/Mace4 automated theorem provers has allowed to advance in the mechanisation of both the meta-theory of Clay and the specifications themselves. For example, some of the theorems about Clay in this thesis have been proved semiautomatically.

**Executable prototype generation**

We have proposed a scheme for the compilation of Clay specifications into logic programs. The main conclusion is that generating executable prototypes from a formal object-oriented is feasible, which opens the possibility of having formal methods for object oriented software development based on the formal specification of requirements and its early validation by means of these prototypes.

Some of the key features of the prototype generator are:

• Code can be generated from implicit specifications, even recursive ones, something hard to find in other tools.

• Our implementation takes advantage of various logic programming techniques in order to achieve *reasonable* efficiency: constraints, constructive negation, *Lloyd-Topor* transforms, incremental deepening search, etc.

**The Clay compiler**

Having blamed the lack of practical tools supporting formal methods as one of the main causes of the scarce use of formal techniques in real-life software development, we had to build an actual tool incorporating the proposal above. The Clay compiler provides:

• syntactic analysis of modular Clay specifications, 

• type checking and annotation of Clay specifications,

• translation of Clay specifications into first-order theories in the concrete syntax accepted by Prover9, and

• synthesis of executable Prolog prototypes.

Implemented in Haskell, the Clay compiler has been developed applying the most actual methods and techniques of functional programming.
Bridging the gap between formal methods and common practises in software development

Framed in the overall goal of promoting the use of formal techniques in software engineering, several application chapters that illustrate possible uses of this technology have been included. Some of those chapters contain previous work carried out while SLAM-SL, the precursor to Clay, was being designed.

- We show how to integrate formal and agile methods, such as extreme programming (XP). More specifically, we have studied unit testing, refactoring and incremental development from the standpoint of formal methods.

- We have defined a syntactic characterisation of a class of specification that allow to synthesise efficient (imperative) code (Java, C++) allowing for an efficient integration of iterative rapid prototyping process and formal methods.

- Finally, we have shown how to formalise design patterns by treating them as class operators. While the core idea was in the design patterns community folklore, we have developed it fully for the first time.

Publications result of this research


How to read this book

Readers interested in the formal aspects involved in the syntactic and semantic description of the Clay language and executable prototype generation can start by reading Chapter 2 where an informal account of the language is given, then move on to Chapters 3 and 4 and, finally, the essentials of logic program synthesis in Chapter 5. Chapters 7, 8 and 9 are specially relevant to understand some of Clay’s constructions and its implications from an engineering point of view.

Readers interested in our contributions to software engineering can jump directly to Chapters 7 and 8 where the integrability of formal techniques into common practise is studied, then go to Chapter 9 in which several design patterns are formalised using our own language. The required formal background in Chapters 3 and 4 can be acquired after reading the informal presentation of Clay in Chapter 2.
Chapter 1

Introduction

1.1 Software Engineering

- Software engineering is the establishment and use of sound methods for the efficient construction of efficient, correct, timely and pleasing software that solves the problems such as users identify them.

- Software engineering extends the field of computer science to include also the concerns of building of software systems that are so large or so complex that they necessarily are built by a team or teams of engineers.

Dines Bjørner [18].

We cannot construct our large programs in one shot. First we need to find the to find the right models, abstractions that capture key properties of parts of the program under construction, abstractions that allow the engineer to foresee the future behaviour of its creation before actually building it. A civil engineer, for instance, would represent a steel bridge by a system of equations and the behaviour of a single truss by a differential equation. An electrical engineer can use graph theory to represent a network, and so on. The application of mathematics to software engineering is crucial if we want to describe such abstractions and to reason about them.

Software engineering, then, should be the discipline of building correct, reliable programs through the use of mathematics (sound methods in Bjørner’s words),
hopefully thanks to suitable models of the software system. However, the term is more often associated to other aspects of engineering (project management, testing, etc.) than to mathematical modelling.

Possibly, the main cause for this lies in the inherent plasticity of software: the possibility to introduce changes at any stage of development – even on a running system – is definitely something out of the question in the rest of fields of engineering, where a sharp separation between the design and production stages is essential. In software development, however, design and coding can occur simultaneously, and a combination of iterative development methods where prototypes are commonly used as a substitute for any kind of formal abstraction, and careful testing are generally the only guarantees for quality in software. To put it short, using small-scale prototypes of a complex software system can be very helpful but sounds like reducing architecture to just maquette building.

There are a few notable exceptions to the general picture. Some software systems are routinely built from formal models in such a way that all of the reasoning is done at the model level and parts of the actual code are automatically generated. Some stages of a compiler, for instance, can be almost completely developed from formal grammars using so-called compiler generators and, although software engineers could be able to develop a full compiler from scratch for any programming language with no prior training on compiler technology, the effort involved would be impractical. Moreover, as part of the compiler is generated from its specification, changes can be introduced in an agile and less costly way, as no errors can be introduced in the code due to changes in the grammars.

Other relevant exception is SQL, a widespread language for managing data in relational database management systems. Thanks to a precise semantics, based upon relational algebra, an SQL query can be transformed in an equivalent but more efficient one.

Relational database theory has been a deep influence for software design methodologies over the last three decades. In fact, some popular object oriented design notations like UML can be seen as a failed attempt at translating some of the virtues of the relational database design to the problem of general software design. Maybe, because of its lack of a mathematical definition.

So, we can see that despite the distinctive characteristics of software production, model-driven engineering is possible and that when the right languages and tools are found, this approach to the construction of software improves over craftsmanship, even economically. Certainly, it can be the case that software engineering is just too young to have found the right abstractions for its growth as a real engineering, a situation similar, perhaps, to that of civil engineering before and after the development of the differential calculus three centuries ago.
The starting point of this dissertation is that software engineering is, or can be, an engineering in the classical sense, as explained above, and that it is our duty to look for the languages to represent those abstractions and to progress towards the development of formal and software tools that make such an approach practical.

Whilst these positions may not be shared by most of the software community today, some reasons for this are discussed in the following section, it is our hope that the techniques developed in this dissertation will be a small but significant step towards the practical use of formal models for the creation of reliable software.

1.2 Formal Methods in Software Engineering

Awareness-raising and adoption of the positions advocated in the preceding section by the industry is scarce and slow. The use of formal methods for software development is still infrequent. Two arguments are often used to explain this situation:

- insufficiently trained practitioners [22], and
- lack of suitable supporting formal methods and, more important, their integration with existing practice and methodology [5].

There is an obvious chicken and egg problem with these two issues: well-trained professionals are needed to create success stories, success stories are needed to boost the creation of tools and without tools it is hard to train practitioners. The first of these causes was already mentioned among the well-known seven myths of formal methods in Anthony Hall’s seminal paper [58]. Hall’s arguments to hold that it is a myth are still valid: “the maths needed for writing specifications are quite simple”, or at least no more complex than in other engineering disciplines. In fact, most formal specification languages do not go beyond set theory and first order logic.

There is, however, some controversy about this issue. Ian Sommerville [110, Chapter 1] considers that the theoretical foundations of formal methods are insufficient to support software engineering like math supports classical engineering. The aforementioned inherent complexity of software and doubts on the scalability of formal methods are blamed for this, and, as Jean-Raymond Abrial says [4], this acts as a bottleneck for the use of automatic verification techniques like theorem proving.

However, despite the theoretical limits imposed by computability and computational complexity, the last years have shown enough positive evidence of the applicability of formal methods in several ways: creation of companies mainly de-
voted to the application of formal methods to software ([49, 100, 37, 11, 41]), tools ([98, 111, 10, 11, 121]) and success stories (being the project METEOR in 2000, the automation of the Paris Metro line 14, the most popular, other references are [123, 81, 109, 59, 31, 23, 21]) that somehow refute Sommerville’s claims. Furthermore, the companies, tools and methods mentioned have a definite application niche: hardware, embedded and critical systems. So, it is hard to criticise the tools on the technical and maturity sides when they have succeeded in such demanding environments.

In our opinion, the problem is subtler and, as David Crocker says in [40], it is founded on a perception of low productivity that is often associated to the use of formal methods:

- Productivity increase is often a more urgent need when developing software. The perception that formal methods decrease productivity contributes to restricting their usage to areas where the risks and costs associated to malfunction are the determinant factor.

- Many formal methods are just applied at the architectural design level while all or most of the code is crafted by hand. This contributes to the perception of formal methods as mostly unproductive.

- Often, applying formal methods increases the time required to deliver a first running prototype. To the managers, this can give the impression of a slower progress, even if the overall time to market, testing included, decreases.

We believe, like Daniel Jackson in [68], that one of the main factors that prevent the integration of formal methods in the development processes applied at the industry is the lack of tools that help in the development of general-purpose software.

This is also one of the points stressed out by the report *The State-of-the-Art in Formal Methods* [13], whose conclusions were that “the industry needs support to build error-free products, on time […]” and that in order to achieve that “it is necessary to work on the integration of formal tools with the rest of development tools.”

### 1.2.1 Formal Methods and Open Problems

In the following discussion, we will restrict ourselves to the area of formal methods and tools for the development of general-purpose software. We are leaving aside, then, many outstanding tools that have proved their performance in very specific application niches, e.g. hardware, firmware and protocols for communications, among others. We are not commenting either on the state of the art of
Theorem proving technology or model checking. While both technologies are crucial for the implementation of efficient tools, they are not themselves the object of our research.

A Taxonomy of Formal Methods

Classical Formal Methods

Their characteristic feature is the application of a rigorous, little automated process based on verified design, a systematic development method that uses the concept of proof as a way of checking design steps that start with a formal specification of the system. Some methods that have managed to survive until these days are VDM [69], Z [113, 42] and the B-Method [3].

Lightweight Methods

The term is applied to those formal methods that sacrifice some characteristics of traditional formal methods, like verified design, on behalf of a greater level of automation. We can mention the industry-level proposals SPARK Ada [111] from Praxis [100] and Perfect Developer [98] from Escher Technologies [49] and, more recently, the academic proposal from Daniel Jackson, Alloy [68]. Some of our previous work could also be considered examples of the lightweight approach [62], [64] and [27].

Program Specification

Other proposals to introduce rigour in software development come directly from the programming languages community, along the lines of Meyer's work [90]. A common feature of these proposals is that they are centered in some programming language. This language can be either an existing one that has been extended to support specifications and more expressive type systems, or a new one specifically designed for that effect. Among those worth to mention we have, of course, Eiffel [47, 91], and more recently JML [79, 80], Spec# [112, 14], Scala [105, 95], Nice [93, 19].

There is a certain confluence between lightweight methods and program specification languages and, in fact, we could have included SPARK Ada among the latter as well.
Object-oriented languages and formal notations

We cannot complete this overview without mentioning languages and formal notations based, or influenced by the object-oriented paradigm. To begin with, we have those which are extensions of notations used in the most established classic formal methods, like Object-Z [44], VDM++ [74].

Then, we have academic proposals inspired by the object-oriented paradigm and which are aimed at specifying general purpose software. We can mention OBLOG/Troll [106,38], OBJ/Maude [34], Larch [57,78], CASL [9] and CCSL [103].

Finally, regarding the widely-used notations for (informal) modelling of object-oriented software, like OMT (Object Modelling Technique) [104] and UML (Unified Modelling Language) [20] several specification languages have been presented that aim at providing a fully formal meaning to parts of those models [39,122,53,119,36].

Open Problems

All the preceding approaches suffer from various problems, which we summarize below.

In some sense, classical approaches assume a certain “all or nothing” philosophy, by requiring a costly verified software development process. Typically, their object-oriented incarnations are built on top of semantics largely unrelated to the object-oriented paradigm. Moreover, the lack of integrated tools and, to a lesser extent, a notation alien to developers, are still deterrents for its usage in real, large projects.

Regarding lightweight methods, in some sense they perpetuate some of the problems of classical methods. On one hand, their usage has been restricted to critical software applications, as shown by the kind of markets targetted by the purveyors of these technologies. On the other, there is a lack of tools that build products perceived useful by the industry of general-purpose software development, e.g. code generation, etc.

The semantics of lightweight methods are usually elegant, relatively simple and well established, but they seldom reflect the paradigms developers are familiar with. Also, many object-oriented concepts have semantics that are similar to the real thing, but not quite the same, providing unexpected outcomes, being inheritance the most remarkable example of this.

The main problem with program specification languages is their low abstraction level. This comes out as no surprise, since they arise as extensions of programming languages, not modelling ones. Besides, their verification techniques
are often based on model checking and model checkers give no support for loose models, also a deterrent for abstraction — a drawback also suffered by lightweight methods.

Modelling notations such as UML/OCL have, of course, been designed with programmers in mind. However, part of the reason for their successful adoption by developers is due to the laxity of their syntax and semantics. When it comes to attempt to formalize (subsets of) UML this is precisely one of the main problems. Of course, such a permissive approach is incompatible with any rigorous development process: no code generation, no early validation of requirements, etc.

Anyway, all of these approaches are there because they represented an improvement on the classical approach in one way or another. This teaches us an important lesson: success in the use of formal technology for software development will be delivered in small packages. Our own proposal, as will be shown in Section 1.5 will also be based on the combination of existing technologies with a different twist.

In the next section, we will focus on why one of these approaches has succeeded and how can we improve on it.

### 1.3 Lightweight Formal Methods

Lightweight formal methods have become relatively popular thanks to their success in early validation of requirements, a smooth learning curve and the availability of usable tools. This simplicity is obtained by replacing formal proof – which often demands human intervention – by model checking, but this also implies giving up correctness in favour of a less stringent criterion for models.

Consider, for example, the stepwise specification of queues in Alloy \[68\]. The specifier might start by just sketching the interface up, like in

```alloy
module myQueue

sig Queue { root: Node }
sig Node { next: Node }
```

that is, stating that *queues* must have a *root node* and nodes will have a *next* node to follow. The description can be “validated” by fixing a number of Queue and Node individuals and letting a tool like *Alloy Analyzer* \[7\] model check the specification and show graphically the different instances found. Of course, some of these instances will be inconsistent with the intuition in the specifier’s mind – e.g. unreachable nodes or cyclic queues, that can be revealed with very small models. Further constraints, like
can be added to the myQueue module in order to supply some of the pieces missing in the original requirements. The first fact states that for every node there must be some queue such that the node lies somewhere in the transitive-reflexive closure of the next relation starting with the root of that node. The second one says that no node can be in the transitive closure of the next relation starting in itself. Model checking the refined specification will generate less instances, thus allowing to explore bigger ones, which will hopefully lead to reveal more subtle corners in the requirements.

As said before, this approach is extremely attractive: requirements are refined in a stepwise manner guided by counterexamples found by means of model checking, and the whole process is performed with the help of graphical tools.

However, there are also some limitations inherent to this approach. Leaving aside the fact that total correctness of the specification is abandoned in favour of a more relaxed notion of being not yet falsified by a counterexample, which can make the whole enterprise unsuitable for safety critical domains, the use of model checking rather than proof based techniques also brings other negative consequences, such as limiting the choice of data types in order to keep models finite, making extremely difficult to model and reason over recursive data types like naturals, lists, trees, etc. (See [68], Ch. 4, Sec. 8.)

### 1.4 Executable Prototypes and Logic Programming

A natural alternative to model checking the initial requirements is to produce an executable prototype from them. Using the right language it is possible to obtain recursive code and validation can be guided by testing, which might also be automated by tools such as QuickCheck [33].

Regarding how to obtain the prototypes, there are several possibilities. One of them is to follow the correct by construction slogan and to produce code from the specification, either by means of a transformational approach that often requires human intervention, or by casting the original problem in some constructive type theory that will lead directly to an implementation in a calculus thanks to the Curry-Howard isomorphism [97, 25, 17, 94].

Another possibility is to use logic programming. In this case, executable specifications are obtained free of charge, as resolution or narrowing will deal with the existential variables involved in any implicit (i.e. non-constructive) specification.
1.5 Our Proposal

Readers familiar with logic programming will remember the typical examples – obtaining subtraction from addition for free, sorting algorithms from sorting test, etc. – and those familiar with logic program transformation techniques will also recognize that these can be used to turn those naive implementations into decent prototypes.

However, when it comes to practical usage, none of these formalisms can compete with the lightweight methods above, due to the great distances separating them from the notations used for modelling object oriented software.

1.5 Our Proposal

This thesis proposes the development of an object oriented formal notation which is, on one hand, simple and close to the way a developer structures her software and on the other, simple and formal to enable the synthesis of logic programs from the specifications.

The specification language, Clay, is being designed around two driving ideas. First, the language must be small but make room for the basic constructs in object oriented programming. Second, specifications must admit at least one canonical translation into executable prototypes to allow the specifier to interactively validate her own specifications.

Clay is a lightweight evolution of SLAM-SL [62], a large object oriented notation designed to bridge the gap between formal methods and more widespread software engineering processes ([125, 65, 64]), and which we developed in the first years of this research. We can see Clay as a desugared version of SLAM-SL where some formal aspects superficially treated in the cited works have been nailed down in order to enable the synthesis of executable prototypes.

From the object-oriented viewpoint, Clay is a stateless object oriented formal notation, a class-based language with a nominal type system. Classes are defined as algebraic types in the form of case classes: complete and disjoint subclasses of the defining class. Classes can be extended by subclassing. Methods are specified with pre- and postconditions, first order formulae involving self (the recipient), parameters and result (the resulting object). Atomic formulae are equalities (=) and class membership (:).

In order to achieve the requirement for an affordable semantics, we wanted to have it expressed using first-order logic (FOL). Thus, an interlingua declarative semantics for Clay is provided by translation into first-order logic. Besides the conceptual simplicity, one added revenue is to take advantage of existing theorem proving technology in order to save work during the language design and also for
sanity tests in the software design process. The goal is that Clay tools generate an axiomatisation in Prover9/Mace4 \[101\] syntax. Then, early detection of inconsistencies is achieved by the combination of automatic theorem proving (Prover9) and model checking (Mace4) of the first order logic theories that reflects the structure of Clay specifications.

Regarding the generation of executable prototypes, the main idea is to refine the encoding of the language into FOL in order to get a naive Horn clause theory from a given Clay specification. Later, using well established techniques of logic-based program synthesis and transformation, the code generator will be able to provide a running prototype which, while less efficient than code crafted by hand, should be usable in order to serve as feedback for the early validation of formalized requirements.

### 1.6 Structure of the Book

Next chapter gives a quick tour of Clay by means of examples. All of their features are presented and motivated. Altogether with the present chapter, they make the first part of the book.

Part II is devoted to the formal semantics of Clay. Chapter 3 introduces the nominal type system and Chapter 4 the dynamic part, by means of an encoding into first-order logic.

Part III contains the formal description of the code generator engine along with experimental results.

In Part IV, chapters ... to ... describe a number of applications of the notation to several common situations in software engineering and object-oriented software development.
Chapter 2

Clay

This chapter presents the main features of Clay by means of examples that will be used throughout the book. Sections 2.1–2.6 introduce the basic constructs of the language. In Section 2.7 we will deal with some nontrivial aspects of our notation that may differ from other object-oriented specification and programming languages. Subsequent chapters will provide a more formal presentation of the language. Chapter 3 is devoted to the type system of Clay, and Chapter 4 provides a logical semantics. For a full language reference, including the concrete syntax, the reader is referred to Appendix A.

2.1 Classes and Objects

The central notion in Clay is that of class. Clay specifications are sets of class specifications which describe the structure and behaviour of objects. Let us begin with an example adapted from [2]:

class Cell {
  state CellCase { isEmpty : Bool,
                   contents : Nat }
  invariant { isEmpty : True ⇒ contents = 0 }
  ... 
}

Class specifications start with the keyword class. In our example, Cell describes a container of, at most, one natural number. Keyword state defines part of the data model by means of object composition: a boolean field isEmpty that indicates
whether the natural field contents is relevant or not.

Objects interact with each other by exchanging messages. Field names are understood as messages and their values are returned as response. The expression $c ← \text{isEmpty}[^1]$ expresses the fact that cell $c$ contains no relevant information.

Known properties and constraints of the domain can be made explicit with the use of invariants. For instance, we can establish that whenever a cell is empty, its content is forced to be 0 using the invariant construct. The invariant establishes it and results in the following Clay formula:

$$\forall c : \text{Cell} \ (c ← \text{isEmpty} : \text{True} \Rightarrow c ← \text{contents} = 0)$$

As we can see, Clay formulae are, in general, first-order. Quantifiers, first-order variables and predicates are at the specifier's disposal. This includes a number of predefined predicates. Among those relevant for the management of classes and objects we have `instance_of` and `eq`, which will be often used in their infix form: “:” and “=”.

Predicate “:” checks whether an object is an instance of a given class. In our example, the type system establishes that $c ← \text{contents} : \text{Nat}$ if $c : \text{Cell}$. More interesting is its use in `isEmpty : \text{True}`, a construct that will be discussed in more detail in Sections 2.3 and 2.7.2. For the moment, it will be enough to say that `True` is the subclass of `Bool` that describes all the boolean data that are `true`.

Regarding the equality predicate, each class introduces its own version. In general, equality is observational, i.e. two objects are regarded equal if they are indistinguishable when sending messages at them. In our example, this boils down to equality of every component of the object: if we have two cells $c_1$ and $c_2$, predicate $c_1 = c_2$ holds if $c_1 ← \text{isEmpty} = c_2 ← \text{isEmpty}$ and $c_1 ← \text{contents} = c_2 ← \text{contents}$ hold.

We will complete our example with the definition of some more methods of class `Cell`. In the form of pre- and postconditions, method definitions specify how instances of `Cell` will respond to messages. These formulae relate the receiver of the message (`self`) and parameters of the message with its answer (`result`):

```clay
class Cell {
...
 modifier set (v : Nat) {
    post { result ← isEmpty : False ∧ result ← contents = v }
}

observer get : Nat {
    pre { self ← isEmpty = Bool ← mkFalse }
```

[^1]: `←` is used in Clay to denote method invocation.
2.2 Inheritance

Messages get and set are messages that instances of Cell respond to. The keyword modifier indicates that the result of the message \( c \leftarrow \text{set}(n) \) represents the modification of \( c \), a non-empty cell with internal value \( n \).

Observer methods introduce messages which allow observing calculated attributes of objects and are declared with the keyword observer. Expression \( c \leftarrow \text{get} \) denotes the response of \( c \) to the message get: its internal value of field contents if \( c \) is not empty, otherwise the response is an unknown instance of Nat.

In Clay classes are themselves objects that respond to messages declared as constructors (keyword constructor):

```clay
class Cell {
    ...
    constructor mkCell {
        post { result←isEmpty = Bool←mkTrue }
    }
    ...
}
```

Expression \( \text{Cell} \leftarrow \text{mkCell} \) denotes an object of class \( \text{Cell} \) that represents an empty cell, and, established by the invariant, with field contents equal to 0.

Since classes are objects, they are instances of other classes. In our example the following predicates hold:

\[
\text{Cell} : \text{MetaCell} \land \text{MetaCell} : \text{Meta} \land \text{Meta} : \text{Meta}
\]

In words: for every class name \( C \) the user defines in Clay a new class name Meta\( C \) is automatically created. To avoid an infinite chain, every Meta\( C \) is an instance of predefined class Meta which is an instance of itself.

### 2.2 Inheritance

Now we extend the behaviour of the Cell class with the ability of restoring the state of cells to their previous content:

```clay
class ReCell extends Cell {
    state ReCellCase { wasEmpty : Bool,
        backup : Nat }
```
invariant { wasEmpty : True ⇒ backup = 0 }

constructor mkReCell {
    post { result = Cell←mkCell
           ∧ result←wasEmpty : True
    }
}
...
}

Class ReCell extends Cell with a new backup field.

Inheritance is declared by means of keyword `extends`, which induces a subclass relation. Again, this is an atomic predicate that will be used more often in its infix form: "<-:". The relation obeys the expected rules of reflexivity, transitivity and subsumption. Clay adopts the `inheritance-is-subtyping` approach what in a class-based language results in no distinction between types and classes.

Also, we have designed Clay to follow the Liskov substitution principle (LSP) so all the properties of the instances of a superclass are inherited by the instances of a subclass. In our example, the first property inherited is the invariant of the superclass:

∀ c : ReCell (c←isEmpty : True ⇒ c←contents = 0)

In our view, the most important aspect of the LSP is that a subclass cannot invalidate by overriding any property specified in its superclasses, otherwise the whole specification will be considered inconsistent in Clay.

This approach is essential when we are specifying in the large: the specifier needs to reason locally to a class and a subclass cannot show a behaviour that forces the specifier to take into account all the subclasses. The approach adds another advantage: specifications can be much more concise since it is not necessary to state properties already stated in superclasses. The main drawback is certain loss of flexibility but, in our view, the decision pays back.

Observe how in the specification of `set` we omit everything already specified in the superclass:

class ReCell extends Cell {
    ...
    modifier set (v : Nat) {
        post { result←wasEmpty = self←isEmpty
               ∧ result←backup = self←contents }
    }
    ...
}
If \( r \) is an instance of \texttt{ReCell} then the postcondition of message \texttt{set} in \texttt{Cell} is inherited and \( r \leftarrow \texttt{set}(5) \leftarrow \texttt{contents} = 5 \) holds.

To end with the example we specify the message \texttt{restore} that recovers the previous state of the cell:

```clay
class ReCell extends Cell {

    modifier restore {
        post { 
            result \leftarrow \texttt{isEmpty} = self \leftarrow \texttt{wasEmpty}
            \wedge
            result \leftarrow \texttt{contents} = self \leftarrow \texttt{backup}
            \wedge
            result \leftarrow \texttt{wasEmpty} = self \leftarrow \texttt{wasEmpty}
            \wedge
            result \leftarrow \texttt{backup} = self \leftarrow \texttt{backup}
        }
    }
}
```

2.3 Case Classes

In Clay, instances of a class are the disjoint and complete sum of the instances of its case classes. Case classes are indicated with the keyword \texttt{state} – the terminology comes from its similarity to the design pattern \texttt{State} [54] – and this implicitly introduces subclasses of the original class.

```clay
class Nat {

    state Zero {}

    state Succ {pred : Nat}

    ... 
}
```

Three classes are introduced with the lines above: \texttt{Nat}, \texttt{Zero} and \texttt{Succ}. If \( n \) is an instance of \texttt{Nat} then it is an instance of \texttt{Zero} or, exclusively, of \texttt{Succ}. The following Clay formula expresses it:

\[
\forall \: n : \texttt{Nat} \: ((n: \texttt{Zero} \lor n: \texttt{Succ}) \land (n: \texttt{Zero} \leftrightarrow \neg n: \texttt{Succ}))
\]

Moreover, \texttt{Zero} and \texttt{Succ} are subclasses of \texttt{Nat} so the following formulae hold for every \( n \):

\[\text{Readers familiar with functional programming will recognize here a mechanism to emulate algebraic data types.}\]
class Nat {
    ...
    modifier add (n : Nat) {
        post {
            (self : Zero ⇒ result = n)
            ∧ (self : Succ
                ⇒ result = Nat←mkSucc(self←pred←add(n)))
        }
    }
    ...
}

Listing 2.1: Adding natural numbers

\( n : \text{Zero} \Rightarrow n : \text{Nat} \)
\( n : \text{Succ} \Rightarrow n : \text{Nat} \)

The case classes \text{Zero} and \text{Succ} also introduce the constructor methods \text{mkZero} and \text{mkSucc}. Both are valid messages of class \text{MetaNat} and can be sent to the object \text{Nat}: \text{Nat←mkZero}. The response of \text{Nat} to message \text{mkZero} is an instance of \text{Nat} and it is an instance of \text{Zero}. When a case class defines fields, its constructor uses them as the definition of the parameters. Constructor \text{mkSucc} has as its parameter an instance of \text{Nat} so we can write \text{Nat←mkSucc(Nat←mkZero)} to represent number 1. We will use 0, 1, 2, etc. to abbreviate the following expressions: \text{Nat←mkZero}, \text{Nat←mkSucc(Nat←mkZero)}, \text{Nat←mkSucc(Nat←mkSucc(Nat←mkZero))}, etc.

The combination of predicate “:”, case classes and fields can be used to emulate the effect of pattern matching, as shown in Listing 2.1.

We show now the specification of class \text{Bool} and we will understand why the construct \text{isEmpty : True} establishes that \text{isEmpty} is true in the invariant of class \text{Cell}:

class Bool {
    state True {}
    state False {}
    ...
}

In fact, \text{isEmpty} = \text{Bool←mkTrue} and \text{isEmpty : True} are equivalent in Clay since \text{Bool←mkTrue} : \text{Bool} and case classes define an algebraic type:

\[ \forall n : \text{Bool} ((n : \text{True} \lor n : \text{False}) \land (n : \text{False} \leftrightarrow \neg n : \text{True})) \]

To finish with case classes we show a simple example of an enumeration:

class RGB {
2.4 Generics

To address the problem of reusability and extendability of specifications Clay supports three key techniques: inheritance (Section 2.2), disjoint sums (case classes, Section 2.3) and parametric polymorphism by means of generics.

The following Clay specification defines generic pairs:

```clay
class Tup2 <x,y> {
  state Tup2_ {fst : x, snd : y}
}
```

Variables x and y are class parameters of the class constructor Tup2 and can be bound to any two classes. We can then create the pair (true,0) with the following expression `Tup2<Bool,Nat>←mkTup2_(true,0)` which responds to `fst` and `snd` messages with the proper instances of Bool (class parameter x) and Nat (class parameter y) observing the following properties:

- `Tup2<Bool,Nat>←mkTup2_(true,0)←fst = true`
- `Tup2<Bool,Nat>←mkTup2_(true,0)←snd = 0`

Generics, case classes and inheritance can be combined, as the following example shows:

```clay
class Seq<x> extends Collection<x> {
  state Empty { }
  state NonEmpty {head : x, tail : Seq<x>}
  ...
}
```

The genericity in the container class is inherited by its case subclasses. Expressions like `Empty<x>` and `NonEmpty<x>` have to be used if we want to do pattern matching on the specification of message `size`:

```clay
class Seq<x> extends Collection<x> {
  ...
  observer size : Nat {
```
Generics in Clay can also be introduced using the mechanism of bounded type parameterisation. Let us assume the following specification for a class Poset:

```plaintext
class Poset {
  observer lte(other : Self) : Bool {
    post { result : True ⇒ other.lte(self) : True ⇒ self = other
           ∧ result : False ⇒ other.lte(self) : True
           ∧ forall o : Self ( result = other←lte(o) ⇒ result = self←lte(o))
  }
}
```

Using it, a new generic class SortedSeq can be specified but constraining its class parameter to be a subclass of Poset:

```plaintext
class SortedSeq<x extends Poset> {
  state SortedSeq_ { s : Seq<x> }
  invariant { ∀ i : Nat ( s←dom←in(i) ⇒ s←dom←in(i+1)
                       ⇒ s←elem(i)←lte(s←elem(i+1))) }
  ...
}
```

Then, SortedSeq class constructor can be instantiated with any class that inherits from Poset like NatPoset:

```plaintext
class NatPoset extends Nat Poset {
  observer lte(other : Self) : Bool {
    post { self : Zero ⇒ result : True
            ∧ self : Succ ⇒ other : Zero
                ⇒ result : False
            ∧ self : Succ ⇒ other : Succ
                ⇒ result = self←pred←lte(other←pred)
  }
}
```

---

3We will offer details about Self in Section 2.7.1. For the moment it can be understood as a class variables bounded by Poset.
but, for instance, SortedSeq<Nat> would not be a valid class name.

2.5 Pre- and Postconditions

Using postconditions we can state requirements of the result of a message leaving out any unnecessary algorithmic detail. Let us look at the specification of message half in our natural numbers example:

```plaintext
class Nat {
    ...
    observer half : Nat {
        post { self = result←add(result) }
    }
    ...
}
```

The postcondition in the post clause forces the following property for every instance \( n \) of Nat:

\[
{n\leftarrow\text{half} + n\leftarrow\text{half} = n}
\]

Any actual implementation would find impossible to comply with the property for odd numbers. To avoid this kind of inconsistencies the specifier can introduce a precondition that states a restriction on the use of a message:

```plaintext
class Nat {
    ...
    observer half : Nat {
        pre { self←even : True }
        post { self = result←add(result) }
    }

    observer even : Bool {
        post {
            result : True ⇔ \exists\ n : \text{Nat} (self = n←add(n))
        }
    }
}
```

Now, the whole specification of half establishes the previous property but is protected by the precondition:

\[
{n\leftarrow\text{even} : True \Rightarrow n\leftarrow\text{half} + n\leftarrow\text{half} = n}
\]
In Clay, sending a message not satisfying the precondition would result in an unknown object. In our example, $1 \leftarrow \text{half}$ is a natural number although we do not know which one. More details about this in Section 2.7.1.

### 2.6 Assertions and Solutions

Clay allows to specify conditions that postconditions must guarantee in the form of assertions:

```plaintext
class ReCell {
...
    modifier restore {
        post { ... }
    }
    assert {¬ result ← isEmpty ⇒ result ← get = result ← restore ← get}
}
```

Such conditions make the model easier to understand and at the same time can be converted in very useful products like proof obligations to check the integrity of the specifications, test cases for future implementations or run-time checks.

In order to document algorithmic details about the way in which the result of a message can be calculated we use solutions. A solution is a formula that entails the postcondition, generally written in a way that allows a more straightforward translation into code:

```plaintext
class Nat {
...
    observer even : clay.lang.Bool {
        post { ... }
        sol {
            self : Zero ∧ result : True
            ∨ self : Succ ∧ result = self ← pred ← even ← neg
        }
    }
}
```

When a `sol` clause is present, its content, rather than that of the `post`, can be used to generate an executable prototypes.
2.7 Clay Idiosyncrasy

In previous sections we have presented characteristics of Clay that should not have led to controversy. In this section we keep the informal line of the chapter but we will answer to questions about what we consider the most interesting aspects of the notation.

2.7.1 The Meaning of a Specification

Although Chapter 4 is devoted to the formal semantics of Clay, we will informally explain in this section the meaning of main constructs of the language. Let us explore this example:

```
class Map<key extends Poset,info> extends Collection<info> {
    state Map_ { map : Seq<Tup2<key,info>> }
    invariant { ... } --- sorted and no duplicated keys

    observer get (k : key) : info {
        pre {
            exists i : info ( self ← map ← in(Tup2 ← mkTup2(k,i)) : True)
        }
        post { self ← map ← in(Tup2 ← mkTup2(k,result)) : True }
    }
}
```

Every construction introduces several facts:

*Class declaration.* Line 1 establishes that Map<K,I> is a valid class, subclass of Collection<I>, if K and I are valid classes and K <: Poset.

*Case class declaration.* Line 2 declares Map<K,I> ← mk(s) : Map<K,I> if s : Seq<Tup2<K,I>>, and Map<K,I> ← mk(s) respects the invariant.

*Method declaration.* Line 5 establishes

∀ s : Map<K,I> ⇒ ∀ k : K ⇒ s←get(k) : I

In words, s←get(k) is inevitably an instance of I.

*Method specification.* Line 9 introduces the following fact about any instance s of Map<K,I> and any instance k of K:

s←map←in(Tup2←mkTup2(k,s←get(k))) : True
guarded by the precondition of line 9:

\[
\text{exists } i : \text{info } (s \leftarrow \text{map} \leftarrow \text{in}(\text{Tup2} \leftarrow \text{mkTup2}(k, i)) : \text{True})
\]

**Self, capitalized**

Every class specification introduces a class variable `Self` bounded by the class under definition. `Self` represents the “current” class in case of inheritance. The specifier can use `Self` at any moment, in Section 2.4 we used it as a parameter type. Keyword `modifier` introduces a `Self` as the class of the resulting object.

In the specification of modifier `add` in class `Nat`, the resulting object is an instance of `Self`. Any instance of a subclass `C` of `Nat` will answer with an instance of class `C`.

For those readers worried about the impact of covariant and contravariant positions we will explain our approach in Section 2.7.1.

**Undefinedness**

What is the meaning of `1 \leftarrow \text{half}` or `m \leftarrow \text{get}(k)` when key `k` is not in map `m` in Clay? We follow the same approach as Z with respect to *undefined* expressions: every expression has a value though sometimes we do not know what it is.

The approach allow us to take `1 \leftarrow \text{half}` to be 42 consistently and everything will go well. If we consistently take it to be 27 everything will go well. What Clay will not do is to take `1 \leftarrow \text{half}` to be sometimes 37 and sometimes 42. Otherwise we would lose the reflexivity of the equality or collapse all the values.

Whichever value `1 \leftarrow \text{half}` is, it cannot be discovered in a deduction process because the precondition `1 \leftarrow \text{even} : \text{True}` acts as guard when we try it.

**Loose Specifications**

When some aspects of the domain are not yet relevant to the requirements we can leave them unspecified. The usual refinement process can be iteratively applied to *tighten* the specification while each new specification has to be proven consistent. Loose specifications naturally appear in abstract classes or interfaces in object oriented specifications.

The class `Collection<\alpha>` defines methods `insert` and `includes` in the following manner:

\[
\text{class } \text{Collection}<\alpha> \{
\]
... 

\begin{verbatim}
modifier insert(e : x) {
    post { result ← includes(e) : True }
}
modifier includes(e : x) : Bool {
}
\end{verbatim}

Subclasses of Collection\(<x>\) will guarantee that its instances will respond with *the* instance True to the sequence of messages insert\((e)\) and includes\((e)\).

If the user specifies a contradictory property in a subclass then the whole specification turns inconsistent.

Since the refinement process and inheritance require the same consistency proofs, for this work we have not introduced a specific construct for declaring refinement. In the same line Clay does not distinguish between interfaces and abstract classes, being interfaces completely loose specifications apart from the typing aspects.

**Nondeterminism**

As mentioned in the previous paragraph about undefinedness, in Clay every expression has a value though sometimes we do not know what it is. Given the following specification of sqrt:

\begin{verbatim}
class Nat {
...
modifier sqrt {
    post { \( \exists r : \text{Nat} \ (r ← \text{mul}(r) = \text{self}) \)
        \Rightarrow result ← \text{mul}(\text{result}) = \text{self}
        \land \( \forall x : \text{Nat} \ (\text{self} ← \text{lte}(x) : \text{True}) \)
        \Rightarrow \text{self} ← \text{sqrt} ← \text{lte}(x ← \text{sqrt}) : \text{True} }
}
\end{verbatim}

the square root of 25 is 5 but the square root of 21 can be 4 or 5. Clay will return one of the values consistently and not 4 sometimes and 5 the others.

**Binding**

Binding is the process by which an expression is associated with a behaviour \[24\]. In Clay, behaviour is given by the properties that affect the expression during a deductive process.
In the following example, method \( m2 \) in class \( A \) delegates its response to the response of message \( m1 \):

```plaintext
class A {
  observer m1 : Nat {
    post { result = 4 \lor result = 5 }
  }

  observer m2 : Nat {
    post { result = self ← m1 ← half }
  }
}
```

The response of \( A ← mkA ← m2 \) will consistently be any natural number, 2 included. Nevertheless, we don’t now which although we now that the following formula holds:

\[ B ← mkB ← m1 = 4 \Rightarrow B ← mkB ← m2 = 2 \]

Let us override the method \( m1 \) of class \( A \) in subclass \( B \):

```plaintext
class B extends A {
  observer m1 : Nat {
    post { result = 4 }
  }
}
```

Clay follows the Scandinavian semantics \([24]\) for method overriding so the deduction process should reflect it by finding \( B ← mkB ← m2 = 2 \) to be true consistently:

- \( B ← mkB ← m2 = 4 \lor B ← mkB ← m2 = 4 \) since \( B ← mkB : A \), and

- \( B ← mkB ← m2 = 4 \) since \( B ← mkB : B \).

**Permissive Overloading**

Clay follows a very permissive overloading of method names to allow the use of substitutibility and specialization mechanisms \([28]\). There is no contradiction in allowing to declare a method in a subclass with arguments being supertypes (contravariant arguments) and the result a subtype (covariant result).

For every class \( C \), let \( \text{Sub}C \) and \( \text{Sup}C \) be classes such that \( C <: \text{Sup}C \) and \( \text{Sub}C <: C \). The meaning of the definition

```plaintext
class A {
  observer m (x : X) : R {
```
is that for every instance $a$ of $A$ and every instance $x$ of $X$ the expression $a - m(x)$ is an instance of $R$.

We can define the subclass $SubA$ that overrides method $m$ in various ways while respecting that arguments are contravariant and the result is covariant:

```java
class SubA extends A {
    observer m (x : X) : R {
    }
    observer m (x : X) : SubR {
    }
    observer m (x : SupX) : R {
    }
    observer m (x : SupX) : SubR {
    }
}
```

The first definition does not add any information to the previous meaning. The second definition is more interesting and establishes that the reply to the message $m$ of an instance of $SubA$ it is something more specific than an instance of $R$, an instance of $SubR$. Third and fourth definitions establish that instances of $SubA$ are able to reply to message $m$ if the argument is something more general than an instance of $X$, an instance of $SupX$.

Covariant arguments are also allowed in Clay and its interpretation is the specialization of the method behaviour in the instances of the subclass:

```java
class SubA extends A {
    observer m (x : SubX) : R {
    }
    observer m (x : SubX) : SubR {
    }
}
```

Obviously, an instance of $SubA$ will reply to any message $m(x)$ being $x$ an instance of $X$. Both definitions establish an specialised or refined behaviour in the case an instance of $SubX$ is used as argument. This opens the introduction of $Self$ in the arguments allowing the definition of binary methods.

Contravariant results are not allowed in Clay. Not because they introduce any inconsistency but because they provide no extra information:

```java
class SubA extends A {
```
If $a'$ is an instance of SubA we already know that $a' \leftarrow m(x)$ is an instance of R and, obviously, it is already an instance of SupR.

### 2.7.2 Booleans and Formulae

In Clay, expressions and formulae live in different syntactic categories. To avoid any confusion in this work we prefer to make this distinction absolutely explicit. This decision leads to cumbersome notation like

```plaintext
observer half : Nat {
  pre { self ← even : True }
}
```

In future versions of the language we plan to allow the user to write formulae as instances of Bool as a kind of syntactic sugar. The previous example could be written as

```plaintext
observer half : Nat {
  pre { self ← even }
}
```

### 2.7.3 Multiple Inheritance

Coloured numbers will be our example to illustrate how Clay supports multiple inheritance\(^4\)

```plaintext
class RGBNat extends Nat RGB {
  constructor mkRedZero {
    post { self : Zero ∧ self : Red }
  }
}
```

Instances of RGBNat inherit all the properties of superclasses. If $p : RGBNat$ we can use it in any circumstance an RGB or a Nat can be used. The following constructions are valid:

- $\text{Nat} \leftarrow \text{mkZero} \leftarrow \text{add}(p)$
- $p : \text{Red}$

\(^4\)Although just single inheritance is allowed in the code synthesis of logic programs (Chapter 5).
2.7 Clay Idiosyncrasy

2.7.4 Equality

“One can’t do mathematics for more than ten minutes without grappling, in some way or other, with the slippery notion of equality.”[87]

We will see that Clay is very much like mathematics regarding this:

```plaintext
class Nat {
    ...
    modifier mul (n : Nat) {
        post { self = Nat←mkZero ∧ result = Nat←mkZero
            ∨ ... }
    }
    ...
}
```

Being RGBNat a subclass of Nat with an instance that represents a red zero, we expect the following property to be true:

∀ n : Nat (RGBNat←mkRedZero←mul(n) = Nat←mkZero)

Nevertheless, the property is true depending on the truth of the first equality in the postcondition of mul:

RGBNat←mkRedZero = Nat←mkZero

Obviously RGBNat←mkRedZero and Nat←mkZero are not the same object. Should both expressions be equal? In my view, they should. Otherwise, the specifier would have to make explicit all the relevant properties that reveal that RGBNat←mkRedZero and Nat←mkZero behave indistinguishably in the context of message mul. That decision would lead, in general, to much more cumbersome specifications.

In Clay, the equality predicate is implicitly indexed by the minimum subtype of the compared instances in the context in which the formula appears. In the example of mul, the minimum type of self and Nat←mkZero is Nat. The semantics of Clay establishes that no properties of self other than those reachable from Nat are checked when compared with Nat←mkZero.

Consider, for instance, adding a red zero to a zero: 0←add(RGBNat←mkRZ). According to Listing 2.1 the result is equal to RGBNat←mkRZ up to Nat, so the property 0←add(RGBNat←mkRZ) : Zero holds and 0←add(RGBNat←mkRZ) : Red is not even a valid expression. Nevertheless, we can write a formula on an instance cn of RGBNat establishing

\[ cn = 0←add(RGBNat←mkRZ) \]
since equality is applied modulo Nat, the minimum common subtype of \( cn \) and 0←add(RGBNat←mkRZ).

As a result of indexing statically the equalities, some specifications may seem strange at first sight, like that of the constructor mkRedZero:

class RGBNat extends RGB Nat {
constructor mkRedZero {
post { result = RGB←mkRed \& result = Nat←mkZero }
}
}

The postcondition seemingly states that RGB←mkRed and Nat←mkZero are equal, by transitivity:

\[
\text{RGBNat←mkRedZero} = \text{RGB←mkRed} \\& \text{RGBNat←mkRedZero} = \text{Nat←mkZero}
\]

A deeper examination reveals that equalities are indexed in the following way:

class RGBNat extends RGB Nat {
constructor mkRedZero {
post { result =_{RGB} \text{RGB←mkRed} \& result =_{Nat} \text{Nat←mkZero} }
}
}

The property is then

\[
\text{RGBNat←mkRedZero} =_{RGB} \text{RGB←mkRed} \\& \text{RGBNat←mkRedZero} =_{Nat} \text{Nat←mkZero}
\]

and RGB←mkRed = Nat←mkZero is clearly invalid.

Informally, \( o_1 =_A o_2 \) holds in Clay if and only if:

- \( o_1 \) and \( o_2 \) are instances of the same case class \( C \) of \( A \),
- \( o_1 \leftarrow f =_B o_2 \leftarrow f \) for each field \( f \) with class \( B \) declared in the case class \( C \), and
- \( o_1 =_{A'} o_2 \) for every subclass \( A' \) of \( A \).

2.7.5 Invariants and Consistency

In most formal methods invariants are one of the main elements of the proof obligation process, i.e. invariants are not assumed and specifiers need to proof that their postconditions ensure the invariants.
2.7 Clay Idiosyncrasy

In Clay, invariants augment the postconditions the specifier writes. We consider that this decision leads to more concise specifications while the proof obligation is turned into an inevitable consistency checking.

2.7.6 Other Features

We remind you that Clay is a lightweight evolution of SLAM-SL [62]. In this work we have advanced in the formal treatment of some aspects only superficially treated in the cited works and, at the same time, many features and syntax from SLAM-SL ([125, 65, 64, 60]) have been left behind. Our plan is to reintroduce them in Clay in the future.
Part II

Semantics
This chapter presents the Clay type system. The Clay type system is based on type names, it does not support structural subtyping to avoid accidental matching of unrelated types and it is harder to define precisely than the usual structural type systems ([2]). To give a precise definition we introduce an elaborated abstract syntax of the language in Section 3.1 and then, in Section 3.2, a whole type system for Clay is elaborated.

### 3.1 Abstract Syntax

Textual representation of Clay specifications is not appropriate at all to describe the semantics of the language. We need a mathematical object to describe and manipulate Clay specifications. This section is devoted to define an abstract syntax of Clay that will be used in Section 3.2 to introduce a type system that validates specifications and extracts important information used during the translation processes treated in Chapters 4 and 5.

Let us start with the abstract representation of Clay specifications. Clay specifications are represented by two partial functions: *Bounds* and *Spec*.

*Bounds* is a partial function from class identifiers (*Cl*) to sequences of class expressions (*CExpr*) that capture the arity and bounds of a class identifier:

\[\text{Bounds} = \text{Cl} \rightarrow \text{CExpr}^*\]
Class expressions ($CExpr$) are terms that follow this concrete syntax:

\[
CExpr \; ::= \; CI < CExpr, CExpr, \ldots CExpr > \\
| \quad CVar
\]

i.e. class variables or a class identifier applied to several class expressions.  

$Spec$ is a partial function from class expressions to class specifications ($CS$):

\[
Spec \; = \; CExpr \rightarrow CS
\]

$Spec$ is undefined if the class expression does not respect the bounds establishes by function $Bounds$.

A class specification is a triple with the set of superclasses ($\mathbb{2}^{CExpr}$), a state environment ($SE$) and a method environment ($ME$):

\[
CS \; = \; \mathbb{2}^{CExpr \times SE \times ME}
\]

A state environment is a partial function from class expressions (the case classes) to field environments ($FE$):

\[
SE \; = \; CExpr \rightarrow FE
\]

A field environment is a partial function from method identifiers ($MI$) to class expressions:

\[
FE \; = \; MI \rightarrow CExpr
\]

A method environment is a partial function from method identifiers ($MI$) to method specifications ($MS$):

\[
ME \; = \; MI \rightarrow MS
\]

A method specification is a triple with a method declaration ($MD$), and two formulae ($Form$) that represent the precondition and the postcondition:

\[
MS \; = \; MD \times Form \times Form
\]

A method declaration is a pair with a sequence of pairs of object variables ($OV$) and class expressions ($CExpr$) and a class expression:

\[
MD \; = \; (OV \times CExpr)^* \times CExpr
\]
3.2 Clay Type System

In this section we follow the standard formal technique used for describing type systems: a language for types is defined, the type environments are introduced,
type judgements are defined and, finally, the type inference rules are presented.

### 3.2.1 Types

Our representation of types are class expressions, that we will call object types, and message types. We extend the conventions introduced in figure 3.1 with the metavariable $M$ to represent message types:

\[
\begin{align*}
\text{Object Types} & \quad (T, T_1, T_2, \ldots) \quad \text{ObjTy} & \quad ::= & \quad A \\
\text{Message Types} & \quad (M) \quad \text{MsgTy} & \quad ::= & \quad A_1 A_2 \ldots A_n \rightarrow A
\end{align*}
\]

### 3.2.2 Type Environments

To describe the type system of Clay we start with the definition of type environments, a structure that relates classes, methods and parameters with their types. A type environment $\mathcal{E}$ in Clay is a tuple $(\Gamma, \gamma, \beta, \Lambda)$ where

- $\Gamma$ is a global environment that maps types $T$ to class environments $\gamma$,
- $\gamma$ is a class type environment, a structure with a set of super types ($\text{super}$) and a message type environment ($\text{menv}$) that relates message identifiers $m$ to message types $M$,
- $\beta$ is a bound environment that maps class variables $t$ to class expressions $A$, and
- $\Lambda$ is a local environment that maps object variables $x$ to object types $T$.

### 3.2.3 Type Judgements

The language for type judgements is given by the following cases:

- Under type environment $\mathcal{E}$, $A$ is a valid object type:
  \[ \mathcal{E} \vdash A \]
- Under type environment $\mathcal{E}$, $M$ is a valid message type:
  \[ \mathcal{E} \vdash M \]
- Under type environment $\mathcal{E}$, the type of the object expression $e$ is $A$:
  \[ \mathcal{E} \vdash e : A \]
3.2 Clay Type System

- Under type environment $\mathcal{E}$, object type $A$ is a subtype of $B$:
  \[ \mathcal{E} \vdash A <: B \]

- Under class type environment $\gamma$, the message type of the message identifier $m$ is $M$:
  \[ \gamma \vdash m : M \]

### 3.2.4 Type Rules

The type inference rules are presented in several fragments that capture the following information: well-formed types, subtyping, typing of object expressions and messages, typing of formulae.

**Well-formed Types**

- Object types are well-formed if they are in the domain of the global environment:
  \[ \mathcal{E} \vdash A \in \text{dom } \Gamma \]

- Message types are well-formed if involved object types are well-formed:
  \[ \mathcal{E} \vdash A \quad \mathcal{E} \vdash A_i \quad \mathcal{E} \vdash A_1 A_2 \ldots A_n \rightarrow A \quad i \in 1..n \]

- Every class identifier $C$ introduces a new class identifier Meta$C$:
  \[ \mathcal{E} \vdash \text{Meta} C < A_1, \ldots, A_n > \quad C < A_1, \ldots, A_n > \in \text{dom } \Gamma \]

- Type variables are well-formed types if they are in the domain of the bound environment:
  \[ \mathcal{E} \vdash t \quad t \in \text{dom } \beta \]

**Subtyping**

- Definition of subtypes:
\[ \frac{\mathcal{E} \vdash A \quad \mathcal{E} \vdash B}{\mathcal{E} \vdash A <: B} \quad A \in \text{super}(\Gamma A) \quad \frac{\mathcal{E} \vdash A}{\mathcal{E} \vdash t <: A} \quad t \rightarrow A \in \beta \]

- Reflexivity, transitivity and subsumption:

\[
\frac{\mathcal{E} \vdash A}{\mathcal{E} \vdash A <: A} \quad \frac{\mathcal{E} \vdash A <: B \quad \mathcal{E} \vdash B <: C}{\mathcal{E} \vdash A <: C} \quad \frac{\mathcal{E} \vdash e : A \quad \mathcal{E} \vdash A <: B}{\mathcal{E} \vdash e : B}
\]

- Message subtyping:

\[
\frac{\mathcal{E} \vdash B_1 <: A_1 \quad \ldots \quad \mathcal{E} \vdash B_n <: A_n \quad \mathcal{E} \vdash A <: B}{\mathcal{E} \vdash A_1 \ldots A_n \rightarrow A <: B_1 \ldots B_n \rightarrow B} \quad \frac{\Gamma A \vdash m : N \quad \mathcal{E} \vdash M <: N}{\Gamma A \vdash m : N}
\]

### Typing Object Expressions

\[
\frac{\mathcal{E} \vdash A}{\mathcal{E} \vdash x : A} \quad x \rightarrow A \in \Lambda \quad \frac{\mathcal{E} \vdash A : \text{Meta} C < A_1, \ldots, A_n >}{C < A_1, \ldots, A_n > \in \text{dom } \Gamma} \\
\frac{\mathcal{E} \vdash B_1 \quad \ldots \quad \mathcal{E} \vdash B_n \quad \mathcal{E} \vdash B}{\Gamma A \vdash m : B_1 \ldots B_n \rightarrow B} \quad m \rightarrow B_1 \ldots B_n \rightarrow B \in \text{menv}(\Gamma A) \\
\frac{\mathcal{E} \vdash e : A \quad \mathcal{E} \vdash e_1 : A_1 \quad \ldots \quad \mathcal{E} \vdash e_n : A_n \quad \Gamma A \vdash m : A_1 \ldots A_n \rightarrow A}{\mathcal{E} \vdash e \leftarrow m(e_1, \ldots e_n) : A}
\]

### 3.2.5 Typing Formulae

\[
\frac{\mathcal{E} \vdash e_1 : A_1 \quad \mathcal{E} \vdash e_2 : A_2 \quad \mathcal{E} \vdash A_1 <: B \quad \mathcal{E} \vdash A_2 <: B}{\mathcal{E} \vdash e_1 = e_2 : \text{Prop}} \\
\frac{\mathcal{E} \vdash F : \text{Prop}}{\mathcal{E} \vdash \ast F : \text{Prop}} \quad \frac{\mathcal{E} \vdash F : \text{Prop} \quad G : \text{Prop}}{\mathcal{E} \vdash F \ast G : \text{Prop}} \\
\frac{\mathcal{E} \vdash A \quad (\Gamma, \gamma, \beta, \Lambda \cup \{x \rightarrow A\}) \vdash F : \text{Prop}}{\mathcal{E} \vdash Qx : A(F) : \text{Prop}} \quad x \notin \text{dom } \Lambda
\]
Chapter 4

Clay semantics in First-order logic

This chapter presents the formal semantics of Clay. This semantics is given via an interlingua-based translation. The interlingua is a first-order language of the sub-sorted first-order logic. The symbols of the first-order language represent Clay concepts. Clay specifications are specifically translated into axioms and constant symbols of the first-order language. We have used the concrete syntax of Prover9 as the target language.

4.1 Methodology

Up to now we have been reasoning about Clay specifications in an informal way. Establishing a formal semantics it is needed as much to unambiguously understand and write specifications as to develop formal method tools for Clay. The main objectives of the Clay’s semantics are:

• To give a formal and unambiguous meaning to Clay specifications.

• To mechanise the reasoning about specifications.

One is tempted to use or to design an extremly powerfull, in the sense of expresiveness, logic, but a very relevant aspect has to be taken into account: Clay is a formal specification language and, as such, its specifications should be understood by different kinds of stakeholders, from customers to developers. There is
an important difference between the semantics one needs to supply for an specification language and that for a programming language. The latter is designed for experts: i.e. (advanced) programmers or automatic tools for safe program manipulation. On the other hand, an specification language need to be equipped with a more intuitive and natural semantics because its specifications are going to be read by non-experts, i.e. customers and ordinary developers.

It is often taught in introductory courses classes in Philosophy, Mathematics, and Computer Science that logic is the universal language of reasoning and rigorous representation of knowledge. This is not unfounded: for example, the entire body of mathematics can be formalized in classical first-order predicate logic (FOL).

Using first-order logic to give a formal semantics to Clay is, for us, twofolds a need and a challenge.

Interlingua-based Semantics

The methodology we have used to give a formal semantics to Clay is named interlingua-based translation. In [12], Van Baalen and Fikes describe a method for providing a declarative semantics for a new language in terms of its translation into a target language (the interlingua). To apply the method, the interlingua must have a declarative semantics and must include logical entailment and a set of top-level and satisfy the following definition.

Definition 4.1 (Interlingua-based semantics) Let $L$ be a language, $L_i$ be an interlingua language with a formally defined declarative semantics. $T_{\text{TRANS}}_{L,L_i}$ be a binary relation between top-level forms of $L$ and top-level forms of $L_i$, and $BT_{L,i}$ be a set of top-level forms in $L_i$. The pair $< T_{\text{TRANS}}_{L,L_i}, BT_{L,i} >$ is called an $L_i$-based semantics for $L$ when for every set $T_L$ of top-level forms in $L$, there os a set $T_{L_i}$ of top-level forms in $L_i$ such that

$$\forall s \in T_L \exists i \in T_{L_i} (T_{\text{TRANS}}_{L,L_i}(s, i))$$

$$\forall i \in T_{L_i} \exists s \in T_L (T_{\text{TRANS}}_{L,L_i}(s, i))$$

and the theory of $T_{L_i} \cup BT_{L,i}$ is equivalent to the theory represented by $T_L$.

Under this definition,

- $L$ is Clay,
4.2 The Logic of Clay

- the interlingua language $L_i$ is first-order logic, and
- $BT_L$ are the axioms that define the semantics of Clay that we call *The Clay Theory*.

When $<\text{TRANS}_{L_i}, BT_L>$ is used to define the semantics of $L$, as it is in our case, then *the theory represented by $T_L$* is equivalent to *the theory of $T_{L_i} \cup BT_L$* by definition.

In order to make the definition of the semantics of Clay easier to understand we introduce an intermediate language in the translation process. The result is a three steps description:

1. The definition of the Clay theory in a subsorted first-order logic that we call OOFOL (*object-oriented first-order logic*). The characteristics of the logic and the details of the first-order language used is given in Section 4.2. The Clay theory is then introduce and explained in Section 4.3.

2. The translation of Clay specifications in the form of abstract syntax trees into axioms of OOFOL (Section 4.4).

3. The encoding of OOFOL into untyped first-order logic following the orientations in [48]. The encoding is explained in a distributed manner throw the sections 4.2 and 4.3. We have used the syntax of Prover9 as a concrete syntax for first-order logic.

4.2 The Logic of Clay

To present the interlingua we will follow Gallier’s notation and style [52]. We start with the alphabet of our *object-oriented first-order logic* (OOFOL), a subsorted first-order logic:

- $S$ is the sets of sorts,
- $FS$ is the set of function symbols,
- $PS$ is the set of predicate symbols, and
- $r$ is the rank function.

4.2.1 Sorts and Subsorts

The set of sorts that we have designed is the following one:

$$S = \{\text{clsidS, msgidS, clsS, objS, msgS, clslstS, objlstS}\} \cup \{\text{anyS, boolS}\}$$
Intended Meaning

Maybe, the main decision is that we are not using the sorts in the logic to reflect the classes of Clay. Instead, classes are encoded as terms of sort clsS. We will discuss on this decision in Section 4.6. For the moment, let us describe every sort:

- clsidS groups constants that reflect class identifiers in Clay.
- msgidS groups constants that reflect message identifiers in Clay.
- clsS groups terms that reflect classes in Clay.
- objS groups terms that reflect objects in Clay. To reflect that every class is an object we establish that sort clsS is a subsort of objS.
- clslstS groups constants that reflect lists of classes.
- objlstS groups terms that reflect lists of objects.
- anyS is the sort that groups all the universe (used to type the equality predefined predicate in the logic).
- boolS is not properly a sort and we use it to characterise the syntactical family of formulae.

The reader can see that the subsorted part of the logic is used just to support the design decision that establishes that classes are objects in Clay.

Encoding in Prover9

To encode OOFOL (a subsorted first-order logic) in untyped first-order logic we follow the standard procedure described in [48]. The essential idea to convert a many-sorted language into a one-sorted language is to add domain predicate symbols $D_s$, one for each sort $s$, and to modify quantified formulas recursively as follows:

Every formula $A$ of the form $\forall_s x(B)$ (or $\exists_s x(B)$) is converted to the formula $A' = \forall x(D_s(x) \Rightarrow B')$, where $B'$ is the result of converting $B$.

We can see that the description of the encoding is a bit imprecise, in particular one can derives that the translation of $\exists_s x(B)$ would be $A' = \exists x(D_s(x) \Rightarrow B')$. Nevertheless, the meaning of the last formula is far from the meaning of $\exists_s x(B)$. The proper transformation and the one we have applied in this work is $A' = \exists x(D_s(x) \land B')$.

The concrete syntax for the untyped first-order language is the syntax for formulae in Prover9. The most relevant aspect of this syntax are:
• Variable symbols start with upper case.

• Function symbols start with lowercase.

• Free variables are considered universally quantified.

Let us go into the initial details of our encoding in Prover9. We start establishing that that sorts have no empty carriers and that they are disjoint except for clsS and objS (clsS is a subsort of objS):

\[ \exists C \text{ clsidS}(C). \]
\[ \exists M \text{ msgidS}(M). \]
\[ \exists O \text{ objS}(O). \]
\[ \exists C \text{ clsS}(C). \]
\[ \exists M \text{ msgS}(M). \]
\[ \exists CL \text{ clslstS}(CL). \]
\[ \exists OL \text{ objlstS}(OL). \]

\[ \text{clsidS}(X) \land \text{msgidS}(Y) \Rightarrow X \neq Y. \]
\[ \text{clsidS}(X) \land \text{clsS}(Y) \Rightarrow X \neq Y. \]
\[ \text{clsidS}(X) \land \text{objS}(Y) \Rightarrow X \neq Y. \]
\[ \text{clsidS}(X) \land \text{clslstS}(Y) \Rightarrow X \neq Y. \]
\[ \text{clslstS}(X) \land \text{objlstS}(Y) \Rightarrow X \neq Y. \]
\[ \text{msgidS}(X) \land \text{clsS}(Y) \Rightarrow X \neq Y. \]
\[ \text{msgidS}(X) \land \text{objS}(Y) \Rightarrow X \neq Y. \]
\[ \text{msgidS}(X) \land \text{msgS}(Y) \Rightarrow X \neq Y. \]
\[ \text{msgidS}(X) \land \text{clslstS}(Y) \Rightarrow X \neq Y. \]
\[ \text{msgidS}(X) \land \text{objlstS}(Y) \Rightarrow X \neq Y. \]
\[ \text{msgS}(X) \land \text{clsS}(Y) \Rightarrow X \neq Y. \]
\[ \text{msgS}(X) \land \text{clslstS}(Y) \Rightarrow X \neq Y. \]
\[ \text{msgS}(X) \land \text{objlstS}(Y) \Rightarrow X \neq Y. \]
\[ \text{clsS}(C) \Rightarrow \text{objS}(C). \]

Any element of the universe belongs to sort anyS:

\[ \text{anyS}(X). \]

\[ ^1 \text{We will activate the flag set(prolog_style_variables)} \]
4.2.2 Function Symbols

The set of function symbols is the following one:

\[ FS = \{c\text{MetaClass, }send, \text{msg, }cls, \text{ }\$\text{nil}_c, \text{ }\$\text{cons}_c, \text{ }\$\text{nil}_o, \text{ }\$\text{cons}_o\} \]

And the rank function on function symbols is defined as follow:

\[
\begin{align*}
  r(c\text{MetaClass}) &= (\epsilon, \text{clsidS}) \\
  r(\text{send}) &= (\text{objSmsgS, objS}) \\
  r(\text{msg}) &= (\text{msgidSobjlstS, msgidS}) \\
  r(\text{cls}) &= (\text{clsidS clslstS, clsS}) \\
  r(\$\text{nil}_c) &= (\epsilon, \text{clslstS}) \\
  r(\$\text{cons}_c) &= (\text{clsS clslstS, clslstS}) \\
  r(\$\text{nil}_o) &= (\epsilon, \text{objlstS}) \\
  r(\$\text{cons}_o) &= (\text{objSobjlstS, objlstS})
\end{align*}
\]

Intended Meaning

A Clay specification will not introduce any other function symbol but class identifiers and message identifiers with ranks \((\epsilon, \text{clsidS})\) and \((\epsilon, \text{msgidS})\) respectively (like Nat or add in Clay). With this pieces we build the following language:

- Classes are terms of the form \(\text{cls}(cid,cs)\) like \(\text{cls}(\text{cMetaClass,[]}),\) \(\text{cls}(\text{cNat,[]}\) or \(\text{cls}(\text{cList,}[\text{cls}(\text{Nat,[]})])\) representing \(\text{cMetaClass}, \text{Nat}\) and \(\text{List}<\text{Nat}>\), respectively\(^2\).

- Messages are terms of the form \(\text{msg(mid,os)}\) like \(\text{msg}(\text{mkZero,[]}\) or \(\text{msg}(\text{add,[]}\) representing \(\text{mkZero}\) and \(\text{addNat}(n)\), respectively (where \(m\) is the representation of \(n\)).

- Objects are terms of the form \(\text{send}(o,m)\) like \(\text{cls}(\text{Nat,[]})\leftarrow\text{msg}(\text{mkZero,[]}\) or \(n\leftarrow\text{msg}(\text{add,[]}\) representing \(\text{Nat}\leftarrow\text{mkZero}\) and \(n\leftarrow\text{addNat}(m)\), respectively.

\(^2\)We follow the standard notation for lists: [], [x], [x,y], etc.

\(^3\)For function symbol send we have introduced the infix version \(_\leftarrow\_\) so we will write \(o\leftarrow m\) also in the logic.
All function symbols represent injections except send so the following axioms are part of the theory:

\[\forall \text{clsid}\, Cid_1 \forall \text{clsid}\, Cid_2 \forall \text{clslst}\, CL_1 \forall \text{clslst}\, CL_2 (\text{cls}(Cid_1,CL_1) = \text{cls}(Cid_2,CL_2) \Rightarrow Cid_1 = Cid_2 \land CL_1 = CL_2)\]

\[\forall \text{msgid}\, Mid_1 \forall \text{msgid}\, Mid_2 \forall \text{objlst}\, OL_1 \forall \text{objlst}\, OL_2 (\text{msg}(Mid_1,OL_1) = \text{msg}(Mid_2,OL_2) \Rightarrow Mid_1 = Mid_2 \land OL_1 = OL_2)\]

\[\forall \text{obj}\, O_1 \forall \text{obj}\, O_2 \forall \text{objlst}\, OL_1 \forall \text{objlst}\, OL_2 (\text{cons}_o(O_1,OL_1) = \text{cons}_o(O_2,OL_2) \Rightarrow O_1 = O_2 \land Os_1 = Os_2)\]

\[\forall \text{cls}\, C_1 \forall \text{cls}\, C_2 \forall \text{clslst}\, CL_1 \forall \text{clslst}\, CL_2 (\text{cons}_c(C_1,CL_1) = \text{cons}_c(C_2,CL_2) \Rightarrow C_1 = C_2 \land Cs_1 = Cs_2)\]

**Encoding in Prover9**

The following axioms establish the rank of every function symbol, included the sorts of the lists:

\text{clsid}(\text{cMetaClass}).

\text{obj}(O) \land \text{msg}(M) \Rightarrow \text{obj}(O \leftarrow M).

\text{msgid}(\text{Mid}) \land \text{objlst}(\text{OL}) \Rightarrow \text{msg}(\text{msg}(\text{Mid},\text{OL})).

\text{clsid}(\text{Cid}) \land \text{clslst}(\text{CL}) \Rightarrow \text{cls}(\text{cls}(\text{Cid},\text{CL})).

\text{objlst}(\text{[]}).

\text{clslst}(\text{[]}).

\text{obj}(O) \land \text{objlst}(\text{OL}) \Rightarrow \text{objlst}(O:\text{OL}).

\text{cls}(C) \land \text{clslst}(\text{OL}) \Rightarrow \text{clslst}(C:\text{OL}).

The injection property for every symbol is encoded as follows:

\text{clsid}(\text{Cid}_1) \land \text{clslst}(\text{CL}_1) \land \text{clsid}(\text{Cid}_2) \land \text{clslst}(\text{CL}_2) \Rightarrow \text{cls}(\text{Cid}_1,\text{CL}_1) = \text{cls}(\text{Cid}_2,\text{CL}_2) \Rightarrow \text{Cid}_1 = \text{Cid}_2 \land \text{CL}_1 = \text{CL}_2.

---

4Prover9 has a predefined syntax for lists: [] (the empty list, $\text{nil}$) and the infix operator _:_ (the constructor of lists, $\text{cons}$).
msgidS(Mid1) \land objlstS(OL1) \land msgidS(Mid2) \land objlstS(CL2) \Rightarrow
msg(Mid1,OL1) = msg(Mid2,OL2) \Rightarrow Mid1 = Mid2 \land OL1 = OL2.

objS(O1) \land objlstS(Os1) \land objS(O2) \land objlstS(Os2) \Rightarrow
[O1:Os1] = [O2:Os2] \Rightarrow O1 = O2 \land Os1 = Os2.

clsS(C1) \land clslstS(Cs1) \land clsS(C2) \land clslstS(Cs2) \Rightarrow
[C1:Cs1] = [C2:Cs2] \Rightarrow C1 = C2 \land Cs1 = Cs2.

4.2.3 Predicate Symbols

The set of predicate symbols is the following one:

\[ PS = \{ \top, \bot, \_ = \_ \} \]
\[ \cup \{ wfo, wfc, subclass, instanceof, eq, eqs, pre, post \} \]
\[ \cup \{ instancesof, wfos, wfcs \} \]

And the rank function on predicate symbols is defined as follow:

\[
\begin{align*}
 r : & FS \cup PS \to S^* \times S \\
r(\top) & = (\epsilon, boolS) \\
r(\bot) & = (\epsilon, boolS) \\
r(\_ = \_ ) & = (anyS, anyS, boolS) \\
r(wfo) & = (objS, boolS) \\
r(wfc) & = (clsS, boolS) \\
r(subclass) & = (clsS, clsS, boolS) \\
r(instanceof) & = (objS, clsS, boolS) \\
r(eq) & = (clsS, objS, objS, boolS) \\
r(eqs) & = (clsS, objS, objS, boolS) \\
r(pre) & = (msgidS, clsS, clslstS, clsS, objS, objlstS, boolS) \\
r(post) & = (msgidS, clsS, clslstS, clsS, objS, objlstS, objS, boolS) \\
r(instanceof) & = (objlstS, clslstS, boolS) \\
r(wfos) & = (objlstS, boolS) \\
r(wfcs) & = (clslstS, boolS)
\end{align*}
\]
4.2 The Logic of Clay

Intended Meaning

A Clay specification will not introduce any other predicate symbol. Let us explain each one:

- $\top, \bot, \_=_$ are the standard symbols in the logic that represent truth, falsehood and equality in first order logic.
- Wfo and wfc characterise well formed objects and classes.
- Subclass is the subtyping relation ($<:$).
- instanceof is the relation that establish when an object is an instance of a class. objects and classes.
- Eq is the ternary equality predicate in Clay: $\text{eq}(C,O_1,O_2)$ establish that objects $O_1$ and $O_2$ are not distinguishable under the properties of class $C$.
- Eqs is a ternary predicate that captures the structural equivalence between two objects. In Section 4.4 we will show how facts of this predicate are generated for every class. The idea is to check that both objects match the same case class and that they are recursively equivalent. Predicate eq is defined in function of eqs and we will see its definition in this section.
- Pre is the predicate that contains the definition of the preconditions in the specifications: $\text{pre}(M,C,Cs,T,S,Os)$ is the precondition of method $M$ in the class $C$ for parameters of classes $Cs$ returning an object of class $T$ being $S$ the recipient of the message and $Os$ the arguments.
- Post is the predicate that contains the definition of the postconditions in the specifications: $\text{post}(M,C,Cs,T,S,Os,R)$ is the precondition of method $M$ in the class $C$ for parameters of classes $Cs$ returning an object of class $T$ being $S$ the recipient of the message and $Os$ the arguments and $R$ an instance of $T$ that fulfill the postcondition.
- Instancesof, wfos and wfc are the result of lifting predicates Instanceof, wfo and wfc to lists.

Encoding in Prover9

According to Enderton [48], the following axioms are not needed in the theory. Nevertheless they are not inconsistent with the rest of the theory and we consider them very informative since they capture the rank of the predicates:
wfo(O) ⇒ objS(O).

wfc(O) ⇒ clsS(O).

subclass(A, B) ⇒ clsS(A) ∧ clsS(B).

instanceof(O, C) ⇒ clsS(C) ∧ objS(O).

eq(C, O1, O2) ⇒ clsS(C) ∧ objS(O1) ∧ objS(O2).

eqs(C, O1, O2) ⇒ clsS(C) ∧ objS(O1) ∧ objS(O2).

pre(Mid, C, CL, RC, O, OL) ⇒
  msgidS(Mid)
  ∧ clsS(C) ∧ clslstS(CL) ∧ clsS(RC)
  ∧ objS(O) ∧ objlstS(OL).

post(Mid, C, CL, RC, O, OL, RO) ⇒
  msgidS(Mid)
  ∧ clsS(C) ∧ clslstS(CL) ∧ clsS(RC)
  ∧ objS(O) ∧ objlstS(OL) ∧ objS(OR).

instancesof(OL, CL) ⇒ objlstS(OL) ∧ clslstS(CL).

wfos(OL) ⇒ objlstS(OL).

wfcs(CL) ⇒ clslstS(CL).

4.3 The Clay Theory

Up to now, we have just introduce the pieces that capture the way in which Clay concepts in particular, and object oriented concepts in general, are represented in first order logic. In this section we give the semantics of Clay in the form of axioms directly in the untyped first-order logic. We present the axioms of the theory in a literated style ([75]) and following some indications of [77].

4.3.1 Well-formedness

- An object is well-formedness if and only if it is an instance of a well-formed class:
4.3 The Clay Theory

\[ \text{objS}(O) \Rightarrow (\text{wfo}(O) \iff \exists C (\text{clsS}(C) \land \text{wfc}(C) \land \text{instanceof}(O, C)) \). \]

- Lists of classes and lists of objects are well formed if all their elements are well-formed:

\[ \text{wfos}([], \text{objS}(O) \Rightarrow \text{objlstS}(OL) \Rightarrow (\text{wfos}([O:OL]) \iff \text{wfo}(O) \land \text{wfos}(OL) \). \]

\[ \text{wfcs}([], \text{clsS}(C) \Rightarrow \text{clslstS}(CL) \Rightarrow (\text{wfcs}([O:OL]) \iff \text{wfc}(C) \land \text{wfcs}(CL) \). \]

4.3.2 Instanceof

- Just well-formed objects and classes are involved in the predicate instanceof:

\[ \text{objS}(O) \Rightarrow \text{clsS}(C) \Rightarrow (\text{instanceof}(O,C) \Rightarrow \text{wfo}(O) \land \text{wfc}(C) \). \]

- Predicate instancesof checks if objects in the first argument are instances of classes in the second argument. One by one:

\[ \text{instancesof}([], [], \text{objS}(O) \Rightarrow \text{objlstS}(OL) \Rightarrow \text{clsS}(C) \Rightarrow \text{clslstS}(CL) \Rightarrow (\text{instancesof}([O:OL],[C:CL]) \iff \text{instanceof}(O,C) \land \text{instancesof}(OL,CL) \). \]

Instanceof and Types in Clay

The translation process of Clay specifications into the logic introduces the axioms that capture the type information of the language.
4.3.3 Subtyping

- Just well-formed classes are involved in the predicate subclass:
  \[
  \text{clsS}(A) \Rightarrow \text{clsS}(B) \Rightarrow \\
  (\text{subclass}(A, B) \Rightarrow \text{wfc}(A) \land \text{wfc}(B))
  \].

- Subclass is reflexive:
  \[
  \text{clsS}(A) \Rightarrow \\
  (\text{subclass}(A, A))
  \].

- Subclass is transitive:
  \[
  \text{clsS}(A) \Rightarrow \text{clsS}(B) \Rightarrow \text{clsS}(C) \Rightarrow \\
  (\text{subclass}(A, B) \land \text{subclass}(B, C) \Rightarrow \text{subclass}(A, C))
  \].

- Subsumption property establishes that any instance of a subclass is an instance of the class:
  \[
  \text{objS}(O) \Rightarrow \text{clsS}(A) \Rightarrow \text{clsS}(B) \Rightarrow \\
  (\text{instanceof}(O, A) \land \text{subclass}(A, B) \Rightarrow \text{instanceof}(O, B))
  \].

4.3.4 Equality

- Just well-formed classes and objects are involved in the predicate eq:
  \[
  \text{clsS}(C) \Rightarrow \text{objS}(X) \Rightarrow \text{objS}(Y) \Rightarrow \\
  (\text{eq}(C, X, Y) \Rightarrow \text{wfc}(C) \land \text{wfo}(X) \land \text{wfo}(Y))
  \].

- Eq is reflexive:
4.3 The Clay Theory

\[ \text{objS}(X) \Rightarrow \text{clsS}(C) \Rightarrow (\text{instanceof}(X, C) \Rightarrow \text{eq}(C, X, X)) \].

- Eq is symmetric:

\[ \text{objS}(X) \Rightarrow \text{objS}(Y) \Rightarrow \text{clsS}(C) \Rightarrow (\text{instanceof}(X, C) \land \text{instanceof}(Y, C) \land \text{eq}(C, X, Y) \Rightarrow \text{eq}(C, Y, X)) \].

- Eq is transitive:

\[ \text{objS}(X) \Rightarrow \text{objS}(Y) \Rightarrow \text{objS}(Z) \Rightarrow \text{clsS}(C) \Rightarrow (\text{instanceof}(X, C) \land \text{instanceof}(Y, C) \land \text{instanceof}(Z, C) \land \text{eq}(C, X, Y) \land \text{eq}(C, Y, Z) \Rightarrow \text{eq}(C, X, Z)) \].

- Two objects are equal under a given class if
  - both are instances of the class,
  - both are structurally equivalents (below we will talk about the structural equality), and
  - both are equal under any common subclass.

Let us show the axiom that capture this information:

\[ \text{objS}(X) \Rightarrow \text{objS}(Y) \Rightarrow \text{clsS}(C) \Rightarrow (\text{eq}(C, X, Y) \Leftrightarrow \text{instanceof}(X, C) \land \text{instanceof}(Y, C) \land \text{eqs}(C, X, Y) \land \forall D (\text{clsS}(D) \Rightarrow (\text{subclass}(C, D) \Rightarrow \text{eq}(D, X, Y)))) \].

- Two terms that represent objects are the same element of the carrier if the cannot be distinguish with the equality of any class:
objS(X) ⇒ objS(Y) ⇒
(  
  X = Y
  ⇔
  (  
    ∀ C (clsS(C) ⇒ ((instanceof(X, C) ⇔ instanceof(Y, C))
      ∧ (instanceof(X, C) ⇒ eq(C, X, Y))))
  )
).

**Structural Equality and Algebraic Types**

The translation process of Clay specifications into the logic introduces the axioms that capture the structural equality for classes. Predicate eqs is defined for every class to reflect the main properties of algebraic types.

### 4.3.5 Pre- and Post-conditions

- Preconditions are well-typed:
  
  \[ \text{msgidS(Mid)} \Rightarrow \text{clsS(C)} \Rightarrow \text{clslstS(CL)} \Rightarrow \text{clsS(RC)} \Rightarrow \text{objS(O)} \Rightarrow \text{objlstS(OL)} \Rightarrow \]
  
  (  
    \text{pre(Mid, C, CL, RC, O, OL)}
    ⇒
    \text{instanceof(O,C)}
    ∧ \text{instancesof(OL,CL)}
    ∧ \text{instanceof(O←msg(Mid,OL),RC)}
  ) .

- Postconditions are well-typed:
  
  \[ \text{msgidS(Mid)} \Rightarrow \text{clsS(C)} \Rightarrow \text{clslstS(CL)} \Rightarrow \text{clsS(RC)} \Rightarrow \text{objS(O)} \Rightarrow \text{objlstS(OL)} \Rightarrow \]
  
  (  
    \text{post(Mid, C, CL, RC, O, OL, R)} \Rightarrow \text{instanceof(O,C)}
    ∧ \text{instancesof(OL,CL)}
    ∧ \text{instanceof(R, RC)}
  ) .
4.4 TRANS\textsubscript{Clay,FOL}

In this section we have formalise the translation of Clay abstract syntax trees (see Chapter 3) into a set of axioms in the OOFOL.

4.4.1 Abstract Syntax for OOFOL

We have already mentioned that the translation process just add class identifiers and message identifiers to the OOFOL language, i.e. constants of sorts clsid\textsubscript{S} and msgid\textsubscript{S}. Our abstract structure for OOFOL theories (Theory) is the following Cartesian product:

\[
\text{Theory} = 2^{CI} \times 2^{MI} \times 2^{OOForm}
\]

For readability reasons, we give the definition of OOForm in the form of a con-
crete syntax:

\[
OOForm ::= \top \\
| \bot \\
| \text{wfo}(OOTerm_{\text{obj}s}) \\
| \text{wfc}(OOTerm_{\text{cls}s}) \\
| \text{subclass}(OOTerm_{\text{cls}s}, OOTerm_{\text{cls}s}) \\
| \text{instanceof}(OOTerm_{\text{obj}s}, OOTerm_{\text{cls}s}) \\
| \text{eq}(OOTerm_{\text{cls}s}, OOTerm_{\text{obj}s}, OOTerm_{\text{obj}s}) \\
| \text{eqs}(OOTerm_{\text{cls}s}, OOTerm_{\text{obj}s}, OOTerm_{\text{obj}s}) \\
| \text{pre}(OOTerm_{\text{msgid}s}, OOTerm_{\text{cls}s}, OOTerm_{\text{clslst}s}, OOTerm_{\text{cls}s}, \\
| \hspace{1cm} OOTerm_{\text{obj}s}, OOTerm_{\text{objlst}s}) \\
| \text{post}(OOTerm_{\text{msgid}s}, OOTerm_{\text{cls}s}, OOTerm_{\text{clslst}s}, OOTerm_{\text{cls}s}, \\
| \hspace{1cm} OOTerm_{\text{obj}s}, OOTerm_{\text{objlst}s}, OOTerm_{\text{obj}s}) \\
| \neg OOForm \\
| OOForm \land OOForm \\
| OOForm \lor OOForm \\
| OOForm \Rightarrow OOForm \\
| OOForm \Leftrightarrow OOForm \\
| \forall_{FS} OOVar(OOForm) \\
| \exists_{FS} OOVar(OOForm)
\]

where \( OOTerm_s \) is the set of terms of sort \( s \) and \( OOVar_s \) is the set of variables of sort \( s \).
4.4.2 Translation of \textit{Bounds} and \textit{Spec}

Clay specifications are represented by two partial functions: \textit{Bounds} and \textit{Spec} (see Section 3.1). The definition of the translation starts at that point:

\[
\text{TRANS}_{\text{Bounds} \times \text{Spec}} : \text{Bounds} \times \text{Spec} \rightarrow \text{Theory}
\]

\[
\text{TRANS}_{\text{Bounds} \times \text{Spec}} (b, s) = (\text{dom } b, m, f)
\]

where

\[
m = \bigcup_{c \in \text{dom } \text{Bounds}} (\text{TRANS}_{\text{OOForm} \times \text{CExpr} \times \text{CS}_o} \pi_1) (g, ce, cs)
\]

\[
f = \bigcup_{c \in \text{dom } \text{Bounds}} (\text{TRANS}_{\text{OOForm} \times \text{CExpr} \times \text{CS}_o} \pi_2) (g, ce, cs)
\]

\[
g = (\text{TRANS}_{\text{CI} \times \text{Bounds}} \circ \pi_1) (c, b)
\]

\[
ce = (\text{TRANS}_{\text{CI} \times \text{Bounds}} \circ \pi_2) (c, b)
\]

\[
cs = s ce
\]

Some comments with respect to the notation we will use in the definition of the translation function:

- \text{TRANS} is an indexed family of functions. We will overload the name of the family to name each function, in other words, in general we will omit the index since it is easy to derive.

- We will use concatenation to represent function application making extensive use of \textit{curryfication}.

- We will also use the infix operators \( \circ \) and \( \cdot \) as function composition with the following meaning: \( (f \circ g) x = g (f x) \) and \( (f \cdot g) x = f (g x) \).

- We will use ellipsis in the form \( e \ldots e' \) where \( e \) is an expression where index 1 is used and \( e' \) is the result of substituting 1 by a given expression \( n \) (that represent a natural number) in \( e \). The meaning is the replication of expression \( e \) by substituting 1 by natural numbers between 1 and \( n \) in that order. Sometimes separators are used, like “,” in \( e, \ldots, e' \), in this case the separator is not replicated in the last expression.

\[
\text{TRANS}_{\text{CI} \times \text{Bounds}} : \text{CI} \times \text{Bounds} \rightarrow \text{OOForm} \times \text{CExpr}
\]

\[
\text{TRANS} (c, b) = (g, cs)
\]

where

\[
g = \text{subclass}(\text{TRANS } x_1, \text{appTRANS}(bc)_1)
\]

\[
\wedge \ldots
\]

\[
\wedge \text{subclass}(\text{TRANS } x_n, \text{appTRANS}(bc)_n)
\]

\[
cs = c < x_1, \ldots, x_n >
\]

\[
x_1 \ldots x_n \text{ are different elements of } \text{CVar}
\]

\[
n = \text{length } (bc)
\]
4.4.3 Translation of class specifications (CS)

\[
\text{TRANS}_{OO\text{Form} \times CExpr \times CS} : \quad \text{OOForm} \times CExpr \times CS \rightarrow 2^\text{MI} \times 2^{OO\text{Form}}
\]

\[
\text{TRANS} \langle g, ce, cs \rangle = \langle \text{mis} \cup \text{mim}, \{\text{sub} \} \cup \text{fs} \cup \text{fm} \rangle
\]

where

\[
\text{mis} = (\text{TRANS}_9 \pi_1) \langle g, ce, se \rangle
\]

\[
\text{mim} = (\text{TRANS}_9 \pi_1) \langle g, ce, me \rangle
\]

\[
\text{fs} = (\text{TRANS}_9 \pi_2) \langle g, ce, se \rangle
\]

\[
\text{fm} = (\text{TRANS}_9 \pi_2) \langle g, ce, me \rangle
\]

\[
\langle s, se, me \rangle = \text{cs}
\]

\[
\text{sub} = g \Rightarrow \text{subclass}(\text{TRANS ce, TRANS } c_1)
\]

\[
\ldots
\]

\[
\ldots
\]

\[
\ldots
\]

\[
\{c_1, \ldots, c_n\} = s
\]

4.4.4 Translation of algebraic types (SE)

\[
\text{TRANS}_{OO\text{Form} \times CExpr \times SE} : \quad \text{OOForm} \times CExpr \times SE \rightarrow 2^\text{MI} \times 2^{OO\text{Form}}
\]

\[
\text{TRANS} \langle g, ce, se \rangle = \langle \text{mis}, \{f\} \cup \bigcup_{ce' \in \text{dom } se} \text{TRANS} \langle g, ce', se ce' \rangle \rangle
\]

where

\[
\text{mis} = \bigcup_{ce' \in \text{dom } se} \text{dom } (se ce')
\]

\[
f = g \Rightarrow \text{instanceof}(\text{Self, TRANS } ce) \Rightarrow \text{algineq}
\]

\[
\text{algineq} = \text{instanceof}(\text{Self, TRANS } c_1) \land \text{conj}_1
\]

\[
\lor \ldots
\]

\[
\land \ldots
\]

\[
\land \ldots
\]

\[
\land \ldots
\]

\[
\land \ldots
\]

\[
\{c_1, \ldots, c_n\} = \text{dom } se
\]

\[
\text{Self} \quad \text{is “fresh”}
\]
4.4.5 Translation of methods (\textit{ME})

\begin{align*}
\text{TRANS}_{\text{OOForm} \times \mathit{CExpr} \times \text{ME}} & : \quad \text{OOForm} \times \mathit{CExpr} \times \text{ME} \rightarrow 2^{\text{OOForm}} \\
\text{TRANS} \langle g, ce, me \rangle & = \langle \text{dom me}, \bigcup_{mi \in \text{dom me}} \text{TRANS} \langle g, ce, mi, mem \rangle \rangle
\end{align*}

\begin{align*}
\text{TRANS}_{\text{OOForm} \times \mathit{CExpr} \times \text{MI} \times \text{MS}} & : \quad \text{OOForm} \times \mathit{CExpr} \times \text{MI} \times \text{MS} \rightarrow 2^{\text{OOForm}} \\
\text{TRANS} \langle g, ce, mi, ms \rangle & = \{ ty, g \land gty \Rightarrow p, g \land gty \Rightarrow q \}
\end{align*}

\begin{align*}
\text{TRANS} \langle ty, gty, argtys, resty \rangle & = \text{TRANS} \langle g, ce, mi, md \rangle \\
\langle md, pre, post \rangle & = \text{ms} \\
p & = \text{pre} \left( \text{TRANS} \langle \text{mi}, \right. \\
& \quad \text{TRANS} \text{ ce, argtys, resty,} \\
& \quad \left. \text{TRANS} \text{ self} \rangle \right) \\
& \quad \Leftrightarrow \text{TRANS} \text{ pre} \\
q & = \text{post} \left( \text{TRANS} \langle \text{mi}, \right. \\
& \quad \text{TRANS} \text{ ce, argtys, resty,} \\
& \quad \left. \text{TRANS} \text{ self, TRANS result} \rangle \right) \\
& \quad \Leftrightarrow \text{TRANS} \text{ post}
\end{align*}
\[\text{TRANS}_{\text{OOForm} \times \text{CExpr} \times \text{MI} \times \text{MD}} : \text{OOForm} \times \text{CExpr} \times \text{MI} \times \text{MD} \rightarrow \text{OOForm}^2 \times \text{OOTerm}^*_{\text{clslstS}} \times \text{OOTerm}_{\text{clslstS}}\]

\[\text{TRANS} \langle g, ce, mi, md \rangle \quad \text{where} \quad
\]

\[\langle \text{args, cer} \rangle = \text{md} \]

\[\langle x_1, ce_1 \rangle \ldots \langle x_1, ce_1 \rangle = \text{args} \]

\[(\text{with } i \in \{1, \ldots, n\}) \quad \text{cet}_i = \text{TRANS} ce_i \]

\[ty = g \land \text{gty} \]

\[\Rightarrow \text{instance} (\text{TRANS self} \leftarrow m(x_1, \ldots, x_n), \text{TRANS cer}) \]

\[\text{gty} = \text{instance}(_\text{self}, \text{TRANS ce}) \]

\[\land \text{instance} (\text{TRANS} x_1, \text{TRANS} ce_1) \]

\[\land \text{instance} (\text{TRANS} x_2, \text{TRANS} ce_2) \]

\[\land \ldots \]

\[\land \text{instance} (\text{TRANS} x_n, \text{TRANS} ce_n)\]

### 4.4.6 Translation of formulae and expressions

\[\text{TRANS}_{\text{Form}} : \quad \text{Form} \rightarrow \text{OOForm} \]

\[\text{TRANS} e_1 = ce e_2 = \text{eqs} (\text{TRANS} ce, \text{TRANS} e_1, \text{TRANS} e_1) \]

\[\text{TRANS} e : ce = \text{instance} (\text{TRANS} e, \text{TRANS} ce) \]

\[\text{TRANS} \lnot f = \lnot \text{TRANS} f \]

\[\text{TRANS} f_1 \land f_2 = \text{TRANS} f_1 \land \text{TRANS} f_2 \]

\[\text{TRANS} f_1 \lor f_2 = \text{TRANS} f_1 \lor \text{TRANS} f_2 \]

\[\text{TRANS} f_1 \Rightarrow f_2 = \text{TRANS} f_1 \Rightarrow \text{TRANS} f_2 \]

\[\text{TRANS} f_1 \Leftrightarrow f_2 = \text{TRANS} f_1 \Leftrightarrow \text{TRANS} f_2 \]

\[\text{TRANS} \forall x : ce(f) = \forall \text{TRANS} x \ (\text{instance} (\text{TRANS} x, \text{TRANS} ce) \Rightarrow \text{TRANS} f) \]

\[\text{TRANS} \exists x : ce(f) = \exists \text{TRANS} x \ (\text{instance} (\text{TRANS} x, \text{TRANS} ce) \land \text{TRANS} f) \]

\[\text{TRANS}_{\text{CExpr}} : \quad \text{Expr} \rightarrow \text{OOTerm}_{\text{clslstS}} \]

\[\text{TRANS} ci < ce_1, \ldots, ce_n > = \text{cls} (\text{TRANS} ci, [\text{TRANS} ce_1, \ldots, \text{TRANS} ce_n]) \]

\[\text{TRANS} t = \text{TRANS}_{\text{CVar}} t \]

\[\text{TRANS}_{\text{Expr}} : \quad \text{Expr} \rightarrow \text{OOTerm}_{\text{objS}} \]

\[\text{TRANS} ce = \text{TRANS}_{\text{CExpr}} ce \]

\[\text{TRANS} x = \text{TRANS}_{\text{Var}} x \]

\[\text{TRANS} e \leftarrow m(e_1, \ldots, e_n) = \text{send} (\text{TRANS} e, \text{msg} (\text{TRANS} m, [\text{TRANS} e_1, \ldots, \text{TRANS} e_n])) \]

\[\text{TRANS}_{\text{CVar}} : \quad \text{CVar} \rightarrow \text{OOTerm}_{\text{clslstS}} \]

\[\text{TRANS} t = _t \]
4.5 Mechanised Reasoning

In this section we present some results directly obtained after feeding Prover9 and Mace4 with the Clay theory.

4.5.1 Consistency of the Clay Theory

In this moment, feeding Prover9 and Mace4 with the Clay theory results in an apparently endless execution. Since neither the prover nor the modelchecker finish their executions out confidence in the consistency of the theory is pretty high.

During the process of writing the semantics of the language, we have created inconsistent theories. Sometimes due to errors in the codification but other times because of the introduction of properties that, although intuitively looked correct resulted inconsistent.

4.5.2 Subject Reduction

As an example of the mechanisation of the reasoning that results after the encoding of Clay in Prover9, we would like to show you a theorem automatically proved. Subject reduction is a desirable property that establishes that if two expressions reduce to the same object then they must belong to the same classes. In our case we do not have a reduction relation so we have writing the property with the equality predicate:

\[
\forall X \forall Y \forall C \\
\{ \\
\text{objS}(X) \Rightarrow \text{objS}(Y) \Rightarrow \text{clsS}(C) \Rightarrow \\
\{ \\
\text{instanceof}(X, C) \land \text{eq}(C, X, Y) \Rightarrow \text{instanceof}(Y, C) \\
\}
\}
\]
Here is the resulting proof:

% Proof 1 at 0.07 (+ 0.00) seconds.
% Length of proof is 16.
% Level of proof is 3.
% Maximum clause weight is 13.000.
% Given clauses 150.

60 objS(X) -> objS(Y) -> clsS(C) -> (eq(C,X,Y) <-
    instanceof(X,C) & instanceof(Y,C) & eqs(C,X,Y) &
    (forall D (clsS(D) -> subclass(C,D) ->
    eq(D,X,Y)))) # label(non_clause). [assumption].

65 subject_reduction <-> (forall X forall Y forall C
    (objS(X) -> objS(Y) -> clsS(C) -> instanceof(X,C) &
    eq(C,X,Y) -> instanceof(Y,C))) # label(non_clause).
    [assumption].

66 subject_reduction # label(non_clause) # label(goal). [goal].

186 ~ objS(A) | ~ objS(B) | ~ clsS(C) | ~ eq(C,A,B) |
    instanceof(B,C). [clausify(60)].

194 subject_reduction | objS(c8). [clausify(65)].

195 subject_reduction | objS(c9). [clausify(65)].

196 subject_reduction | clsS(c10). [clausify(65)].

198 subject_reduction | eq(c10,c8,c9). [clausify(65)].

199 subject_reduction | ~ instanceof(c9,c10). [clausify(65)].

200 ~ subject_reduction. [deny(66)].

345 ~ instanceof(c9,c10). [back_unit_del(199),unit_del(a,200)].

346 eq(c10,c8,c9). [back_unit_del(198),unit_del(a,200)].

348 clsS(c10). [back_unit_del(196),unit_del(a,200)].

349 objS(c9). [back_unit_del(195),unit_del(a,200)].

350 objS(c8). [back_unit_del(194),unit_del(a,200)].

469 $F. [resolve(346,a,186,d),unit_del(a,350),
    unit_del(b,349),unit_del(c,348),unit_del(d,345)].

================================ end of proof ==========================

THEOREM PROVED

Exiting with 1 proof.

Process 8172 exit (max_proofs) Tue Jul 27 16:50:54 2010
4.6 Why Not an Object Oriented Typed Logic?

At first sight, the decision of not using a specifically designed logic that captures object oriented type concepts like classes and inheritance is surprising. During a year, we have been studying several logical frameworks in other object oriented formal specification languages. Our studies and approaches were on typed logics because we wanted for free the logic to manage all the typing issues. The results were disappointing, in particular because typed logics introduce, irremediably, information that are not dynamically usable. In other words, we have been unable to find a logic suitable to capture and introduce dynamic type information, i.e. type information actually usable during deduction.

VDM and Z have its object oriented versions: VDM++ ([120]) and Object-Z ([108]). VDM++ and Object-Z are ad-hoc extensions of VDM and Z that includes some object oriented features and during its design a lot of constraints was imposed by the original languages. Their underlying logics were unchanged and object oriented concepts are captured in their semantics by the axiomatisation and no new logics were designed. The resulting semantics is, at least, as complex as the original one, more complex anyway than FOL. The effect is that knowledge transfer and sharing between business and development is not effective enough.

Languages like CASL ([116]) or COLD ([71, 76]) are object oriented and their logics are claimed to be considered as object oriented logics in the sense that main object oriented concepts are directly supported. Nevertheless, a deeper analysis reveals that inheritance, for instance, are represented directly with subsorting and subsorting is interpreted as set inclusion in the model. An elegant solution if it were realistic: the meaning of a class ColoredNat inheriting from Nat has nothing to do with a subset of Nat. Equational languages based on equational logic like OBJ ([55]) or Maude ([83]) presents the same problem.

This interpretation is avoid in CASL [29] by introducing injections into subclasses and projections into superclasses, nevertheless, this does not actually avoid to understand inheritance as inclusion. Its main advantage is that restrictions on signatures are not so strict and that strong semantic relationships are established between overloading symbol names. This semantics restrictions are really interesting and we have included some of them through proof obligations in Clay.

A different approach is the use of an executable calculus like the Abady and Cardelli’s one [2]. Abadi and Cardelli’s calculus seems ideal for giving semantics to programming languages but the expresiveness is worse, in principle, than the expresiveness of a non-executable logic. Furthermore, Abadi and Cardelli work is based on structural typing while, from our point of view, type names and explicit relations between them are extremely important in specifications. The difficulty of precisely defining a type system name based does not justify not trying to manage
types by names.

As we have mentioned, our interlingua is a language of the subsorted first-order logic. Nevertheless, we want the product of this chapter to be really effective in the proof of the consistency of Clay specifications so we have decided to encode the language into untyped first-order using the concrete syntax of Prover9 [101][88].
Part III

The Clay System
Chapter 5

Synthesis of Logic Programs

In this chapter we define and study how to synthesise Prolog programs from Clay specifications.

- Definition of the subset of Clay formulae whose translation into first-order logic results in Horn-Clauses (logic program).

- Transformation of Clay data structures and *idioms* into more efficient logic programs.

- Proof that synthesised logic programs are correct and complete with respect to the semantics given in Chapter 4.

5.1 Interacting with Clay

The material presented in this chapter led to the implementation of a prototype that allows us to interact with formal specifications in Clay. In Section 5.4 we will check the actual performance of our prototype with some use cases that we describe below.
Inheritance

Recursive Specifications

Implicit Specifications

Requirements Validation

5.2 Translating Clay Specifications into Logic Programs

Given a Clay specification we will synthesise facts that represent its abstract syntax tree: classes, inheritance, case classes, fields, and method’s pre and post-conditions. In Section 5.3 we give a formalisation of the translation, for the moment Figure 5.1 contains the informal meaning of the target predicates.

The heart of our prototype is a common theory for all specifications: the Clay theory. The most important predicates of this axiomatisation are (instanceof/2, reduce/2, and eq/3), definitions that rely on the facts translated from the source specifications. Their meaning:

- Predicate `instanceof(NF, A)` is a generator of instances `NF` of a class `A`. `NF` is a normal form of an instance of `A`: a flexible representation as an incomplete data structure that we will study in Section 5.2.1.

- Predicate `eq(A, NF1, NF2)`, the Clay’s equality, decides if the representation of two instances are indistinguishable in class `A`.

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class(C)</code></td>
<td><code>C</code> is a Clay’s class identifier.</td>
</tr>
<tr>
<td><code>inherits(A, [B])</code></td>
<td><code>B</code> is the superclass of <code>A</code></td>
</tr>
<tr>
<td><code>cases(C, Cs)</code></td>
<td><code>Cs</code> is the list with cases classes of class <code>C</code></td>
</tr>
<tr>
<td><code>fields(C, Fs)</code></td>
<td><code>Fs</code> is the association list with the field names and field types of the case class <code>C</code></td>
</tr>
<tr>
<td><code>msgtype(C, M)</code></td>
<td><code>M</code> is a message identifier of a method defined or overridden in class <code>C</code></td>
</tr>
<tr>
<td><code>pre(C, S, M, As)</code></td>
<td>Precondition of sending message <code>M</code> with arguments <code>As</code> to instance <code>S</code> in class <code>C</code></td>
</tr>
<tr>
<td><code>post(C, S, M, As, R)</code></td>
<td>Postcondition that establishes that <code>R</code> is the resulting instance of sending message <code>M</code> with arguments <code>As</code> to instance <code>S</code> in class <code>C</code></td>
</tr>
</tbody>
</table>

Figure 5.1: Representing Clay in Prolog
• Finally, predicate \( \text{reduce}(E, NF) \) reduces any Clay object expression \( E \) to its normal form. Predicates \( \text{eq} \) and \( \text{reduce} \) will be study in Section 5.2.3.

5.2.1 Representing Clay Instances in Prolog

How will we represent the natural number 1 (an instance of \( \text{Nat} \))? We need to capture all the information of known superclasses (\( \text{Num} \)) and to capture all the information about the specific case class (\( \text{Nat} \)). We can use a list where each element contains the part of the representation for a given class of the instance: \((C, S, F)\) where \( C \) is the class, \( S \) is the particular case class, and \( F \) is an association list of field names to the representation of the instances of their classes. Let us show the representation of number 1:

\[
[(\text{'Num'}, \text{'Num'}, []), (\text{'Nat'}, \text{'Succ'}, [(\text{pred}, [(\text{'Num'}, \text{'Num'}, []), (\text{'Nat'}, \text{'Zero'}, [])])])]
\]

Under subtyping, during a deduction process where 1 is expected the successor of a \text{red 0} could appear. If we follow our rules, the representation of the coloured number would be:

\[
[(\text{'Num'}, \text{'Num'}, []), (\text{'Nat'}, \text{'Succ'}, [(\text{pred}, [(\text{'Num'}, \text{'Num'}, []), (\text{'Nat'}, \text{'Zero'}, []), (\text{'RGBNat'}, \text{'Red'}, [])])])]
\]

The representation of 0 and \text{red 0} are partially the same:

\[
[(\text{'Num'}, \text{'Num'}, []), (\text{'Nat'}, \text{'Zero'}, [])]
\]

\[
[(\text{'Num'}, \text{'Num'}, []), (\text{'Nat'}, \text{'Zero'}, []), (\text{'RGBNat'}, \text{'Red'}, [])]
\]

We propose to \textit{make room} for not yet known information of subclasses. Our proposal is to use an incomplete data structure where the incomplete part represents the \textit{room} for the information of potential subclasses of \( \text{Nat} \):

\[
[(\text{'Num'}, \text{'Num'}, []), (\text{'Nat'}, \text{'Zero'}, []), \_\text{Room}]
\]

\[
[(\text{'Num'}, \text{'Num'}, []), (\text{'Nat'}, \text{'Zero'}, []), (\text{'RGBNat'}, \text{'Red'}, []), \_\text{Room}]
\]

Apart from carrying all the information needed by methods specified in the superclasses, our normal form has the following properties:

• Information of the case classes allows us to reflect the disjoint sum (case classes) of products (fields).

• The incomplete part might be instantiated with data of an instance of a subclass (like \( \text{RGBNat} \)) during the deduction process. The most interesting benefit is that the instantiation can be implemented with the unification of our logic language engine. The example above shows how a \text{red 0 fits} in a 0.
5.2.2 Instance of

Predicate “:” (instance of) is translated into the Prolog predicate `instanceof/2`. `instanceof/2` generates the representation of all instances (first argument) of all classes (second argument) of a specification. Let us see some outputs of this predicate:

```prolog
?- instanceof(O,C).
C = 'Num', O = [('Num','Num',[])] ∨ ;
C = 'Bool', O = [('Bool','True',[])] ∨ ;
... C = 'RGBNat',
O = [('Num','Num',[]),('Nat','Zero',[]),('RGBNat','Red',[])] ∨ ;
C = 'RGBNat',
O = [('Num','Num',[]),('Nat','Zero',[]),('RGBNat','Green',[])] ∨ .
```

Thanks to our incomplete structures every instance of a subclass is an instance of a superclass, a technique that makes the desirable property of subsumption to be a theorem in our Prolog axiomatisation.

5.2.3 Equality

Clay equality (=) is the other predicate used in the atomic formulae of Clay in this work. Our translation of Clay equality into Prolog consists of two steps: a reduction to normal form of the objects expressions and the unification of the obtained representations. Let us see a description of the implementation of the reduction step and postpone the formalisation of the translation of the equality literals to section 5.3. `Reduce/2` relates terms that represent abstract syntax trees of Clay expressions with its normal form (assume :- `op(200,yfx,’<--’).` declared to emulate the Clay syntax):

```prolog
reduce(O<--M,NF) :- M =.. [Mid|Args],
reduce(O,ONF), reduceall(Args,ArgsNF),
knownclasses(ONF,Cs),
checkpreposts(Cs,ONF,Mid,ArgsNF,NF,defined).
```

`Reduceall/2` reduces a list of expressions, second argument of `knownclasses/2` contains the known classes (Cs) of the recipient of the message and `checkpreposts` checks pre and postconditions of every class of Cs in which method `Mid` is defined.

Safe Inheritance. We already mentioned in Chapter 2 the danger of overriding the properties of methods in subclasses: the practical impossibility of reasoning in large programs. The above implementation of predicate `reduce/2` will fail if any
5.3 The Translation, Formalised

Figure 5.2 shows the translation of Clay specifications, into Prolog programs (Prog), although extended programs (EG, EC, EP) allowing general first-order constructs in goals are used as an intermediate format and then translated into standard logic programs via a Lloyd-Topor like transform ([86, 85]). A concise abstract syntax of logic programs is given in Figure 5.2. The part of translation dealing with formulae and expressions is shown in Figure 5.4. Due to space limitations, the definitions of several auxiliary functions are given informally.

5.4 Experimental Results

Let us show some details of the code generated for the examples in our test set.

**Recursive definitions.** Our implementation of the Lloyd-Topor transformation applied to the postcondition of add produces four clauses:

```
post('Nat', _self, add, [_n], _result) :-
    instanceof(_result, 'Nat'),
    \+ instanceof(_self, 'Zero'),
```

postcondition in the inheritance hierarchy is inconsistent with the postconditions specified in superclasses.
\[ \text{tr\_spec} : \text{Spec} \rightarrow \text{Prog} \]
\[ \text{tr\_spec}[S] = \text{Clay\_Theory} \cup \bigcup_{A \in \text{domS}} \text{llloyd\_topor} \circ \text{tr\_class}[A, S(A)] \]

\[ \text{lloyd\_topor} : \text{EP} \rightarrow \text{Prog} \]
\[ \text{lloyd\_topor}[\text{ep}] = \text{The Lloyd-Topor Transformation} \quad (86, 85) \]

\[ \text{tr\_class} : \text{CI} \times \text{CS} \rightarrow \text{EP} \]
\[ \text{tr\_class}[A, (\text{super, states, methods})] = (\text{class}(\text{tr\_exp}[A]), T)) \]
\[ \cup \text{tr\_se}[A, \text{states}] \cup \text{tr\_me}[A, \text{methods}] \]

\[ \text{tr\_se} : \text{CI} \times \text{SE} \rightarrow \text{EP} \]
\[ \text{tr\_se}[A, \text{states}] = \{(\text{cases}(\text{tr\_exp}[A], \text{tr\_set}[\text{dom states}]), T))\}
\[ \cup \bigcup_{A \in \text{domS}} (\text{tr\_stt}[A, B, \text{states}(B)] \cup \text{tr\_fe}[A, B, \text{states}(B)]) \]

\[ \text{tr\_stt} : \text{CI} \times \text{CI} \times \text{FE} \rightarrow \text{EP} \]
\[ \text{tr\_stt}[A, B, \text{fields}] = \{(\text{msgtype}(\text{tr\_exp}[A], \text{mk} + \text{tr\_exp}[B]), T))\}
\[ \cup \text{tr\_fe}[A, B, \text{fields}] \]

\[ \text{tr\_fe} : \text{CI} \times \text{CI} \times \text{FE} \rightarrow \text{EP} \]
\[ \text{tr\_fe}[A, B, \text{fields}] = \bigcup_{m \in \text{dom fields}} \text{tr\_field}[A, B, \text{fields}(m), i, m] \]

\[ \text{tr\_field} : \text{CI} \times \text{CI} \times \text{CI} \times \text{MI} \times \mathbb{N} \rightarrow \text{EP} \]
\[ \text{tr\_field}[A, B, C, i, fn] = \{(\text{msgtype}(\text{tr\_exp}[A], \text{tr\_mi}[fn]), T))\}
\[ \cup \text{tr\_ms}[A, m, (\text{md, pre, post})] \]

\[ \text{tr\_ms} : \text{CI} \times \text{MI} \times \text{MS} \rightarrow \text{EP} \]
\[ \text{tr\_ms}[A, m, (\text{md, pre, post})] = \text{tr\_md}[A, m, md] \cup \]
\[ \{(\text{pre}(\text{tr\_exp}[A], _self, \text{tr\_mi}[m], \text{tr\_args}[md]), \text{tr\_form}[\text{pre}])\}
\[ \cup \{(\text{post}(\text{tr\_exp}[A], _self, \text{tr\_mi}[m], \text{tr\_args}[md], _result), \text{tr\_form}[\text{post}])\} \]

\[ \text{tr\_md} : \text{CI} \times \text{MD} \rightarrow \text{EP} \]
\[ \text{tr\_md}[A, m] = \{(\text{msgtype}(\text{tr\_exp}[A], \text{tr\_mi}[m]), T))\}

The Lloyd-Topor Transformation generates a valid Prolog term to represent a message identifier generates the class a given class A is instance of: metaA
5.4 Experimental Results

\[\text{tr\_form} : \text{Form} \rightarrow \text{EG}\]

\[\text{tr\_form}[e_1 = e_2] = \text{reduce}(\text{tr\_exp}[e_1], _\text{NF1}) \land \text{reduce}(\text{tr\_exp}[e_2], _\text{NF2}) \land \text{eq}(\text{tr\_exp}[A], _\text{NF1}, _\text{NF2})\]

where \(A\) is the minimum common type of \(e_1\) and \(e_2\) and \(_\text{NF1}\) and \(_\text{NF2}\) are new variables

\[\text{tr\_form}[e : A] = \text{reduce}(\text{tr\_exp}[e], _\text{NF}) \land \text{instanceof}(\_\text{NF}, \text{tr\_exp}[A])\]

where \(_\text{NF}\) is a new variable

\[\text{tr\_form}[\neg F] = \text{negate}(\text{tr\_form}[F])\]

\[\text{tr\_form}[F \ast G] = \text{tr\_form}[F] \ast \text{tr\_form}[G]\]

where \(\ast \in \{\land, \lor, \Rightarrow, \Leftrightarrow\}\)

\[\text{tr\_form}[\forall x : A(F)] = \forall \text{tr\_exp}[x](\text{tr\_form}[x : A] \Rightarrow \text{tr\_form}[F])\]

\[\text{tr\_form}[\exists x : A(F)] = \exists \text{tr\_exp}[x](\text{tr\_form}[x : A] \land \text{tr\_form}[F])\]

\[\text{tr\_exp} : \text{Expr} \rightarrow \text{Term}\]

\[\text{tr\_exp}[A] = \text{mk\_const}[A]\]

\[\text{tr\_exp}[x] = \text{mk\_var}[x]\]

\[\text{tr\_exp}[e \leftarrow m(e_1, \ldots, e_n)] = \text{send}(\text{tr\_exp}[e], \text{tr\_mi}[m](\text{tr\_exp}[e_1], \ldots, \text{tr\_exp}[e_n]))\]

\(\text{tr\_fv}\) generates a list with a variable per field

\(\text{tr\_fety}\) generates \text{instanceof} calls given a list of vars and a list of classes

\(\text{tr\_fenv}\) generates a partial function from field names to vars

\(\text{tr\_args}\) generates a list of vars per arg in a message

Figure 5.4: Translating Clay formulae and expressions
\+ instanceof(_self, 'Succ').
post('Nat', _self, add, [\_n], _result) :=
  \+ instanceof(_result, 'Nat'),
  \+ instanceof(_self, 'Zero'),
  reduce('Nat'<--mkSucc(_self<--pred<--add(_n)), _NF_Nat_mkSucc),
eq('Nat', _result, _NF_Nat_mkSucc).
post('Nat', _self, add, [\_n], _result) :=
  \+ instanceof(_self, 'Succ').
post('Nat', _self, add, [\_n], _result) :=
  \+ instanceof(_result, 'Nat'),
  eq('Nat', _result, _n),
  reduce('Nat'<--mkSucc(_self<--pred<--add(_n)), _NF_Nat_mkSucc),
eq('Nat', _result, _NF_Nat_mkSucc).

Since not enough intelligence has been imprinted, the first and the last clause are, respectively, the result of falsifying the antecedents of implications and checking the consequents. None of those clauses yield any new result in our case: self cannot be an instance of Zero and an instance of Succ and the last clause is establishing that result = n and result = self + n forcing self to be 0 and repeating the results of third clause. The result of some messages:

?- reduce('Nat'<--mkZero,_Zero), reduce('Nat'<--mkSucc(_Zero),_One),
   reduce(_Zero<--add(_One),_One), reduce(_One<--add(_Zero),_One),
   reduce('Nat'<--mkSucc(_One),_Two), reduce(_Two<--add(Two),_Four).
Zero = Zero{},
One = Succ{pred : Zero{}},
Two = Succ{pred : Succ{pred : Zero{}}}
Four = Succ{pred : Succ{pred : Succ{pred : Succ{pred : Zero{}}}}}?

Our next example is the translation of half. For the moment we can see that the synthesised code is structurally the same that the specified in Clay:

post('Nat', _self, half, [], _result) :=
  \+ instanceof(_result, 'Nat'),
  reduce(_result<--add(_result), _NF_result_add),
eq('Nat', _self, _NF_result_add).

Will our prototype find the resulting instance of \!!42\! \leftarrow \text{half}\!? And, what about \!!27\! \leftarrow \text{half}\!?

?- num2nat(42,_E42), num2nat(21,_E21),
   reduce(_E42<--half,NF21), reduce(_E21,NF21).
NF21 = [(\!'Num',\!'Num',[]),(\!'Nat','Succ',[(\!pred,[])...])]?

\(^1\)Prolog terms that represent normal forms are presented in a more readable format.
5.4 Experimental Results

Yes
?- num2nat(27,_E27), reduce(_E27<--half,_).
no

Yes\textsuperscript{2} is the answer to 42--half with variable NF21 representing the normal form of our natural number 21. Since precondition of 27--half (27--even : True) does not hold our prototype's answer is no.

Equality. Let us show the answers to queries about the equality of several coloured and non-coloured naturals:

?- reduce('RGBNat'<--mkZeroRed,ZR),
   reduce('CMYNat'<--mkZeroCyan,ZC), eq('Nat',ZR,ZC).
ZR = [('Num','Num',[]),('Nat','Zero',[]),('RGBNat','Red',[])|_],
ZC = [('Num','Num',[]),('Nat','Zero',[]),('CMYNat','Cyan',[])|_] ?

?- reduce('Nat'<--mkZero,Z)
   reduce('CMYNat'<--mkZeroCyan,ZC), eq('Nat',Z,ZC).
Z = [('Num','Num',[]),('Nat','Zero',[])|_],
ZC = [('Num','Num',[]),('Nat','Zero',[]),('CMYNat','Cyan',[])|_] ?

Overriding. We show now the effects of the safe inheritance:

?- reduce('Cell'<--mkCellCase(0),R).
R = 'CellCase'{contents : 'Zero'{}}
?- reduce('Cell'<--mkCellCase(0)<--set('Nat'<--mkSucc(0)),R).
R = 'CellCase'{contents : 'Succ'{pred : 'Zero'{}},}
?- reduce('Cell'<--mkCellCase(0)<--set(1)<--get,R).
R = 'Succ'{pred : 'Zero'{}},
?- reduce('ReCell'<--mkReCellCase(0)<--set(1)<--get,R).
R = 'Succ'{pred : 'Zero'{}},
?- reduce('ReCell'<--mkReCellCase(0)<--set(1)<--restore<--get,R).
R = 'Zero'{}
?- reduce('Cell'<--mkCellCase(0)<--set(1)<--restore<--get,R).
no

Performance. We finish the presentation of our results with a pair of performance figures. Our experiments have been produced in a Ubuntu box running GNU/Linux 2.6.31-21 SMP on a machine with an Intel Dual Core CPU T7200@2.00GHz, 4096KB of cache and 2 GB of RAM. Our Prolog engine is Ciao 1.13.0-11293. In the following table we show the performance of our prototypes. Column \textit{depth} indicates the limit in the depth for the iterative deepening strategy for predicate instanceof.

\textsuperscript{2}Predicate \texttt{num2nat/2} translates Prolog positive integers into Clay abstract syntax trees that represent such numbers.
### 5.5 Why does projection (upto) + unification is not enough?

Example with `eq_uppto` with unification in the case classes:

```
?- reduce(angel_CMYNat<--mkZeroCyan,X), reduce(angel_RGBNat<--mkZeroRed,Y),
   eq(clay_lang_Nat,X,Y).
X = [(clay_lang_Num,clay_lang_Num,[]), (clay_lang_Nat,clay_lang_Zero,[]),
    (angel_CMYNat,angel_Cyan,[])]
Y = [(clay_lang_Num,clay_lang_Num,[]), (clay_lang_Nat,clay_lang_Zero,[]),
    (angel_RGBNat,angel_Red,[])]
yes
?- reduce(angel_CMYNat<--mkZeroCyan,ZC), reduce(angel_RGBNat<--mkZeroRed,ZR),
   reduce(clay_lang_Nat<--mkSucc(ZC),OC),
   reduce(clay_lang_Nat<--mkSucc(ZR),OR),
   eq(clay_lang_Nat,OC,OR).
no
?- 
```

The first answer is what we expect.

The second answer is not what we expect: under Nat, OR and OC should be equal since they are not distinguishable.
Chapter 6

The Clay Compiler

To show the validity of our Proof-of-Concept we have built a compiler for Clay. The compiler that we present in this chapter translates Clay specifications into first-order logic, in particular to the concrete language of Prover9, and synthesises prototypes in Prolog. Our implementation has been crucial in the development of the theoretical material of this thesis. Implemented in Haskell, a methodological approach [67] and state-of-the-art techniques of functional programming has been applied in its construction.

6.1 Architecture of the Compiler

To design the architecture of our compiler we have applied typical modularisation of compilers following orientations from [8], from [67] and from state-of-the-art of Haskell techniques. At first level we have two components:

Front-end. This component *knows* about the static semantics of Clay (concrete syntax, abstract syntax and type system described in 3). In Figure 6.1 we can appreciate that Clay specifications are parsed and then transformed into abstract representations in the form of abstract syntax and environments that are type-checked.

Back-ends. After the typechecking stage we find two back-ends:

- The first one implements the translation of environments of Clay into the abstract syntax of OOFOL as described in Chapter 4 Figure 6.2 shows the internal components.
6.2 More than Parsing

In the implementation of the Clay compiler we have applied our own techniques described in [67]:

- Implemented abstract syntax have been derived from the concrete syntax.

- Concrete syntax has been transformed in a LALR(1) grammar to automatically generate a parser with the tool Happy.

- Semantic actions just create the abstract syntax tree.

- Abstract syntax trees are traversed in order to construct the environments.

We illustrate the application of our techniques with a pair of examples of productions of the concrete syntax of Clay: class specifications and object expressions.
6.2 More than Parsing

Figure 6.2: Clay Compiler Architecture: Back-end to FOL

Figure 6.3: Clay Compiler Architecture: Back-end to Prolog
6.2.1 Class Specifications

A class specification consists of a class declaration, optionally a class invariant, a possibly empty sequence of case classes declarations (state declarations) and a possibly empty sequence of method specifications. Its GONF (Generalised Object-Oriented Normal Form, [67]) production is the following one:

\[
\text{class} \_ \text{spec} ::= \text{class} \_ \text{decl} \{ \\
\quad \text{invariant}^\circ \\
\quad \text{state} \_ \text{decl}^* \\
\quad \text{method} \_ \text{spec}^* \\
\}
\]

In the definition of the abstract syntax constant lexemes like '{' and '}' are ignored and the Haskell type that represent class specifications is the following one:

\[
\text{data ClassSpec} = \text{ClassSpec} \\
\quad \text{ClassDecl} \\
\quad (\text{Maybe Invariant}) \\
\quad [\text{StateDecl}] \\
\quad [\text{MethodSpec}]
\]

After the application of some straightforward rules we obtain the following productions in Happy, the most popular parser generator in Haskell:

\[
\text{class}_\text{spec} :: { \text{ClassSpec} } \\
\text{class}_\text{spec} : \text{class}_\text{decl} '{' \\
\quad \text{invariant}_\text{opt} \\
\quad \text{state}_\text{decl}_\text{seq}_0 \\
\quad \text{method}_\text{spec}_\text{seq}_0 \\
\quad '}' \\
\quad { \text{ClassSpec} $1$ $3$ $4$ $5$ }
\]

\[
\text{invariant}_\text{opt} :: { \text{Maybe Invariant} } \\
\text{invariant}_\text{opt} : {\text{- empty -}} \\
\quad \{ \text{Nothing} \} \\
\quad | \text{invariant} \\
\quad \quad \{ \text{Just$1$} \}
\]

\[
\text{state}_\text{decl}_\text{seq}_0 :: { [\text{StateDecl}] } \\
\text{state}_\text{decl}_\text{seq}_0 : {\text{- empty -}} \\
\quad \{ [] \} \\
\quad | \text{state}_\text{decl}_\text{seq}_0 \text{ state}_\text{decl} \\
\quad \quad \{ $2:1$ \}
\]

\[
\text{method}_\text{spec}_\text{seq}_0 :: { [\text{MethodSpec}] }
\]
method_spec_seq0 : { empty }
  { [] } 
  | method_spec_seq0 method_spec 
  { $2:$1 }

As we mentioned, semantic actions just create the appropriate abstract syntax nodes.

### 6.2.2 Object Expressions

Object expressions consists of four alternatives: a class expressions, a variable, a message sent and a sugared expression. Its GONF is the following one:

\[
\text{obj_expr ::= class_expr} \\
| \text{var_id} \\
| \text{sent_expr} \\
| \text{sugared_expr}
\]

\[
\text{sent_expr ::= obj_expr\rightarrow msg_expr}
\]

\[
\text{sugared_expr ::= bin_expr} \\
| \text{una_expr} \\
| \text{num_expr}
\]

In the derivation of the abstract syntax from the concrete syntax we have applied the directive \textit{collapse [67]} which leads to a \textit{flatten} hierarchy of the alternatives. Binary and unary expressions are sugared versions of sent expressions. The resulting type in Haskell is the following one:

\[
data \text{ObjExpr} = \text{Class ClsExpr} \\
\text{(Maybe ObjType)} \\
| \text{Var VarId} \\
\text{(Maybe ObjType)} \\
| \text{Send ObjExpr MsgExpr} \\
\text{(Maybe ObjType)} \\
| \text{IntExpr IntLit} \\
\text{(Maybe ObjType)} \\
| \text{DecExpr DecLit} \\
\text{(Maybe ObjType)} \\
\text{deriving (Data, Typeable)}
\]

Observe that, for each case, a semantic value is added: \textit{Maybe ObjType}. Some abstract syntax nodes have been decorated with semantic values that are computed by the successive stages in the compiler. In this case, each object expression will be annotated with its most precise type during the typechecking phase.
Let us show the resulting productions in Happy:

\[
\begin{align*}
\text{obj_expr} &:: \{ \text{ObjExpr} \} \\
\text{obj_expr} &: \text{cls_expr} \\
&\quad \{ \text{Class $1 \textbf{Nothing}$} \} \\
&\quad | \text{var_id} \\
&\quad \{ \text{Var $1 \textbf{Nothing}$} \} \\
&\quad | \text{obj_expr} \texttt{\textasciitilde \textasciitilde} \text{msg_expr} \\
&\quad \{ \text{Send $1 \texttt{\textasciitilde \textasciitilde} \texttt{\textasciitilde \textasciitilde} \textbf{Nothing}$} \} \\
&\quad | \text{sugared_expr} \\
&\quad \{ $1 \} \\
\text{sugared_expr} &:: \{ \text{ObjExpr} \} \\
\text{sugared_expr} &: \text{bin_expr} \\
&\quad \{ $1 \} \\
&\quad | \text{una_expr} \\
&\quad \{ $1 \} \\
&\quad | \text{num_expr} \\
&\quad \{ $1 \}
\end{align*}
\]

### 6.3 Environments

To allow an efficient access to the information in the abstract syntax tree and to avoid the duplication of some computations we introduce a more abstract representation of the Clay specifications in the form of environments. We explore, briefly, the definition of our environments and the techniques applied in its construction.

#### 6.3.1 The Environment Definition

Data type `GlbEnv` represents a symbol table in the form of an association list from class identifiers to (polymorphic) class environments:

```haskell
type GlbEnv = [(ClsId,PolyClsEnv)]
```

```haskell
data PolyClsEnv = PolyClsEnv { bndEnv :: BndEnv , clsEnv :: ClsEnv }
```

The `bndEnv` (of type `[(VarId,[ObjType])]`) is a bound environment that associates class variables to their bounds. The `clsEnv` field, of type

```haskell
data ClsEnv = ClsEnv { clsId :: ClsId , isCaseCls :: Bool }
```
6.3 Environments

```haskell
, superTys :: [ObjType]
, clsInv :: Formula
, sttEnv :: SttEnv
, msgEnv :: MsgEnv }

deriving (Data, Typeable)
```

is the heart of the Clay environments and its fields represent the following information:

- `clsId` is the class identifier.
- `isCaseCls` establishes if the class is a case class.
- `superTys` contains the supertypes of the class.
- `clsInv` represents the class invariant.
- `sttEnv` represents the cases classes of the class and their definitions.
- `msgEnv` represents the specification of the methods specified in the class.

### 6.3.2 The Environment Construction

The construction of the global environment follows a monadic approach to introduce arbitrary control flows in the incremental construction of the environment.

Every node of the abstract syntax that modify a environment will be declared as an instance of class `EnvModifier`.

```haskell
class EnvModifier env node where
  modify :: Monad m => env -> node -> m env
```

Let us show the precise instantiation for the `ClassSpec`:

```haskell
instance EnvModifier Env ClassSpec where
  modify env (ClassSpec cd inv sds mss) =
    do env <- setCurClsEnv env emptyClsEnv
       env <- setCurBndEnv env emptyBndEnv
       env <- modify env cd
       env <- modify env inv
       env <- pushClsInv env
       env <- modify env sds
       env <- modify env mss
    return env
```

The type we attach to the calculation of environments is
data Env = Env {
  glbEnv :: GlbEnv,
  toLoad :: [ClsId],
  curModId :: Maybe ModId,
  curImps :: [ClsId],
  curClsId :: Maybe ClsId,
  curBndEnv :: Maybe BndEnv,
  curClsEnv :: Maybe ClsEnv,
  curSttInfo :: Maybe StttInfo,
  curMsgInfo :: Maybe MsgInfo,
  curLocEnv :: Maybe LocEnv,
  curFormula :: Maybe Formula
}

Env contains, mainly, a global environment and intermediate information that are being constructed:

- toLoad is the list of classes to be loaded.
- curModId is the module of the class which environment is under construction (if any).
- curImps is the current imported classes.
- curClsId is the current class identifier (if any).
- curBndEnv is the current environment with the formal type parameters of the current class (if any).
- curClsEnv is the current class environment under construction (if any).
- curSttInfo is the current case class (state) being traversed (if any).
- curMsgInfo is the information of the current method being traversed (if any).
- curLocEnv represents the local parameters of the node being traversed (if any).
- curFormula contains a representative formula of the last traversed node (if any).

Now we can understand the line `env <- pushClsInv env`: monadic function `pushClsInv` pushes the current formula (curFormula) in the class invariant field (clsInv) of the current class environment (curClsEnv).

During the construction of the internal environments these are traversed by a generic function that qualify every class identifier with the current module in the environment. Here we make use of Haskell SyB (scrap your boilerplate) technology:

```haskell
qualifyClsIds :: (Monad m, Data a) => Env -> a -> m a
qualifyClsIds env = everywhereM (mkM (qualifyWrt (curImps env)))
```
6.4 Type-checking

With all the environment constructed the type checking stage is not very difficult to implement: all environments are traversed until object expressions and then object expressions are annotated with their types and the consistency of those types are checked with respect to the typing rules of chapter 3.

The most important functions in the type checker are `tcObjExpr` and `tcAtomicFormula` (in particular the rule for the equality predicate). Let us show the implementation of both.

```haskell
tcAtomicFormula :: Monad m => Env -> AtomicFormula -> m AtomicFormula
tcAtomicFormula env (Equal oe1 oe2 _) =
  do tcoe1 <- tcObjExpr env oe1
     tcoe2 <- tcObjExpr env oe2
     case decoration tcoe1 of
        Nothing ->
          return $ Equal tcoe1 tcoe2 Nothing
        Just ty1 ->
          case decoration tcoe2 of
            Nothing ->
              return $ Equal tcoe1 tcoe2 Nothing
            Just ty2 ->
              do ty <- minCommonSupertype env ty1 ty2
                 return $ Equal tcoe1 tcoe2 (Just ty)
```

The most relevant code is in the last but one line where function `minCommonSupertype` implements the typing rule for equality.

```haskell
tcObjExpr :: Monad m => Env -> ObjExpr -> m ObjExpr
tcObjExpr _env e@(Class ce _type) =
  do return $ Class ce (Just [metaType [objTypeFromClsExpr ce]])

tcObjExpr env e@(Var vid _type) =
  do lenv <- getCurLocEnv env
     case lookup vid lenv of
        Nothing -> do benv <- getCurBndEnv env
                      case lookup vid benv of
                        Nothing -> fail $ "Variable not found: '\''
                                      ++ show vid
                                      ++ ",'\''
                                      ++ (show env)
                        Just _ -> return $ Var vid (Just $ TVar vid)
        Just ot -> return $ Var vid (Just ot)

tcObjExpr env e@(Send oe me _type) =
  do tcoe <- tcObjExpr env oe
     let Just ot = decoration tcoe
```

In this case the most relevant code is also in the last but one line where function checkResultType implements the typing rule for the syntax of sending messages.

### 6.5 Translations

Both back-ends follow the same architecture: given the annotated environments result of the front-end a translation process generates an abstract syntax tree that is encoded as sentences of the concrete language of the back-end.

#### 6.5.1 Translation into Prover9

**Abstract Syntax of OOFOL**

The logic presented in Chapter 4 is represented in Haskell using phantom types to capture the sort information of terms. We start with the introduction of the sorts:

```haskell
data ClsIdS
data ClsS
data MsgIdS
data MsgS
data ObjS

class Sort s where
    ...

instance Sort ClsIdS where
    ...
instance ObjSort ClsS where
    ...
instance Sort MsgIdS where
    ...
instance Sort MsgS where
    ...
instance ObjSort ObjS where
    ...
```

Typed terms are then defined with the type

```haskell
data Sort s => TypedTerm s = TT Term
```
6.5 Translations

The class `Sort` define the creation of variables and the projection of the sort name:

```haskell
class Sort s where
  var :: String -> TypedTerm s
  var ident = TT (Var ident)
  sortName :: TypedTerm s -> String
```

With this infrastructure we have an abstract syntax that do not allow bad-formed OOFOL formulae. The type that represent these formulae is

```haskell
data Formula = Top
  | Bot
  | Wfo (TypedTerm ObjS)
  | Wfc (TypedTerm ClsS)
  | Subclass (TypedTerm ClsS) (TypedTerm ClsS)
  | Instanceof (TypedTerm ObjS) (TypedTerm ClsS)
  | Eq (TypedTerm ClsS) (TypedTerm ObjS) (TypedTerm ObjS)
  | Pre (TypedTerm MsgIdS)
    (TypedTerm ClsS)
    [(TypedTerm ClsS)]
    (TypedTerm ClsS)
    (TypedTerm ObjS)
    [(TypedTerm ObjS)]
  | Post (TypedTerm MsgIdS)
    (TypedTerm ClsS)
    [(TypedTerm ClsS)]
    (TypedTerm ClsS)
    (TypedTerm ObjS)
    [(TypedTerm ObjS)]
    (TypedTerm ObjS)
  | Neg Formula
  | Conj Formula Formula
  | Disj Formula Formula
  | Impl Formula Formula
  | Equiv Formula Formula
  | forall s . Sort s => Forall (TypedTerm s) Formula
  | forall s . Sort s => Exists (TypedTerm s) Formula
```
Translation of Clay into OOFOL

The implementation of the translation function formalised in Chapter 4 is the Haskell function `trans` of the type class `Trans`:

```haskell
class Trans node fol where
    trans :: node -> fol
```

Let us show a pair of examples of instantiation of the class for `AtomicFormula` and `ObjExpr`.

```haskell
instance Trans AtomicFormula Log.Formula where
    trans (Equal x y oty) = Eq (trans oty) (trans x) (trans y)
    trans (InstanceOf o c) = Instanceof (trans o) (trans c)
    ...

instance Trans ObjExpr (TypedTerm ObjS) where
    trans (Class ce _type) = trans ce
    trans (Var v _type) = trans v
    trans (Send o m _type) = transSend (trans o) (trans m)
```

Encoding of OOFOL in Prover9

Encoding OOFOL formulae in FOL is more or less straightforward. We have added a class that introduce a `show9` function that translate a given construction into the syntax of Prover9:

```haskell
class ShowAsProver9 a where
    show9 :: a -> String
    show9asTree :: Int -> a -> String
```

Then, `TypedTerm` and `Formula` are instances of `ShowAsProver9`. The most relevant portion of code is the translation of quantifiers following the Enderton [48] indications:

```haskell
instance ShowAsProver9 Formula where
    show9 Top = "$T"
    show9 Bot = "$F"
    ...
    show9 (Forall v f) =
        "(forall $v)" ++ show9 v
        ++ "(" ++ whichSort v
        ++ "->" ++ show9 f ++ ")"
```
show9 (Exists v f) =
  "(exists_\_" ++ show9 v
  ++ "(" ++ whichSort v
  ++ ")_\_\_"
  ++ show9 f ++ ")"
  ++ ")"

6.5.2 Synthesis of Prolog Programs

Abstract Syntax of Extended Programs

The Lloyd-Topor transformation take arbitrary first order formulas in the body of clauses, and transform this into general programs, programs where single negation is allowed in front of each atom in the body of a clause. We have introduced one abstract grammar for extended programs and other one for general programs:

newtype EP = EP [EPC]
data EPC = EPC Atom [Literal] [Formula]

newtype GP = GP [GPC]
data GPC = GPC Atom [Literal]

The meaning of \( h \ a \ b \rightarrow h \) (where \( a \) and \( b \) are interpreted as the conjunction of the formulae in \( a \) and \( b \)).

The definition of the types Atom, Literal and Formula are

data Atom = Atom String [Term]

data Term = Var String
  | Structure String [Term]
  | List [Term]
  | Tuple [Term]
  | Integer Integer

data Literal = Pos Atom
  | Neg Atom

data Formula = Top
  | Bot
  | At Atom
  | Not Formula
  | Conj Formula Formula
  | Disj Formula Formula
Translation of Clay into General Programs

We have applied the same approach we used in the back-end to first-order logic to implement the translation function of Chapter 5:

```haskell
class Trans node fol where
  trans :: node -> fol
```

Let us show a pair of examples of instantiation of the class for AtomicFormula and ObjExpr.

```haskell
instance Trans AtomicFormula Prolog.Formula where
  trans (Equal x y oty) =
    let (redX, xTr) = flatSend x
        (redY, yTr) = flatSend y
    in Conj redX
        (Conj redY
          (At (mkEqAtom (maybe (Prolog.Var "_") trans oty) xTr yTr)))

  trans (InstanceOf o c) =
    let (redO, oTr) = flatSend o
        cTr = trans c
    in Conj redO
        (At (mkInstanceofAtom oTr cTr))

  ...

instance Trans ObjExpr Term where
  trans (Class ce _type) =
    trans ce
  trans (Var v _type) =
    trans v
  trans (Send o m _type) =
    send (trans o) (trans m)
  trans (IntExpr il _type) =
    Integer (intLitValue il)
```

The function flatsend introduces a formula with the predicate reduce that will reduce the translated expression to a normal form:

```haskell
flatSend :: ObjExpr -> (Prolog.Formula, Term)
```
flatSend oe =
  let oeTr = trans oe
  in case oeTr of
    Structure "send" ts ->
      let nfVar = Prolog.Var("_NF_" ++ headsOfTerms ts)
      in (At (mkReduceAtom oeTr nfVar), nfVar)
    Integer _i ->
      let nfVar = Prolog.Var("_NF_" ++ headOfTerm oeTr)
      in (At (mkReduceAtom oeTr nfVar), nfVar)
    _ -> (Prolog.Top, oeTr)

Finally, function transLloydTopor implements the Lloyd-Topor transformation.

Encoding of Extended Programs in Prolog

Encoding general programs into Prolog is straightforward and we did implement it defining the following instances of Show:

instance Show GP where
  show (GP gpcs) = showAnyListWith show "" "\n\n" " gpcs

instance Show GPC where
  show (GPC h b) = show h ++
    if null b
    then "."
    else "_\:-\n"
         ++ showAnyListWith show "_\n_\n_" "_\n_\n_" "." b
  show (GPCDec d) = "_:-_" ++ d ++ "."

instance Show Literal where
  show (Pos a) = show a
  show (Neg a) = "\+_" ++ show a

instance Show Atom where
  show (Atom p ts) = p ++ if null ts
    then ""
    else showAnyListWith show "(" ",_" ")" ts
Part IV

Applications and Implementation
Chapter 7

Formal agility in Clay

The material in this chapter is part of our previous work [64]. The example we have used to illustrate some customer stories is written in Slam-sl. We think the reader can understand the example since the syntax does not differ from that presented in Chapter 2, nevertheless Section 2 in the paper [64] contains a brief description of Slam-sl.

The chapter is by the following question: “can Formal Methods (FM) interact with agile processes in general and Extreme Programming (XP) and rapid prototyping in particular?” Our thesis is that most of XP practices (pair programming, daily build, the simplest design or the metaphor) are technology independent and therefore can be integrated with FM. FM can benefit from XP practices improving its productivity or, at least, changing the perception of potential practitioners. Additionally, other practices like test first, incremental development and refactoring can be improved by using FM. In this chapter we explore the iteration of both processes in certain detail.

At first sight, XP [15] and FM [124, 69] are water and oil: an impossible mixture. Maybe the most relevant discrepancy is that while one of the strategic motivation of XP is “spending later and earning sooner” FM require “spending sooner and earning later”. However, a deeper analysis reveals that FM and XP can benefit their selves.

The use of formal specifications is perceived as improving reliability at the cost of lower productivity. XP and other agile processes focus on productivity so, in principle, using FM following XP practices could improve its efficiency. In particular, pair programming, daily build, the simplest design or the metaphor are XP practices that in our view are independent of the concrete development technology used to produce software and the declarative technology and FM is just a different devel-
On the other hand, the main criticism to XP is that it has been called *systematic hacking* and, probably, the underlying problem is the lack of a formal or even semi-formal approach. But, what XP practices are liable to incorporate a formal approach? We think that *unit testing*, *incremental development* and *refactoring* are three main XP practices where FM can be successfully applied:

- When you write a formal specification you are saying *what* your code must do, when you write a test you are doing the same so one idea is to use formal specifications as tests.

- Incremental development is quite similar to the refinement process in FM: specifications evolve to code maintaining previous functionality.

- Finally FM can help to remove redundancy, eliminate unused functionality and transform obsolete designs into new ones, and this is refactoring.

After all, it might be possible to dilute FM in XP. We would like to point out that we are not claiming to *formalise* XP (as could be understood from the joke in the title), but just to study how the declarative technology can be integrated in XP and how XP can take advantages of this technology.

In our opinion, the XP process could be adopted by using Clay (or any other FM tool) instead of an ordinary programming language and tool. In other words, we propose to write formal specifications instead of programs. A number of advantages appear:

- Rephrasing a XP rule, “The specification is the documentation” because we have a high level description with a formal specification of the intended semantics of the future code. One of the bigger efforts in the Clay development has been to ensure that the generated code is readable enough. Therefore, the “answer is still in the code” (but also in the specification).

- FM tools (theorem provers, model checkers, etc.) help to maintain the consistency of the specification and the correctness of the implementation.

- Important misunderstandings and errors can be captured in the early stages of the development but close enough to code generation.

While in Agile Methods the emphasis is on staying light, quick, and low ceremony in the process, FM could make it sometimes heavier, sometimes not. Even in the first cases we have that: i) it is still can be considered a light method in the FM area, and ii) the benefits should compensate in many cases the increase of work.
Let us focus on in three XP pieces where we consider that FM can play an interesting role: in Sections 7.1 and 7.2.1 we briefly present how formal specifications can be used in the practices of testing and refactoring. Section 7.2 focuses in the formalisation of the incremental development under the prism of FM.

7.1 Unit Testing

In XP the role of writing the tests in advance is similar to the role of writing a precise requirement: it is used to indicate what the program is expected to do. Tests in XP solves two different problems:

- The detection of misunderstandings in the intended specifications.
- The detection of errors in the implementation.

The perspective under both problems is completely different when using FM. The detection of inconsistencies in formal specifications are supported by formal tools, mainly by a generator of proof obligations and by a theorem prover assistant. With both tools the user get information about possible inconsistencies.

The detection of errors in the implementation is absolutely unneeded thanks to the verified design process: a process that ensures that the code obtained from an original specification is correct with respect to it. Notice that the use of tests do not ensure that requirements are satisfied, just “convince” the programmer that it happens. The FM approach overcome this limitation.

So we propose to replace the tests by chk formulas expressed in Clay. There are several advantages of this approach:

1. tests can be complex enough but the Clay system takes care of the code generation is feasible,
2. tests are executed automatically every time the program is run in debugging mode,
3. testing properties can be carried out in all the incremental versions of the code, i.e. they are automatically checked in all the iterations, and
4. automated formal tools can be used to improve the behaviour, for instance proving that some test are inconsistent with the specification by using a theorem proving.
7.2 Incremental Development

In this section we present the logical properties that the iterative development of software by the incremental addition of requirements must fulfil. We have called the set of those properties the Combination Property and it formally establishes that the combination of the code already obtained to solve the previous requirements and the code needed to solve the new one must fulfil all the requirements. The incremental development of XP needs to ensure that: i) at every step we develop the minimal code needed to solve the corresponding requirement, and ii) this code is combined with the previous code in such a way that the old requirements still hold. To solve this goal we establish the minimal properties that must be proved to ensure a correct behaviour.

We will call story$_i$ the formula expressing requirements at step $i$. At every step we want to develop a function $f_i$ that covers all the requirements story$_1, \ldots, $story$_i$. To obtain $f_i$ we depart from:

- the function $f_{i-1}(\bar{x}, \bar{y})$ with postcondition post$_{i-1}(self, \bar{x}, \bar{y}, result)$, and
- a function $g_i(\bar{x}, \bar{z})$ that solves requirement story$_i$.

Additionally, function $f_i$ computes “more things” than $f_{i-1}$, i.e. the result of $f_i$ includes the result of $f_{i-1}$, and maybe more data. Formally, there exists a projection $\pi_i$ that relates both results.

Let us discuss some remarks with respect to these formulas before establishing the main properties. The fact that $g_i$ is developed for requirements story$_i$ means that its postcondition entails story$_i(self, \bar{x}, \bar{z}, result)$. We assume that some of the arguments for $g_i$ are still present in the previous code, i.e. arguments represented by variables $\bar{x}$ are still present in $f_{i-1}$, while some previous arguments $\bar{y}$ are not needed for story$_i$ and some new $\bar{z}$ are required.

Now, the main property to be proved can be formulated. Let us assume that the function $f_i(\bar{x}, \bar{y}, \bar{z})$ has been specified with postcondition post$_i(self, \bar{x}, \bar{y}, \bar{z}, result_i)$. To ensure that this function is correctly defined we must prove the Combination Property:

\[
\text{post}_i(self, \bar{x}, \bar{y}, \bar{z}, \text{result}_i) \Rightarrow \\
\text{story}_i(self, \bar{x}, \bar{z}, \text{result}_i) \land \\
\text{post}_{i-1}(self, \bar{x}, \bar{y}, \text{result}_{i-1}) \land \\
\pi_i(\text{result}_i) = \text{result}_{i-1}
\]

Now we can formally establish that this is the only property (at every step $i$) needed to ensure that the final code (i.e. $f_n$) entails all the requirements.
Theorem 7.1  For every $i \in \{1, \ldots, n - 1\}$ the following formulas hold:

$$post_n(self, x, y, z, result_n) \Rightarrow story_i(self, x, z, result_i)$$

$$post_{i+1}(self, x, y, z, result_{i+1}) \wedge post_i(self, x, y, z, result_i) \Rightarrow \pi_{i+1}(result_{i+1}) = result_i$$

The proof proceeds by induction on $i$.

A Simple Example

In the following example, we will show three customer stories for the development of a small telephone database ([113]). The customer wants a telephone database where information can be added and looked up maintaining two different tables: one with the persons and other one with the entries (pairs of person and phone). The specification written by development is the following one:

```plaintext
class Phone_DB

state (members : {Person},
        phones : {((Person,Phone))})

constructor make_phone_DB
  call make_phone_DB
  post result.members = {}

modifier add_entry (Person, Phone)
  pre person in self.members and
      not (person, phone) in self.phones
  call add_entry(person, phone)
  post result.phones = self.phones + {(person,phone)}

modifier add_member (Person)
  pre not person in members
  call add_member(person)
  post members = self.members + {person}

observer find_phones (Person) : {Phone}
  pre person in dom(phones)
  call find_phones(person) = self.phones(person)
```

In the second story, the customer asks for including a way to remove entries in the data base and this is the result of the development task:
The combination property in this case is trivial to prove because we only have added a new operation. A consistency check is also trivial.

In the third customer story, she asks for removing the person from the database of members if its removed entry is the last one:

```plaintext
modifier remove_entry (Person, Phone)
pre (person,phone) in phones
call remove_entry(person, phone)
post phones = self.phones − {(person,phone)} and
    if (exists phone : Phones with (person, phone) in phones)
        then members = self.members
        else members = self.members − {person}
end
```

In this step, the postcondition of `remove_entry` must be proved to entail the previous postcondition. A theorem prover can automatically do the work: let $A$ be the formula $\text{phones} = \text{self.phones} − \{(\text{person,phone})\}$ and $B$ the right hand side of the conjunction, the proof obligation is

$$A \land B \Rightarrow A$$

what is directly the scheme of an inference rule in first order logic.

### 7.2.1 Refactoring

The declarative technology makes easier to find and remove redundancy, eliminate unused functionality and transform obsolete designs into new ones, i.e to refactor code [51]. Thanks to the reflective properties of Clay, generic patterns can be specified and it can be proved that a specification is an instantiation of such a generic pattern. The idea it is having a relevant collection of generic patterns trust the prover technology of FM were able to match specifications with specifications in those patterns. Some works in formalising design patterns [54] have been done using Clay [66].

However, we need to be sure that the resulting code from refactoring is still readable enough. In any case, taking into account that it is for free, the programmer can spend some time in documenting it.
7.3 Conclusions

We have presented how some XP practices can admit the integration of Formal Methods and declarative technology. In particular, *unit testing*, *refactoring*, and, in a more detailed way, *incremental development* have been studied from the prism of FM.

Probably there is more room for FM ideas helping agile methodologies and XP, and we will study this as a future work.

One of the goals of the Clay system is to make FM and their advantages closer to any kind of software development. Obviously FM are specially needed for critical applications but combining it with rapid prototyping and agile methodologies could make them affordable for any software construction. Up to know we have not equipped Clay with an automatic interface generator that precludes the use of our system for heavy graphical interface applications. The automatic generation of graphical interfaces is another matter of future work.
Chapter 8

Specifying in the Large

The material in this chapter is part of our previous work [63]. The concrete syntax of Slam-sl is used and introduced in this chapter. The reader will appreciate how Clay is a light-weight notation that evolved from Slam-sl.

The chapter discusses how formal methods, and, in particular, object oriented specification languages can be integrated in the software development process in an effective way. We depart from an object specification language in the SLAM system that combines characteristics of algebraic languages as well as pre and post-conditions for class methods specification. We study how to specify classes as well as the formal relations that class relationships must hold (in particular, inheritance).

One of the main features of the specification language is that it is supported by an integrated environment that, among other facilities, includes the generation of readable and (enough) efficient imperative code. We also address how this translation can be done and the extra capabilities of the environment regarding the use of formal method in the development process, for instance program validation.

One of the most important problems in software development is software integrity and reliability. In the recent history of computing several software errors that caused considerable damage can be found. For instance, the failure of the Ariane 501 was classified by the ESA official reports as a software error. Another examples is the lost of the Mars Climate Orbiters due to a software problem mixing measures in European and American metric units. The presence of computers in new activities, disciplines, and systems makes even more important software correctness. Moreover, problems like Y2K or the Euro bug show that the situation is not restricted to apparently safety critical applications. The situation is well described in the PITAC report, developed by the USA President’s Information Technology Advisory Committee as part of the Computer Science Research plan of the
USA government ([99]).

The standard solution to this problem is the use of specification languages for software description and the use of formal methods for proving properties of programs. There are advantages concerning program development, like the possibility of automatically obtaining a prototype, or the help in program evolution by reusing and modifying existing specifications by (semi) automatic manipulation. From the point of view of software validation there are some additional benefits, like the formal verification of properties, program debugging by dynamically checking formal specifications, and program maintenance, reuse and documentation.

However, there are some drawbacks, namely those concerning the reduced number of automated tools, and the lack of use of formal methods because the extra cost.

Most of the advantages are present in the Iterative Rapid Prototyping Process (IRPP): an iterative development of prototypes from some (partial) specifications that can be modified from the user information, checking the prototype after a validation step. A key role in this model is played by the specification language used for requirement description. The IRPP assumes specifying-in-the-large, i.e. the specification language must have an structure allowing for the convenient specification of large systems and tools for managing them. This encompasses two characteristics: one is a high expressiveness of the specification languages, the other one is the ability of the development environment to produce a significant amount of code from the specification.

Algebraic specification languages, like OBJ ([55], FOOPS [102], Maude [35] as well as information systems specification languages, like TROLL [72], Albert [43], Oblog [106], and OASIS [82], permit animating the specification in such a way that they are executable.

Despite the obvious benefits of the IRPP, some other authors have pointed out some additional problems ([110] like the difficulty to express non-functional requirements, and that the cost of the prototype development could represent an unacceptably large fraction of overall system cost taking into account that re-implementation is recommended. Even in the case of using an executable specification language, one of the problem remains: the prototype is still throw-away. As prototypes can be executed in an interpreted manner, once the prototype is accepted the developer needs to re-implement it in a productive language, usually an imperative one.

The situation would be improved if readable and efficient imperative code could be generated from the specification language. Some years ago this scenario was considered unfeasible, but in our opinion the current technology in programming languages and specification languages is mature enough to achieve this goal. Basically, the methodology departs from a formal specification of the product in order
to obtain systematically a program that fulfill the original specification. Most of the relevant ideas for this objective come from declarative languages development and implementation, for instance program transformation techniques. It is an obvious fact that declarative systems have a very low presence in industrial development, and having been involved in the development of declarative languages and techniques and in several development projects we are very sceptical about the real possibility to introduce declarative languages in software enterprises. From our point of view the only feasible way to incorporate those techniques into software production is to adapt them to imperative systems (see for instance the success of ILOG, a constraint logic programming library for C/C++).

Our proposal to fully implement the IRPP and to support specifying-in-the-large is the SLAM system. The system contains several components based on formal methods: an object oriented specification language, an advanced development environment including efficient and readable code generation and a library to allow a high level specification.

The most novel feature of the SLAM specification language (Slam-sl) is that is designed as a trade-off between the high expressiveness of the underlying logic and the possibility of an efficient compilation. Slam-sl formula, an extension of logic formula, are used to specify functions by means of a precondition (condition to apply the function with success), a postcondition (that relates input arguments and the result), and a solution (an effective method to compute the function). One key concept is the operational use of ‘quantifiers’ (extending usual logic quantifiers). Quantifiers allow the expressiveness of logic while the basis for their efficient implementation are the characterization of the classes (in the sense of object orientation) that can be traversed and the method to do it. By using program transformation techniques that will be discussed later, it is possible to obtain code in a high level programming language, an object oriented one preferably, like Java or C++, although SLAM is language independent, that code is efficient and \textit{readable}, so it can be modified by the user. By readable, we understand a code where original modelling is translated into equivalent imperative types and function descriptions are moved to easy to follow code (in particular, using loops) with adequate declarative annotations.

With respect to related work, the closest proposals have been mentioned above: executable algebraic specification languages (the OBJ family) and animable information system description languages. The goal to automatize the IRPP is present both in those proposal and SLAM. There are some other experiences adding declarative features to imperative languages (the most recent and interesting is Pizza \cite{96}) but the level of abstraction is not higher enough to be considered as specification languages. In any, to our knowledge SLAM is the first complete approach to obtain full code from specifications without adding limitations to the original language.
The rest of the paper is organized as follows. Section 8.1 presents the main characteristics of the specification language Slam-sl. Next two sections are devoted to code generation, showing how to compile algebraic types into imperative ones in section 8.2 and how to transform Slam-sl specifications into efficient imperative code in section 8.3. Finally, we conclude and sketch some hints for future work.

8.1 The SLAM Specification Language

This section presents the main constructions of the language Slam-sl. Slam-sl is part of the SLAM project, a software construction development environment that is able to synthesize reasonably efficient and readable code in different high level object oriented target languages like C++ or Java. Among other features, the user can write specifications in a friendly way, track her hand-coded optimizations, and check, in debug mode, those optimizations through automatically synthesized assertions.

In order to facilitate the understanding of Slam-sl we will show its elements with a concrete syntax that does not necessarily correspond neither with an internal representation nor the environment presentation, so the reader should not pay attention to the concrete syntax but to the abstract one.

A Slam-sl program is a collection of specifications that defines classes and class properties. The specification of method behaviour is given through preconditions and postconditions but with a functional flavour as we will see.

8.1.1 Classes and Class Relationships

In Slam-sl, a class is defined by specifying its properties: name, relationships with other classes, and methods.

As in many object oriented programming languages, different kind of relationships between classes cannot be distinguished as UML allows. For instance, aggregation cannot be distinguished from composition, and some associations are implicit through the semantics of methods. Anyway, the following relationships that can be caught statically are listed:

**Aggregation:** the state specification of a class defines an aggregation or composition among class instances.

---

1 In fact, Slam-sl programs are stored in XML format and its presentation in the environment can be customized.
8.1 The SLAM Specification Language

Inheritance: class properties can be defined from scratch or by inheriting them from already defined classes. Overriding of such properties are constrained in Slam-sl, not only the signatures but also the meaning (see subsection 8.1.2).

Polymorphism: generic polymorphism is introduced by permitting introducing arguments in types. Slam-sl allows deferring classes in the style of Eiffel but adding some features from theories (in OBJ terminology [55]) as well as type classes (à la Haskell [70]) playing a more powerful role than C++ templates.

Let us see a simple example:

```plaintext
class Stack inherits Collection
state empty
state non_empty (top : Object, rest : Stack)
```

The first line declares a new class called `Stack` and it establishes that class `Stack` inherits properties from `Collection`. Lines starting with `state` define attributes that are the internal representation of the class instances. Slam-sl permits defining algebraic types to indicate that a syntactical construction represents class instances. Syntactically algebraic types allow for alternative definitions of constructor rooted tuples. Semantically, different descriptions of type elements are combined in a declarative fashion. In our example, the constructors `empty` and `non_empty` cover the two descriptions of the state of a stack. The concrete values `empty` and `non_empty` (5, empty) represent an empty stack and the state of a stack with a unique object (the constant 5) respectively.

Each state can be associated with an invariant over the attributes that the state defines. Important properties of the class instances can be captured in the invariants.

```plaintext
class Point
state polar (r : Float, a : Float)
invariant r ⩾ 0 and 0 ≤< a and a < 2 * pi
```

In section 8.2 algebraic types in Slam-sl are shown in detail.

8.1.2 Method specifications

Slam-sl has a clear functional flavour, so methods are represented by functions but the user can classify different kinds of methods: constructors, modifiers and observers.

- Object constructors. An object constructor is a function designed to create new instances of a class.
• **Object observers.** Observers allow to access properties of an object without *modifying* it. Slam-sl provides free observers for record fields (with the name of the field).

• **Object modifiers.** Modifiers are designed to *modify* the value of an object.

In spite of the classification above, all the methods are functions that involve several objects.

The standard methods over stack objects permit creating an empty stack, decide if a stack is empty, consulting the top of the stack, and push and pop elements. Let us complete the stack class specification with the definition of its methods:

```plaintext
constructor make_empty
pre  true
call make_empty
post result = empty

observer is_empty : Bool
pre  true
call is_empty
post result = (self = empty)

observer top : Object
pre not self.is_empty
call top
post result = self.top
modifier push (Object)
pre  true
call push(x)
p post result = non_empty(x, rest)

modifier pop
pre not self.is_empty
call pop
post result = self.rest
```

A method is specified by a set of *rules*, every rule involves a *guard* or a *precondition* that indicates if the rule can be triggered, an *operation call scheme*, and a *postcondition* that relates input state and output state. The general form of a rule is the following:

```plaintext
function op(T) : R
pre := P(x,self)
```
8.1 The SLAM Specification Language

\[
\text{op}(x) \\
\text{post} : \quad Q(x, \text{self}, \text{result})
\]

where \( P(x, \text{self}) \) is a Slam-sl formula (see section 8.1.4) involving variables in the argument \( x \) and the recipient of the message \( \text{self} \) in case of the operation to be either an observer or a modifier. \( Q(x, \text{self}, \text{result}) \) is another formula involving variables in the argument, the reserved symbol \( \text{result} \) that represents the computed value of the function and \( \text{self} \) that represents the state of the receipt of the message before the method invocation.

Some shorthands help the user to write formulas concisely and readably: \( \text{self} \) can be omitted for accessing attributes, explicit function definitions, as in VDM, are allowed, and unconditionally true preconditions can be skipped.

Method overriding

Let us explain in some details how Slam-sl handles method overriding. Suppose you have a class \( C \) with a method \( m \) with precondition \( P \) and postcondition \( Q \). Now, a subclass \( C' \) of \( C \) is declared supplying a new specification for \( m \): precondition \( P' \) and postcondition \( Q' \). As Slam-sl is a formal specification language, it is forces that the following statements hold:

\[ P \rightarrow P' \]
\[ (P \land Q') \rightarrow Q \]

Interfaces

In Slam-sl it is quite easy to declaratively specify interfaces, i.e. class with no state and methods that must be redefined in the subclasses. The way to declare such method is to indicate that the precondition is false. This means that this method is not applicable in any case. Notice that it is still possible to supply an adequate postcondition. This postcondition must be preserved in all derived classes. Those methods that have no definition are implicitly considered to have the precondition false and the postcondition true. For the practical use of Slam-sl as an specification language a dedicated syntax for interfaces can be introduced. We just want to stress the point that they are just translated into this simplified form.
Encapsulation

Encapsulation is an important concept in programming languages which permits the user to control coupling and maximize cohesion. In general, encapsulation is not encouraged in formal methods. In Slam-sl the user can indicate the visibility scope of each property: *public*, *protected* or *private*. If an attribute is indicated as public the user gets for free an observer, for instance, in the stack example the definition of the observer *top* could have been avoided in this way:

```
state non_empty (public top : Object, rest : Stack)
```

If a complete state is declared as public then a free constructor is obtained and the definition of *make_empty* could have been avoided with:

```
public state empty
```

‘Inheritance by Composition’

Slam-sl also introduces a broad notion of inheritance by composition. Let us see an example, the following Slam-sl specification defines a *read only wrapper* for stacks:

```
class ROStack

public state wrap (target : Stack accept
  public top : Object
  public is_empty : Bool)
```

Now, *wrap* creates instances of *ROStack* which state is a stack and which methods are *top* and *is_empty* that resends those messages to the state object. This is a pretty unexplored feature that we have called ‘inheritance by composition’.

8.1.3 Slam-sl Predefined Classes

As many other specification languages, Slam-sl has a powerful toolkit with predefined types representing booleans, numbers, characters and strings, records and tuples, collections (sequences, sets, etc.), dictionaries (maps, relations, etc.).

Slam-sl type syntax reflects value syntax, for instance, the type ‘sequence of integers’ is written as `[Integer]` and its values are written as `[1,2,3]`, a tuple type can be written as `(Char,Integer)` and its values as (`'a',32`), a set type like `{String}` groups together values like `{"Hello","world"}` or `{}`.
The most interesting feature of sets and sequences is that they inherit from the predefined abstract class `Collection`. All the classes that inherit from it must define a traversal\(^2\). This will be shown in detail in section \[8.3\]

8.1.4 Slam-sl Formulas and Quantifiers

Slam-sl formulas are basically logic formulas built using the usual logical connectives (`and` - conjunction, `or` - disjunction, `not` - negation, `implies` - implication, and `equiv` - equivalence), predefined and user defined functions and predicates, and quantified expressions. Slam-sl formulas are typed in a similar way than other expressions. In fact, expressions and formulas share the syntax – every formula is an expression of type boolean. Slam-sl expressions can combine objects with its own operations. Operations can be combined in any consistent way to produce new expressions.

Expressions can use quantifiers over elements of adequate types. Quantifiers are a key feature in Slam-sl and extend the notion of quantifier in logic. They can compute not only the truth of an assertion but also any other value. In Slam-sl, a quantified expression is written in the following way:

\[
q \ x \ in \ d \ [\text{where } F(x)] \ with \ E(x)
\]

The above expression scheme is a quantified expression:

- `q` is the quantifier symbol that indicates the meaning of the quantification by a binary operation and a starting value,

- `d` is an object of a special predefined class `Collection`,

- `x` is the variable the quantifier ranges over,

- `F` is an optional boolean expression that filters elements in the collection, and

- `E` represents the function applied to elements in the collection previous to computation.

Some predefined quantifiers are shown in the following table with an informal description:

\(^2\)A specific method to traverse the collection from which iterative or recursive code can be automatically generated.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Generalizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>exists</td>
<td>$\lor$ with false.</td>
</tr>
<tr>
<td>exists1</td>
<td>as exists but limiting the count to 1</td>
</tr>
<tr>
<td>forall</td>
<td>$\land$ with true</td>
</tr>
<tr>
<td>sum</td>
<td>$+$ with 0</td>
</tr>
<tr>
<td>prod</td>
<td>$\times$ with 1</td>
</tr>
<tr>
<td>count</td>
<td>inc with 0 (counting)</td>
</tr>
<tr>
<td>select</td>
<td>searching</td>
</tr>
<tr>
<td>max</td>
<td>max</td>
</tr>
<tr>
<td>maxim</td>
<td>maximizers</td>
</tr>
<tr>
<td>map</td>
<td>apply a function to every element</td>
</tr>
</tbody>
</table>

Let us show some examples and their intended meaning:

<table>
<thead>
<tr>
<th>Example and meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall $x$ in ${1,2,4,7,8}$ with $x &lt; 11$ = true</td>
</tr>
<tr>
<td>count $x$ in ${1,2,4,7,8}$ with $x$.isPrime = 3</td>
</tr>
<tr>
<td>sum $x$ in $[1..10]$ where $x &lt; 5$ with $x.pow(2)$ = 30</td>
</tr>
<tr>
<td>map $x$ in ${1..10}$ with $x / 2$ = ${0,1,2,3,4,5}$</td>
</tr>
<tr>
<td>map $x$ in $[1..10]$ with $x / 2$ = $[0,1,1,2,2,3,3,4,4,5]$</td>
</tr>
</tbody>
</table>

**Solutions**

In Slam-sl constructive formulas, those that code can be generated from, are called solutions. They can be syntactically characterized by assigning a value to the result variable and restricting quantifiers to finite collections. The user can supply a solution when the postcondition does not offer a method to compute the function.

**8.2 Algebraic Types and Pattern Matching**

Two important features peculiar to functional programming languages are algebraic types and function definition using pattern matching. ‘Variant records’ and ‘union types’ are the imperative version, but algebraic types and pattern matching favour conciseness and readability.
We believe that algebraic types increase the language's expressive power by extending standard type systems with a strong theoretical foundation. Because of this, algebraic types have been introduced in Slam-sl, considering that they are a natural and abstract way to represent values. Moreover, the algebraic types allow the user to define its own types as allow Slam-sl developers to specify the whole predefined types. This is an important advantage because the formal engines for refinement, transformation, verification and synthesizing will be based in a smaller set of deduction rules.

The following example tries to illustrate the former discussion. The predefined Slam-sl type for lists can be specified in the language itself by using no other predefined domain but algebraic types:

```plaintext
class List

public state empty
    state non_empty (public head : Object, public tail : List )

modifier add_to_front (Object)
call add_to_front(y) = non_empty(y, self)

observer length : Nat
call (empty).length = 0
call (non_empty(x,xs)).length = 1 + xs.length

observer includes (Object) : Bool
call (empty).includes(y) = false
call (non_empty(x,xs)).includes(y) =
    (x = y) or xs.includes(y)

modifier remove (Object)
call (empty).remove(y) = empty
call (non_empty (x,xs)).remove(y) =
    if x = y
    then xs
    else non_empty(x, xs.remove(y))
end

modifier append (List)
call (empty).append(ys) = ys
call (non_empty(x,xs)).append(ys) =
    non_empty(x, xs.append(ys))
```

For those readers acquainted with functional programming the specification above is ‘standard’. However, no experience with functional languages is required because the previous specification is easy to understand: every operation is defined by several rules and every rule specifies the behaviour of the operation depending on the state of the object (empty or non_empty).

In order to notice the expressiveness of algebraic types versus ‘standard’ modelling in procedural language we can discuss how to implement the above specification in one of those languages. The domain description can be basically kept by using union types or variant records. However, recursive domains need for the inclusion of pointers. Of course, the different level of abstraction can be justified by the different goal: specification versus implementation. Our proposal try to make both steps closer by generating the implementation in mind automatically.

### 8.2.1 Compiling Algebraic Types

In [96], a method for compiling algebraic types and pattern matching is given based on the association of object types with algebraic types. Nevertheless, the ideas in Pizza should be revisited because in strongly typed languages (like Java), objects cannot dynamically change their class and functional interfaces are forced. Some different compilation schemes that avoid that problem are proposed.

**Compilation scheme 1.**

Our first proposal being presented is likely the most efficient one. Every state is distinguished by a tag (the discriminate attribute) and actual attributes in every state are included as attributes in the target class. Attributes can be meaningless depending on the tag. The compilation scheme of algebraic types can be found in figure 8.1.

The function

\[ trans_{decl} : [(String, Type_Name)] \]

formalizes the translation scheme. \( trans_{decl}(\bar{x}: \bar{T}) \) represents the translation of Slam-sl attributes into Java. Following the scheme, the specification of the list class can be automatically translated into Java in the following way:

```java
public class List {
    private int state; // represents the object state

    private static final int EMPTY = 1;
    /* no attributes when state == EMPTY */
```
8.2 Algebraic Types and Pattern Matching

Slam-sl target code

class A
state K_1
...
state K_n
state C_1 (x_1 : T_1)
...
state C_m (x_m : T_m)

Java object code

class A {
    private int state; // represents the object state

    private final static int K_1 = 1;
    /* no attributes when state == K_1 */
    ...
    private final static int K_n = n;
    /* no attributes when state == K_n */

    private final static int C_1 = n+1;
    /* attributes when state == C_1 */
    trans_decl(x_1 : T_1);
    ...
    private final static int C_m = n+m;
    /* attributes when state == C_m */
    trans_decl(x_m : T_m);
}

Figure 8.1: Compilation scheme 1 of algebraic types
private static final int NON_EMPTY = 2;
/* attributes when state == NON_EMPTY */
private Object head;
private List tail;

private List () { }
/* A factory method for empty stacks */
public static List empty () {
    List result;
    result = new List();
    result.state = EMPTY;
    result.head = null;
    result.tail = null;
    return result;
}

private static List nonEmpty (Object head, List tail) {
    List result;
    result = new List ();
    result.state = NON_EMPTY;
    result.head = head;
    result.tail = (List)tail.clone();
    return result;
}

Compilation scheme 2.

The main idea in this proposal is to introduce a new class AState for representing the state of the target class A and as many subclasses as states have been declared in A. Our compilation can be understood as the application of the state design pattern ([54]).

For the list class example, the state of a list is represented by an instance of ListState that can exclusively be an instance of ListStateEmpty or an instance of ListStateNonEmpty. Attributes in ListState subclasses come from the state declarations in Slam-sl. The translation follows:

public class List {
    ListState state;

    private List () { }
}
private List (ListState state) {
    this.state = state;
}

/* A factory method for empty stacks */
public static List empty () {
    return new List(ListState.emptyState());
}

abstract class ListState {
    public static ListStateEmpty emptyState () {
        return new ListStateEmpty();
    }
    public static ListStateNonEmpty nonEmptyState (
            Object head,
            ListState tail) {
        return new ListStateNonEmpty(head, tail);
    }
}

class ListStateEmpty extends ListState {
    public ListStateEmpty () { }
}

class ListStateNonEmpty extends ListState {
    public final Object head;
    public final ListState tail;

    private ListStateNonEmpty () {
        this.head = null; this.tail = null;
    }

    public ListStateNonEmpty (Object head,
            ListState tail) {
        this.head = head; this.tail = tail;
    }
}

Optimization.

The latter compilation scheme allows introducing an optimization: when the algebraic type contains a constant case, then that case can be represented by null and one of the classes representing states can be dropped. In the example, the class
ListStateEmpty can be avoided by representing the empty case with \textit{null} in the state attribute.

### 8.2.2 Compiling Pattern Matching

In order to decide if a rule must be applied, every rule is compiled into Java code checking its invocation pattern. Depending on the translation scheme the pattern matching compilation will be different.

For the first one, pattern matching is compiled into \textit{case} expressions discriminating with respect to the value of the tag. The translation of the length and append methods would be the following ones:

```java
public int length () {
    int result = 0;
    switch (this.state) {
    case EMPTY:
        result = 0;
        break;
    case NON_EMPTY:
        result = 1 + this.tail.length();
        break;
    }
    return result;
}
```

```java
public void append (List ys) {
    switch (this.state) {
    case EMPTY:
        this.state = ys.state;
        this.head = ys.head;
        this.tail = (List)ys.tail.clone();
        break;
    case NON_EMPTY:
        this.tail.append(ys);
        break;
    }
}
```

For the second compilation scheme, pattern matching is compiled taking advantage of dynamic binding in the object-oriented paradigms and the \textit{state} design pattern characteristics. Every rule is actually implemented in the corresponding state subclass. Let us see the translation of both methods length and append:

```java
public class List {
    public int length () {
```
In the target class the same method is invoked over the state attribute. Every rule is translated into a method in its corresponding (state) class:

```java
abstract class ListState {
    public abstract int length();
    public abstract ListState append(List ys);
}

class ListStateEmpty extends ListState {
    public int length() {
        return 0;
    }
    public ListState append(List ys) {
        return ys.state;
    }
}

class ListStateNonEmpty extends ListState {
    public int length() {
        return 1 + tail.length();
    }
    public ListState append(List ys) {
        return ListState.nonEmptyState
               (this.head, this.tail.append(ys));
    }
}
```

### 8.3 Compiling Slam-sl Solutions into Efficient Code

A solution in Slam-sl is a constructive formula with the same meaning of a postcondition but with the property that an effective method for computing the function can be extracted from. The current characterization of solutions is syntactical.

*Set comprehension* is a powerful construct in many formal methods ([69][113][3]). From collection construction to filtering, the concept of comprehension provides expressiveness and conciseness. Some programming languages have gener-
alized the concept to its main data structures, as Haskell does with lists or Smalltalk does with collections. The notion of comprehension needs not to be restricted to sets, and comprehension over any kind of 'collection' could be allowed. Several problems can be solved by specifying a way to traverse any collection while some result is computed. Besides expressiveness, another important advantage is that synthesizing efficient code is possible.

In Slam-sl, traversals over collections is a generalization of 'collection comprehension'. Patterns for traversing collections have been introduced in the language. If the user wants to define a new collection, his class must be declared as a subclass of `Collection` and the way in which values of the collection are traversed must be specified. Currently, our decision is that the user must specify an 'abstraction' function from collections to sequences. Let us see an example of the traversal definition in a class for representing trees:

```plaintext
class Tree (T) inherits Collection (T)

state nil
state node (left : Tree (T),
    root : T,
    right : Tree (T))

/* Definition of the preorder traversal */
public observer traversal : [T]
    (empty).traversal = []
    (node(ls,r ,rs)).traversal = [r] + ls.traversal + rs.traversal

By using program transformation techniques it is possible to obtain code in a high level programming language that is efficient and readable, and that, consequently, it can be modified by the user. Informally, the following quantified expression

\[ q \, x \, \text{in} \, d \, \text{where} \, F(x) \, \text{with} \, E(x) \]

represents an 'iteration' over the elements of \( d \). \( \text{traversal} \) computes an accumulated result with the meaning of \( q \) and values of \( E(x) \), except those that do not fulfil \( F(x) \).

The main idea is to combine the recursive definition of the traversal with the recursive definition of the quantifier. The folding-simplify-unfolding technique from program transformation in functional programming is used in order to obtain an efficient recursive code without the need of an intermediate sequence. This code is easily translated into imperative recursive code in case of multiple recursive traversals. However, when the traversal is defined by linear recursion, the usual methods
to translate recursive definitions to sequential code is used in order to obtain efficient imperative code.

Let us see a couple of examples. The first one relies on trees. The tree traversal was previously defined:

```c
/* Definition of the preorder traversal */
public function traversal (Tree (T)) : [T]
call traversal (empty) = []
call traversal (node(ls, r, rs)) = [r] + ls.traversal + rs.traversal
```

On the other hand we have the recursive definition of, let say, the universal quantifier over sequences:

```c
forall x in [] where F(x) with E(x) = true
forall x in [x_1] + xs where F(x) with E(x) = (F(x) ∧ E(x_1)) and
  forall x in xs where F(x) with E(x)
```

so the following equational reasoning can be made:

```c
forall x in tree where F(x) with E(x) =
forall x in traversal(tree) where F(x) with E(x) =
  true
  if s = []
    forall x in s.suffix (2) where F(x) with E(x)
      if not F(s(1)) and s /= []
        E(s(1)) and
      forall x in s.suffix (2) where F(x) with E(x)
        if F(s(1)) and s /= []
      where s = traversal(tree)
= (Unfolding of traversal(tree))
  true
  if tree = empty
    forall x in traversal(ls) + traversal(lr)
      where F(x) with E(x)
      if not F(r) and tree = node(ls, r, rs)
        E(r) and
    forall x in traversal(ls) + traversal(lr)
      where F(x) with E(x)
      if F(r) and tree = node(ls, r, rs)
= (Distribution of quantifiers over sequences)
true
  if  \( \text{tree} = \text{empty} \)
for all  \( x \) in traversal(\( ls \)) where \( F(x) \) with \( E(x) \) and
for all  \( x \) in traversal(\( lr \)) where \( F(x) \) with \( E(x) \)
  if  not \( F(r) \) and  \( \text{tree} = \text{node}(ls, r, rs) \)
\( E(r) \) and
for all  \( x \) in traversal(\( ls \)) where \( F(x) \) with \( E(x) \) and
for all  \( x \) in traversal(\( rs \)) where \( F(x) \) with \( E(x) \)
  if  \( F(r) \) and  \( \text{tree} = \text{node}(ls, r, rs) \)
= (Folding of quantifier definition)
true
  if  \( \text{tree} = \text{empty} \)
for all  \( x \) in \( ls \) where \( F(x) \) with \( E(x) \) and
for all  \( x \) in \( rs \) where \( F(x) \) with \( E(x) \)
  if  not \( F(r) \) and  \( \text{tree} = \text{node}(ls, r, rs) \)
\( E(r) \) and for all  \( x \) in \( ls \) where \( F(x) \) with \( E(x) \) and
for all  \( x \) in \( rs \) where \( F(x) \) with \( E(x) \)
  if  \( F(r) \) and  \( \text{tree} = \text{node}(ls, r, rs) \)

Let us now show an example of linear recursion. Suppose the following traversal over list instances are defined:

function traversals (List (T)) : [T]
call  traversals (non_empty(x, r)) = [r] + traversals (r)

Now the select quantifier is used as the running example:

select  \( x \) in  \[ \] where  \( F(x) \) with  \( E(x) \)  \=  \text{UNDEFINED}
select  \( x \) in \( [x_1] + xs \) where  \( F(x) \) with  \( E(x) \)  \=  \( x_1 \)
  if  \( F(x_1) \) and  \( E(x_1) \)
select  \( x \) in  \( xs \) where  \( F(x) \) with  \( E(x) \)
  otherwise

and the following equational reasoning can be made:

select  \( x \) in  \( l \) where  \( F(x) \) with  \( E(x) \)  \= 
select  \( x \) in traversal(\( l \)) where  \( F(x) \) with  \( E(x) \)  \=  \( s(1) \)
  if  \( F(s(1)) \) and  \( E(s(1)) \)
select  \( x \) in  \( s.suffix(2) \) where  \( F(x) \) with  \( E(x) \)
  otherwise
\( \text{where} \  s = \text{traversals}(l) \)
8.4 Conclusion

We have presented how some formal methods, and declarative techniques can be fully integrated in the software development process. The object specification language Slam-sl is expressive enough to describe software applications. We have collected a small but significant set of examples (bank transactions, population simulation, seat assignment in elections, etc. that can be found in our papers). It can also be used in program debugging. The expressiveness of Slam-sl and algebraic specification languages is difficult to compare. While Slam-sl includes logical formulas, the level of abstraction of the description of certain operations is higher in the language. However, Slam-sl needs to provide a model of the class (attributes) in order to generate code, but this code is directly an imperative one. If we restrict to the specification and the debugging facilities, every OBJ specification can be easily translated into a Slam-sl specification. As a future work we plan to generate the proofs needed to ensure the correctness of solutions w.r.t postconditions, providing verified generated code.

Moreover, the SLAM system is able to generate efficient and readable code. We have described how to obtain imperative data types from the Slam-sl class at-
tributes even in the case of recursive definitions. Furthermore, some techniques from program transformation used for program optimization can be adapted to Slam-sl for the generation of imperative code.

As SLAM can cope with these two issues (expressiveness and efficient code generation) it can be be used for specifying-in-the-large and can promote the use of formal methods in industrial developments. The case of the B language and development method [3] and the experience on the formal specification and development of the line 14 of the Paris underground [16] (among other examples of this kind) have shown that it is not only feasible but productive to use formal methods in software development. The novel approach of SLAM is mainly focused on the ability to generate readable and efficient code in such a way that the use of IRPP and formal methods is closer, cheaper, and more effective.

Acknowledgments

We are grateful to the anonymous referees for their comments to improve this work. We are also grateful to Alvaro Drukaroff for his work in the implementation of the Slam-sl compiler.

The SLAM project is supported in part by the Spanish Ministry of Science and Technology by grant TIC2000-1632.
Chapter 9

Modelling Design Patterns in Clay

The material in this chapter is part of our previous work [1, 61, 125, 115]. The needed concrete syntax of Slam-sl is used and introduced in this chapter. The reader will appreciate how Clay is a light-weight notation that evolved from Slam-sl.

The objective of this chapter is twofold. First to present a formalization of design patterns. Second to give an important application of Clay. Most material is an adaptation to Clay of published work: [1, 61, 125, 115], being the last particularly relevant.

Although design patterns are informal, its formalization needed if tools that mechanised them are wanted. We present a formalization of (some) design patterns as operators between classes. Being Clay object oriented and reflective this is done with the “standard” API of the language (clay.lang.reflective.Class, etc.). Potentially, we can check a design follows a patterns, detect patterns in code, and guide a refactoring process.

9.1 Introduction

Design patterns [54] and refactoring [51] are two sides of the same coin: the aim of the application of both concepts is creating software with an underlying high quality architecture. Design patterns are “descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context” ([54]), theoretically the domain of application of design patterns is the set of
problems. Refactoring is “the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure” ([51]), its application domain is the set of solutions. In practice design patterns are not directly applied to problems but to *vain* solutions, sometimes to conceptual solutions sometimes to concrete solutions: in many cases design patterns are *model refactoring descriptions*.

The thesis of the works of Tokuda [117] and Cinnéide [32] is that automating the application of design patterns to an existing program in a behaviour preserving way is feasible, in other words, refactoring processes can be automatically guided by design patterns. In this work we show our pattern design formalization and its relation with design and refactoring automation (as well as other useful applications).

The first unavoidable step is to introduce a formal reading of design patterns. This will be done in terms of class operators. Then some practical applications will be presented: how to use the formalism to reason about design patterns and how to incorporate this model into design tools and software development environments.

Let us provide an informal and intuitive description of our proposal. A given (preliminary) design is the input of a design pattern. This design is modelled as a collection of classes. The result of the operation is another design obtained by modifying the input classes and/or by creating new ones, taking into account the description of the design pattern.

For instance, let consider you have an interface *Target* that could be implemented by an existing class *Adaptee* but its interface does not match the target one. The design pattern *Adapter*, considered as an operator, accepts classes *Target* and *Adaptee* as input, and returns a new class *Adapter* that allows for connecting common functionality. Similarly, when a client needs different variants of an algorithm, it is possible to put each variant of the method in different classes and abstract them automatically “inventing” the abstract class that configures the *Strategy* pattern.

In order to define design patterns as operations between collections of classes, we use specification languages as the working framework, like Z [113], VDM [69], OBJ [55], Maude [35], or Larch [78]: a function that models the design pattern is specified in terms of pre and postconditions. A precondition collects the logical conditions required to apply the function with success. In our case, it allows specifying some aspects of the design pattern description in a non ambiguous way. Talking in terms of the sections used to describe a pattern, the pattern function precondition establishes the *applicability* of the pattern. For instance, in the pattern *Strategy* mentioned above, the precondition needs to ensure that all the *input* classes define a method to abstract with the same signature. A postcondition relates input arguments and the result. In the *Adapter* operator, the postcondition establishes that input classes (*Target* and *Adaptee*) are not modified, and that a new
class (Adapter) is introduced, inheriting from the input classes. The Adapter methods are described by adequate calls to the corresponding Adaptee methods. The postcondition encompasses most of the elements of the intent and consequences sections of the pattern description.

These specification languages have a well established semantics and provide elements to describe formally software components. Among the wide variety of mentioned specification languages we will use our own language SLAM-SL \textsuperscript{107} along the paper for practical reasons, basically, because we have a complete model of object oriented aspects in the language itself by the use of reflection, and a firm knowledge of the tools around it. It will be clear to the reader that any other language can be used instead. One key point of SLAM-SL is the reflection capabilities \textsuperscript{66}, i.e. the ability of the language for representing and reasoning on aspects of itself (classes, methods, pre and postconditions, etc.) As the work is not devoted to SLAM-SL we restrict the presentation of SLAM-SL to an appendix, that could be omitted in the final version. Anywhere, the examples are relatively easy to understand.

Formalization is one of the main advantages of our approach because it allows for formal reasoning about design patterns. The purpose of formalization is to resolve questions of relationships between patterns (when a pattern is a particular case of another), validation (when a piece of program implements a pattern), and, specially, it is a mandatory basis for tool support. Additionally, the view of design patterns as class operators allows for a straightforward incorporation into object oriented design and development environments: it can be used to modify an existing set of classes to adapt them to fulfil a design pattern, which is an example of refactoring.

The chapter organization is as follows: Section \ref{sec:9.2} provides an adequate background for the work, namely a brief presentation about our specification language SLAM-SL, and specially its reflective capabilities. Section \ref{sec:9.3} is devoted to the main subject of the chapter: how design patterns can be described as class operations. Additionally, we describe two possible applications: how to reason about design patterns, and how a design can be automatically refactored using design patterns. Future and emerging trends are included in Section \ref{sec:9.4}. Finally we provide some concluding remarks (Section \ref{sec:9.5}. Section \ref{sec:9.6} presents more examples of formalization of patterns.

\section{9.2 Background}

This section is focused on two aspects. The first one is to introduce our specific way of modelling object oriented specifications. As we have mentioned, we use object
oriented specification languages and, in particular, our own proposal SLAM-SL. We focus on the reflexive features of the language as they are crucial for the specification of design patterns. The second goal of the section is to discuss some related work that represents alternative ways for the formal approach to design patterns.

9.2.1 Modelling object oriented specifications

Object oriented concepts must be modelled in order to formalise design patterns and refactoring. There are two options: to model all these characteristics in some specification language (as an additional theory, or a library), or to use a reflective object oriented specification language. For this chapter we have decided to use SLAM-SL, an object oriented formal specification language that fulfils the second option (see [66]).

Data modelling

The two fundamental means of modelling in the object-oriented approach are composition and inheritance. The abstract syntax to specify composition and inheritance in SLAM-SL is virtually identical to those found in widespread object-oriented programming languages like Java and design notations like UML so an ordinary developer should feel comfortable with the notation.

Class Declaration

A class is declared with the following construction:

```
class C
```

where the symbol C represents a valid class name. Once a class is declared, the user can add properties for that class. We use the term property to name every characteristic of instances of the class, including methods. Composition and inheritance relationships allow the user to model data.

Generics

SLAM-SL also supports generic classes by using bounded parametric polymorphism ([26]). Syntax for generics are similar to the syntax in Java 1.5 and templates in C++. The declaration of a generic class in SLAM-SL follows this syntax:

```
class B<X> inherits A
```
9.2 Background

$X$ is a class variable, it can be used in the specification of $B$ and must be instantiated when $B$ is used. The syntax for instantiation is $B<D>$ where $D$ is a subclass of $A$.

**Composition**

In class based object oriented languages, a class is a template that defines the structure of the information of class instances. Usually, the data are modelled through a set of fields containing the information. The construction

\[ \text{state } s \left( l_1 : C_1, \ldots, l_n : C_n \right) \]

in a class specification establishes that instances have $n$ fields ($l_i$) being each one an instance of a class ($C_i$).

The syntax designed to access to the properties of instances is the infix \textit{send} operator $\leftarrow$. Fields are instance properties so in the example above, if $o$ is an instance of the class $C$ then $o \leftarrow l_i$ is an instance of the class $C_i$. Formally $l_i$ is a function from $C$ in $C_i$.

SLAM-SL has an important \textit{toolkit} with predefined classes like \textit{Boolean}, \textit{Integer}, \textit{Float}, \textit{String}, \textit{Seq} (generic sequences), \textit{Tup_2} (generic tuples of two components), etc.

Let us show a concrete simple class specification of a telephone database (adapted from an example in [124]):

\begin{verbatim}
class PhoneDB
state phoneDB (members : Seq<Person>, phones : Seq<Tup_2<Person,Phone>>)
\end{verbatim}

Every instance $db$ of \textit{PhoneDB} has two properties: \textit{members} is a finite sequence of instances of \textit{Person}, and \textit{phones} that is a finite sequence of pairs of instances of \textit{Person} and \textit{Phone}. The syntax to \textit{access} to the information of fields \textit{members} and \textit{phones} is $db \leftarrow \text{members}$ and $db \leftarrow \text{phones}$.

The name given after the keyword \textit{state} is a constructor of instances of the class. This allows us to write an expression that represents an instance of \textit{PhoneDB} (assuming that the names \textit{mary} and \textit{jones} are persons and 7254 is a telephone number, and that the syntax $[x_1, \ldots, x_n]$ represents sequences):

\begin{verbatim}
phone_DB([mary, jones], [(mary,7254)])
\end{verbatim}

An specifically design syntax for sequences and tuples has been introduced in SLAM-SL: $[X]$ is a class for sequences of instances of $X$ (Seq$<X>$) and $(X,Y)$ is a class for tuples of instances of $X$ and $Y$ as components (Tup$\_2<X,Y>$).

\footnote{The reader can ignore the metasymbol $s$.}
\footnote{The \textit{standard} notation in object oriented languages is the \textit{dot} ($\cdot$).}
Invariants

In order to constraint the domain, the use of invariants are allowed. In the case of the database, a reasonable constraint could be that every person in an entry of the collection of phones were in the collection of members:

\[
\text{invariant } \forall e:(\text{Person}, \text{Phone}) \ (e \in \text{phones} \Rightarrow e\leftarrow\text{fst} \in \text{members})
\]

where \(\text{fst}\) is a method that returns the first component of a tuple and \(\in\) is a method that decides if an instance belongs to a sequence of instances.

Invariants establish which expressions are really valid instances. The expression

\[
\text{phone_DB(}\{\text{mary, jones}\}, \{(\text{jim,7254})\})
\]

would not be valid because \(\text{jim}\) is not in the collection \(\{\text{mary,jones}\}\). These invariants can be understood as preconditions of the state constructor.

A second example is shown where data representation of the class \(\text{Point}\) is specified by mean of two fields that represent Cartesian coordinates:

\[
\text{class Point} \\
\text{state cart (x : Float, y : Float)}
\]

For the moment, observable properties of any instance of \(\text{Point}\) are \(x\) and \(y\) and points can be represented by expressions \(\text{cart}(a, b)\) where \(a\) and \(b\) are instances of \(\text{Float}\).

Inheritance

The construct to specify that a class is a subclass of other classes is the following:

\[
\text{class } C \text{ inherits } C_1 \ C_2 \ldots \ C_n
\]

The informal meaning of the declaration above is that class \(C\) is a subclass of \(C_1, C_2, \ldots, C_n\) (with \(n \geq 0\)) and inherits properties from them. Although the meaning of subclassing is controversial we do not pretend to change its intended semantics in the object oriented paradigm so its actual meaning in SLAM-SL is similar to that in most of object-oriented languages:

- an instance of a subclass can be used in every place in which an instance of the superclass is allowed (subtyping), and
- properties in superclasses are directly inherited by subclasses (inheritance).
9.2 Background

A very important property, not so extended in object-oriented languages, is the following one: the behaviour of subclasses are consistent with the behaviour of the superclasses.

For the moment, let us show a pair of paradigmatic examples:

```class`` ColoredPoint inherits Point, Color

As expected from the declared inheritance relationship, ColoredPoint instances must inherits properties from Point and Color, properties like x in Point or is_red in Color.

The user can specify a condition that every instance of a subclass must fulfil in order to be considered a valid instance. This invariant condition is given over public properties of the superclasses. For instance in the specification

```class`` NoRedColoredPoint inherits Point, Color
```
```
invariant not self ← is_red
```

the invariant establishes that for every instance of NoRedColoredPoint its property is_red is false.

**Predefined collections**

Class Collection plays an important role in SLAM-SL. All predefined containers classes in SLAM-SL inherits from the class Collection and all instances of Collection can be quantified. SLAM-SL introduces several predefined quantifiers like universal and existential quantifiers, counters, maximisers, etc. The most important predefined classes that inherit from Collection are sequences and sets.

**Visibility**

Reserved words `public`, `private` and `protected` have been introduced in SLAM-SL with the usual meaning in object oriented notations for design and programming. By default, state names and fields are private while the methods are public.

**Behaviour modelling**

The SLAM-SL notation allows the user to distinguish among several kinds of methods:

**Constructors.** Constructors are class members that build new instances of the class.

**Observers.** Observers are instance members that observe instance properties.
Modifiers. Modifiers are *instance members* that *modify* the state of an instance.

Functions. Functions are *class members* that *observe* properties of the class.

The SLAM-SL semantics is *stateless*. This means that the classification above is, from the semantics point of view, artificial because methods will be interpreted as functions. From a pragmatic point of view, the classification given by the specifier is used in static analysis and code synthesis stages.

Methods are specified by given a precondition and a postcondition. Preconditions and postconditions are SLAM-SL formulae involving explicitly declared formal parameters and two implicit formal parameters: *self* and *result* where *self* represents the state of the instance before the message is received and *result* represent the result of the method (the state of the instance after the *execution* of the method if the method is a modifier). Obviously, when constructors or functions are being specified the formal parameter *self* is not accessible.

The syntax of SLAM-SL formulae is similar to first-order logic formulae, in fact, SLAM-SL specifications are axiomatised into first-order logic.

The specification of a method in SLAM-SL has the following scheme:

```
class A
...
method m (T_1, ..., T_n) : R
pre P(self, x_1, ..., x_n)
call m (x_1, ..., x_n)
post Q(self, x_1, ..., x_n, result)
sol S(self, x_1, ..., x_n, result)
```

A method specification involves a *guard* or *precondition*, the formula $P(self, x_1, ..., x_n)$, that indicates if the rule can be triggered, an *operation call scheme* $m(x_1, ..., x_n)$, and a *postcondition*, given by the formula $Q(self, x_1, ..., x_n, result)$, that relates input and output states. The informal meaning of this specification is given by the following formula:

$$\forall s, x_1, ..., x_n (s \leftarrow \text{pre}_m(x_1, ..., x_n) \Rightarrow s \leftarrow \text{post}_m(x_1, ..., x_n, s \leftarrow m(x_1, ..., x_n)))$$

where

$$s \leftarrow \text{pre}_m(x_1, ..., x_n) \triangleq P(s, x_1, ..., x_n) \land \text{inv}(s)$$
$$s \leftarrow \text{post}_m(x_1, ..., x_n, r) \triangleq Q(s, x_1, ..., x_n, r) \land \text{inv}(r)$$

---

3The reserved word `method` represents any reserved word for the different kind of methods: `constructor`, `observer`, `modifier` or `function`.
Precondition, call scheme and postcondition must be considered the specification of the method. The procedure to calculate the result of the method is called a solution in the SLAM-SL terminology and it has been indicated by the reserved word sol followed by the formula $S(self,x_1,\ldots,x_n,result)$. Notice that the formula is written in the same SLAM-SL notation, but must be an executable expression (a condition that can be syntactically checked). The objective is that the SLAM-SL compiler to synthesise efficient and readable imperative code from solutions. Solutions must be considered as a refinement of the postcondition and the user, with the help of the system, must prove that every solution entails its postcondition $\text{post}_m$ (i.e. invariant included).

Some shorthands help the user to write formulae concisely and readably: self identifier can be omitted for accessing attributes, explicit function definitions, as in VDM, are allowed, and unconditionally true preconditions can be skipped.

Let us show an example of specification of a sortable sequence. A sortable sequence is a generic class and its type argument must inherit from the predefined class Ordered (that introduces a partial order relation):

```
class SortableSeq<X inherits Ordered> inherits Seq<X>
```

Constructor empty creates instances with the property length inherited from Seq equal to 0:

```
constructor empty
call empty
post result ← length = 0
```

The distinction between postconditions and solutions is crucial for code generation. A proof obligation establishes that the solution entails the postcondition. Code is obtained from solutions. The fact that both formulae are written in the same language has a number of advantages: i) it is a very abstract way of defining operational specifications from the user point of view, ii) it is easier to manipulate for optimisation of generated code, and iii) the task of ensuring the correctness property is easier.

The following example is the specification of a sorting method for sortable sequences where a postcondition and a solution is offered:

```
modifier sort
call sort
post result ← isPermutation(self) and result ← isSorted
sol result = self if self ← length < 2
    and result = self ← tail ← sort ← insertSort(self ← head)
    if self ← length > 1
```
Methods *isPermutation*, *isSorted* and *insertSort* are specified below. Methods *length*, *tail* and *head* are inherited from class *Seq*.

The mentioned proof obligation in the SLAM-SL underlying logic is

\[(\text{length}(l) < 2 \Rightarrow \text{sort}(l) = l) \land \]
\[(\text{length}(l) \geq 2 \Rightarrow \text{sort}(l) = \text{insertSort(\text{sort(tail}(l)), head(l)))} \]
\[\Rightarrow \]
\[\text{isPermutation(\text{sort}(l), l) \land isSorted(\text{sort}(l))}\]

The above specification of method *sort* used the following specification of methods:

**observer isSorted** : Boolean
**call isSorted** =

- self ← length < 2
- or (self ← elementAt(1) ← self ← elementAt(2)
  and self ← tail ← isSorted)

given as an explicit (functional) definition,

**observer count (X)** : Integer
**call count(x)** = countQ quantifies x = y with y in self

by using the predefined quantifier that counts elements in a collection with a given property,

**observer isPermutation(Seq<X>)** : Boolean
**call isPermutation**
**post**

- forallQ quantifies self ← count(x) = result ← count(x) with x in self
  and forallQ quantifies self ← count(x) = result ← count(x) with x in result

by using the universal quantifier, and

**modifier insertSort(X)**
**pre** self ← isSorted
**call insertSort(x)** =

- if self ← length = 0 then self ← cons(x)
- else if self ← head ← x then self ← insertSort(x) ← cons(x)
- else self ← cons(x)
Quantification

In SLAM-SL some constructs have been added in order to make writing of executable specification easier to write. One of those constructs consists of the generalisation of the quantifier concept.

In standard logic, the meaning of a quantified expression \( \forall x \in C(P(x)) \) is the conjunction \( true \land P(x_1) \land P(x_2) \land ... \) with each \( x_i \) in \( C \). The quantifier \( \forall \) determines the base value \( true \) and the binary operation \( \land \). In SLAM-SL we have extended quantified expressions with the following syntax:

\[
q \text{ quantifies } e(x) \text{ with } x \text{ in } d
\]

Where \( q \) is a quantifier that indicates the meaning of the quantification by a binary operation (let us call it \( \otimes \)) and a starting value (let us call it \( b \)), \( d \) is an object of a special predefined class \( Collection \), \( x \) is the variable the quantifier ranges over, and \( e \) represents the function applied to elements in the collection previous to computation. The informal meaning of the expression above is:

\[
b \otimes e(x_1) \otimes e(x_2) \otimes e(x_3) \otimes ...
\]

The abstract class \( Collection \) in SLAM-SL has the following interface:

\[
\text{class } Collection <T> \\
\text{observer traversal : Seq<T>}
\]

In SLAM-SL the user can specify the way in which a collection is traversed by inheriting from \( Collection \) and by specifying the way in which it is traversed. For instance, if the collection is a tree, it can be traversed in three different depth-first ways: preorder, inorder, and postorder.

The abstract base class for quantifiers has the following interface:

\[
\text{class } Quantifier<Element, Result> \\
\text{state quantifier } (public accumulated : Result) \\
\text{modifier next (Element)}
\]

and a pair of concrete quantifier specifications are:

\[
\text{class } Forall \text{ inherits } Quantifier<Boolean,Boolean>
\]

\[
\text{constructor forallQ} \\
\text{post accumulated = true}
\]

\[
\text{modifier next (Boolean)}
\]

---

\( Seq \) is the generic predefined class for representing sequences
call next(c)
post result ← accumulated = accumulated and c

class Count inherits Quantifier<Boolean,Integer>

constructor countQ
post accumulated = 0

modifier next (Integer)
call next(c)
post result ← accumulated = accumulated + if c then 1 else 0

Reflection

A SLAM-SL program is a collection of specifications that defines classes and their properties: name, relationships with other classes, and methods. Relationships with other classes are the inheritance relationship, and aggregation, or composition among classes, the last defined in state specification.

In this section the specification of SLAM-SL classes, properties, expressions, etc. are presented. Authors are sure that the reader understands that the specification of any construct needs the specification of the others so we will need to refer constructs that have not been specified yet.

class Class

public state mkClass (name : String,
  inheritance : {Class},
  inv : Formula,  
  states : {State},
  methods : {Method})

invariant
  forallQ quantifies s ← noCycle({}) with s in inheritance
  and forallQ quantifies m1 ← differ(m2) if m1 /\= m2
  with m1 in methods, m2 in methods

observer noCycle ({Class}) : Boolean
call noCycle(c) = not self in c
  and forallQ quantifies s ← noCycle(c ← add(self))
  with s in self ← inheritance

For modelling classes, we have made a natural reading of ‘what a class is’: a name, an inheritance relationship, an invariant, and its properties (states and meth-
ods), respectively: a string, a collection of instances of \textit{Class}, an instance of \textit{Formula}, a collection of instances of \textit{State} and a collection of instances of \textit{Method}. The syntax \{\textit{X}\} or \textit{[X]} is used to denote sets (respectively sequences) of type \textit{X}.

The invariant in \textit{Class} establishes that

- there is no cycle in the inheritance relationship,
- properties are correctly specialized: method overloading is allowed, but there must be an argument of different type. Notice that thanks to this declarative specification SLAM-SL is able to identify those properties that a class must fulfil what is much more expressive and powerful than the reflective features of Java or C# that are merely syntactic.

Among the interesting methods of classes, let us show a couple of them. Whether a class is just an interface is detected by checking if among the properties there is no states or constructors defined and if all the methods are undefined. Finally, a class is a subtype of another one if the latter can be found in the inheritance sequence of the former.

\begin{verbatim}
public observer isInterface : Boolean
call  isInterface =
    states = {} and forallQ quantifies m←undefined with m in methods

public observer isSubtype (Class) : Boolean
call  isSubtype(c) =
    c = self
    or existQ quantifies cl←isSubtype(c) with cl in inheritance
\end{verbatim}

\textbf{Formulae}

SLAM-SL formulae and expressions are the heart of SLAM-SL specifications. Therefore we discuss reflective features related to \textit{formula} management what, at the same time, gives an idea about how a SLAM-SL formula is. The SLAM-SL runtime environment can manage formulae in the same way the compiler does, this means that formulae can be created and compiled at runtime so the user can specify programs that manage classes and class behaviours. The following specification of formulae reflects its abstract syntax in SLAM-SL:

\begin{verbatim}
class Formula

public state mkTrue
\end{verbatim}
public state mkFalse
public state mkNot (f : Formula)
public state mkAnd (left : Formula, right : Formula)
public state mkOr (left : Formula, right : Formula)
public state mkImpl (left : Formula, right : Formula)
public state mkEquiv (left : Formula, right : Formula)
public state mkForall (var : String, type : Class, qf : Formula)
public state mkExists (var : String, type : Class, qf : Formula)
public state mkEq (lexpr : Expr, rexpr : Expr, type : Class)
public state mkPred (name : String, args : [Expr])

public observer wellTyped (ValEnv) : Boolean
    call wellTyped(env) =
        (is_mkTrue or is_mkFalse)
    or
        ((is_mkAnd or is_mkOr or is_mkImpl or is_mkEquiv) 
            and left ← wellTyped(env) and right ← wellTyped(env))
    or isMkNot and f ← wellTyped(env)
    or isMkEq and leexpr ← isSubtype(type) and reexpr ← isSubtype(type)
    or (isMkForall or isMkExists) and qf ← wellTyped(env ← put(var, type))
    or isMkPred and forallQ quantifies env ← get(name) ← argSig(i)
        ← isSubtype(args ← type(env))
        with i in [1 .. args ← length]

public modifier substitute (String, Expression)
    call substitute (var, expr)
    post result = self if is_mkTrue or is_mkFalse
        and result = mkNot(f ← substitute(var, expr)) if is_mkNot
        and result = mkAnd(left ← substitute(var, expr),
            right ← substitute(var, expr)) if self ← is_mkAnd
        and result = mkOr(left ← substitute(var, expr),
            right ← substitute(var, expr)) if self ← is_mkAOr
        ...

public observer isExecutable: Boolean
    call isExecutable =
        is_mkEq and leexpr = mkVar("result") and reexpr ← isExecutable

Class Formula represents the abstract syntax of SLAM-SL formulae that are those in the underlying logic plus the introduction of meta names for formulae. Methods have been added for checking if a formula is well typed, for substituting variables with expressions and for checking if a formula is executable.
Properties

The classes modelling properties are called State, and Method. Its models are the following:

```plaintext
class State
state mkState (name : String, attributes : { Attribute }, inv : Maybe<Formula>)
  invariant forallQ quantifies a1←differ(a2) if a1 /\= a2
      with a1 in attributes, a2 in attributes
```

In SLAM-SL, a composition relationship among classes is defined by the state specification. A state defines attributes that are the internal representation of the class instances. A state can have an invariant that establishes properties of the attributes and/or relationships between them.

```plaintext
class Method
public state mk_method (kind : MethodKind, visibility : Visibility , name : String, signature : Signature, precondition : Formula, postcondition : Formula, solution : Maybe<Formula>)

public observer type_sig : [ Class ]
call type_sig = mapQ quantifies d←type with d in sig

observer invocation : [String]
call invocation = mapQ quantifies d←name with d in signature
```

In the class Method, we have also introduced a couple of useful operations: constructing a method, abstracting the type signature just using the argument types (the names are almost irrelevant except for the pre and postconditions), and composing a method call with the argument names.

On top of them, we can describe a number of interesting operations on methods. The first one (isCompatible) indicates when two methods are equivalent (same name, types and equivalent pre and postconditions). The second one (canInherit) specifies when a method can override another definition. They must have a coherent definition (same name and arguments/return type) and the inheritance property must hold.

```plaintext
public observer is_compatible (Method) : Boolean
call is_compatible (m) =
    kind = m←kind and name = m←name
```
and type_sig = m←type_sig
and return = m←return
and (prec1 implies prec2)
and (post2 implies post1)
where
    prec1 = orallQ quantifies r←get_prec
             with r in rules;
    post1 = andallQ quantifies r←get_prec implies r←get_postc
             with r in rules;
    prec2 = orallQ quantifies r←get_prec
             with r in m←rules;
    post2 = andallQ quantifies r←get_prec implies r←get_postc
             with r in m←rules

Finally, we specify operations to decide when two methods are really different (up to argument names) and when a method implements an interface method (i.e. precondition false):

public observer differ (Method) : Boolean
call  differ (m) =
    name /= m←name
or sig←length /= m←sig←length
or existsQ quantifies sig(i)←is_subclass_of(m←sig(i))
    width i in sig←dom
    width i in sig←dom
and return = m←return
and (prec1 implies prec2)
and (post2 implies post1)
where
    prec1 = orallQ quantifies r←get_prec
             with r in rules;
    post1 = andallQ quantifies r←get_prec implies r←get_postc
             with r in rules;
    prec2 = orallQ quantifies r←get_prec
             with r in m←rules;
    post2 = andallQ quantifies r←get_prec implies r←get_postc
             with r in m←rules
with i in sig—dom

\begin{verbatim}
public observer do_nothing : Boolean
   call do_nothing =
      existsQ quantifies (r←get_prec = false and r←get_postc = true)
         with r in rules
\end{verbatim}

For the sake of simplicity, we assume that all record components of classes 
\texttt{Class}, \texttt{Method} and \texttt{State} are public. In fact, good object oriented methodologies recom-
mend to make them private and to declare adequate methods to access them. We omit such definitions to avoid an overloaded specification.

Notice that what we have presented is only a subset of the full SLAM-SL spec-
ification, just selected to show the main elements of the language as well as to
make design pattern description easy to follow. The full reflective specification is
included in the \texttt{reflect} module of the SLAM-SL distribution, more details can be
found in \texttt{[66]}.

\section*{SLAM-SL sentences and substitution}

An instance of the class \texttt{Class} represents a SLAM-SL class, an instance of the class 
\texttt{Method} represents a SLAM-SL method, an instance of the class \texttt{Formula} represents
a SLAM-SL formula. Instead of using expressions based on constructors and meth-
ods, the user can write those instances by using SLAM-SL sentences directly. This
makes the specification much more concise. Let us show an example for represent-
ing the class \texttt{Point} by using constructors and methods:

\begin{verbatim}
mkClass ("Point",
   {},
   mkTrue,
   {mkState("cart",[mkField("x","Float") ,
      mkField("y","Float") ] },
   {})
\end{verbatim}

SLAM-SL introduces a syntax that allows the user to give the class by using the
SLAM-SL own syntax for classes:

\begin{verbatim}
<scode>
class Point
state cart (x : Float, y : Float)
</scode>
\end{verbatim}

Both expressions are equivalent.
Every SLAM-SL (sub)sentence representing any object oriented concepts can be given between `<scode>` and `</scode>` and its meaning is an instance of the class that models such a concept.

Substitutions has been added to SLAM-SL as a metalanguage capability. Its syntax is $S[x:=e]$ where $S$ is an instance that represent a SLAM-SL (sub)sentence, $x$ is a string to be substituted and $e$ is an instance that represent other SLAM-SL subsentence. The SLAM-SL compiler check that substitutions are well typed.

Let us show an example of substitution: the following expression

```<scode>
class Point
state cart (x : Float, y : Float)
</scode>`"Float" := `<scode>Integer</scode>`
```

is equivalent to this one

```<scode>
class Point
state cart (x : Integer, y : Integer)
</scode>`
```

### 9.2.2 Other formalizations of design patterns

The LePus project \[45,46\] develops an ambitious idea: a visual language for specifying design patterns. A design pattern is described by a (limited form of) verbal specification, and a diagram that includes the constructs and relations the design pattern involves, and the constraints it imposes on conforming implementations. A tool can read it and produces what they call a *trick*, basically an algorithm to manipulate programs. Of course, the work is in principle more general than our approach but we claim that we can get a similar power with a simpler technique, and that SLAM-SL can also be considered as a pattern language.

Some other papers \[6,92,114\] differ in their goals, and are more interested in describing temporal behaviour and relations between design patterns, by using variations of temporal logic: \[6\] is focused on the formalization of architectural design patterns based on an object oriented model integrated with a process oriented method to describe the design pattern; \[92\] is concentrated on communication between objects; \[114\] has the aim to describe the structural aspect of a design pattern.

As we said in Section 9.1, the work of Tokuda and Batory \[118,117\], already points out that some design patterns can be expressed as a series of program trans-
formations applied to an initial software state, where these program transforma-
tions are primitive object oriented transformations.

The work of Cinnéide [32] also points out that design patterns can guide the
refactoring process. A methodology for the construction of automated transfor-
mation, that introduces design patterns to an existing program preserving its be-
avour, is presented. The main difference between this approach and our proposal
is that we can detect the patterns to apply in a given design.

The detection of situations in which refactoring can be applied is what Mens
names bad smells in his paper [89].

Finally, [56] uses UML and OCL as specification languages for design patterns.
While the paper contains some useful ideas in order to develop a tool, it also hon-
estly shows the severe limitations of UML and OCL for this goal, and particular ex-
tensions are proposed.

9.3 Design patterns as class operations

A design pattern consists of the description of a valuable design that solves a gen-
eral problem. Strictly speaking, design patterns cannot be formalized because its
domain of application are problems. Nevertheless, relevant parts of design pat-
terns are susceptible of formalization: structure, participants and, more difficultly,
collaborations. Our proposal is to view design patterns as class (set of) operators
that receive a collection of classes that will be instances of (some) participants and
return a collection of classes that represents a new design.

In our model, a given (preliminary) design is the input of a design pattern. This
design is modelled as a collection of classes. The result of the operation is another
design obtained by (possibly) modifying the old classes, and potentially creating
new ones, according to the description of the design pattern.

For instance, let consider you have a collection of classes leaves (e.g. Line, Circle,
Rectangle, …) that share some operations (e.g. draw, rotate, resize, …) and you
want to compose all of them in a wider object that either has all of them as partic-
ular cases and also can collect some of them inside (e.g. a Figure). The Composite
pattern, considered as an operator, accepts classes (leaves) as input and returns two
new classes Component (merely an interface) and Composite (for the collection of
components) with the common operations as methods, and modifying classes in
leaves to inherit from Component.

More specifically, a design patterns is modelled as a class with a single function
apply that is a class operator. The precondition for this function collects the logical
conditions required to use the pattern with success. Basically, this means that the
pattern precondition establishes the *applicability* of the pattern, talking in terms of the sections in the pattern description. For instance, in the *Composite* pattern we mentioned above, the precondition needs to ensure that all the classes in *leafs* define the common methods with the same signature.

On the other hand, the postcondition encompasses most of the elements of the *intent* and *consequences* sections of the pattern description. In the *Composite* pattern, the postcondition establishes that the input classes *leafs* now inherit from *Component* and classes *Composite* and *Component* are introduced, the first one inheriting from the second one. The *Composite* state is a collection of *Components* and its methods are described by iterative calls to the corresponding *leafs* methods.

In order to describe all these elements, the reflective features play a significant role because they allow inspecting argument classes and describing new classes as result [66]. Design patterns can be described by a (polymorphic) class *DPattern*. The method *apply* describes the behaviour of the pattern by accepting a collection of classes as arguments (the previous design) and returning a new collection of classes. This method can describe a general behaviour of the pattern, or can describe different *applications* of the pattern with different *consequences*, each one in a different rule. The class argument (coming from the polymorphic definition) is occasionally needed to instruct the pattern about the selection of classes, methods, etc. that take part in the pattern. This argument is stored in the internal state of the class *DP*:

```java
class DP <Arg>

private state dp (protected arg : Arg)

public observer apply ([Class]) : [Class]
```

Inheritance is used to derive concrete design patterns. It is also needed to instantiate the type argument and supplying a value for the state. Notice that design patterns variants are easily supported in our model through inheritance.

Let us describe in detail the method by using a couple of examples taken from [54]. A graphical description complements the formal definition using an UML based notation taken again from [54]. A preliminary version of these ideas can be found in [61], where a good number of examples (*AbstractFactory*, *Bridge*, *Strategy*, *Adapter*, *Observer*, *TemplateMethod*, ...) are described. Most of them can be found in Appendix 9.6. This collection clearly shows the feasibility of our approach.

---

[5]: without any contribution about how to reason with design patterns or how to develop a tool.
### 9.3.1 Composite pattern

The *Composite* pattern is part of the object structural patterns. It is used to compose objects into tree structures to represent part-whole hierarchies. Using the pattern, the clients treat individual objects and compositions of objects uniformly.

When we treat it as a class operator, we have the collection of basic objects as argument (called the *leaves*). The result “invents” two new classes *Component* and *Composite*. *Component* is just an interface for all the common methods in all the leaf classes plus some methods to add, remove and consult internal objects. *Composite* inherits from *Component* and stores the collection of components. The result also collects all the classes in *leaves* that are modified by inheriting from *Component*. The methods in *Composite* can be grouped in two parts. On one hand, we have methods to *add* and *remove* a component, and also to consult the ith element in the component collection (*getChild*). On the other hand, we have all the common methods of the *leaves* that have a very simple specification by iterative calling the same operation in all the components. See Figure 9.1 and Figure 9.2 for a complete SLAM-SL specification.

![Composite class diagram](image)

**Composite arguments**  
**Composite results**

![Composite class diagram](image)

**Figure 9.1: Composite class diagram.**

### 9.3.2 Decorator pattern

The *Decorator* pattern is classified as object structural and it is used to attach additional responsibility to an object dynamically. It can be seen as the following class operator: A collection of concrete components and a collection of decorators are used as arguments. They share some operations that the pattern abstracts in two steps. First of all, a new *Decorator* class abstracts the operation of the decorators.
**class** Composite inherits DP<Unit>

**public** constructor composite (Unit)
call composite (unit)
post result ← arg = unit

**public** observer apply ([Class]) : [Class]
let common_meths = {m with cl in leafs | m in cl←methods} with m in cl.methods)
pre (not leaf←isEmpty ) and (not common_meths←isEmpty)
apply (leafs)
post result = [component, composite]
+ [c \ inheritance←insert(component) with c in leafs ]

where

component =
mkClass("Component", {Component}, <slamcode>true = true</slamcode>,{},
{m \ prec = <slamcode>
false and q = true
</slamcode>[q := postc]
| m in commonMethods}
+ {create, add, remove, getChild})

composite =
mkClass("Composite", {}, <slamcode>true = true</slamcode>,
{children}, {create, add, remove, getChild}
+ {gen(m) | m in commonMethods})

children = <slamcode>state mkComposite (children : [C])</slamcode>
[C := component]

create = <slamcode>
constructor create
pre true = true
call create
post result = {}
</slamcode>

add = <slamcode>
modifier add (C)
pre true = true
call add(c)
post result = children←insert(c)
</slamcode>

remove = <slamcode>
modifier remove (C)
pre true = true
call remove(c)
post result = children←remove(c)
</slamcode>[C := component]
Then another newly created class \textit{Component} abstracts the operation either for the concrete components and for the decorator.

The class argument is used to split the sequence of classes into the concrete components and the decorators. Concrete components are forced to inherit from \textit{Component}, while decorators inherit from \textit{Decorator} and modifies the common methods to add a call to the decorator operation. The \textit{Decorator} class contains a \textit{Component} in the state and offers the common methods as public. They are implemented as simple calls to the equivalent operations in the stored component. Finally, \textit{Component} is merely an interface for the common methods.

class Decorator_DP inherits DPattern\textless\text{Natural}\textgreater

public constructor decorator (Natural)
call decorator(n)
p= result ← arg = n

public observer apply ([\{Class\}] : [Class])
let common_meths = \{ m with cl in classes | m in cl\rightarrow\text{methods}\}
pre (classes ← length > arg) and (not common_meths ← isEmpty)
call apply (classes)
p= result = [component, decorator]
+ mapQ quantifies
  c \; \text{inh} ← insert (decorator)
  \; \text{methods} = mapQ quantifies add\_call (m, c ← methods)
    with m in c ← methods
  with c in concrete\textunderscore classes
+ mapQ quantifies c \; \text{inh} ← insert (component)
  with c in concrete\textunderscore classes
where
concrete\textunderscore classes = classes ← prefix (arg);
decorators = classes ← suffix (arg);
component =
mk\_Class
  ("Component", {}, {}),
  mapQ quantifies
  m \ (mapQ quantifies r \; \text{prec} =$false$
    \; \text{postc} =$true$
      with r in rules)
  with m in common\_methods);
decorator =
mk\_Class
("Decorator",
{mk_State(
    [mk_Field(
        "component", component)]),
    {component},
    mapQ quantifies
    m \ mapQ quantifies
    r \ postc = (<slamslcode>
        result = component←mk_Call(m←name,
                                    m←parameters)
    </slamcode>)
    with r in rules
    with m in common_methods);

As we can see, thanks to its declarative reflection features, SLAM-SL can be considered as a design patterns language. Once you can model a pattern as a class operator, SLAM-SL can be used to specify it and this specification can be used to instruct the associated tool to apply the pattern to existing designs and programs.

9.3.3 Different modelling possibilities

For several patterns, as Factory Method, the most general case specification is a thorny issue, nevertheless one can specify in a simpler way, different situations in which the pattern could be applied with different consequences. It can be made, as we said in Section 9.3, by generating a different rule for each situation you want to manage, i.e., a different precondition-postcondition pair.

For example, you can find the situation in which several classes in a hierarchy implement a common method similarly except for an object creation step so you can apply a refactoring by the Factory Method. The rule precondition specifies this situation in a easy way by reflection. The postcondition establishes that a new Abstract class will be created, a inheritance relation will be created between initial classes and the new one, in the new class two methods will be placed: a new abstract factory method and the common method in which the object creation step will be replaced by a call to the former, and in the initial classes the common method will be removed as long as the factory method will be added with a call to the concrete object constructor.

Now, if the designer finds a new application of the Factory Method pattern, she will only have to add a new rule to describe this application and its consequences.

---

6This example, "Introduce Polymorphic Creation With Factory Method", has been taken from [73]
9.3.4 Design patterns composition

Viewing design patterns as operators over classes allows us to create *new* design patterns by *composition*. For instance, the composite design pattern can be applied to a collection of *leafs* and then a decorator can be applied to the new design.

In the case study presented in chapter two in [54], the design of a document editor is guided by the application of several design patterns. Some of those design patterns are applied to (a part of) the result of a previous one. Because design patterns have been modelled as class operators, we can specify the composition of them:

```plaintext
composite = instance (Empty);
glyph = composite ← apply([border, scroll, character, rectangle, polygon]);
decorator = instance (3);
mono_glyph = decorator ← apply(glyph ← prefix(3))
```

9.3.5 Application: reasoning with design patterns

An immediate application of the formalisation of design patterns is to reason about certain properties. In this section, some properties that can be stated with our formalism are presented.

**Commuting Patterns**

Proving that the application of two patterns commutes is less relevant for the user at least at the design level, but it is useful for a software team, to know that these tasks are interchangeable in time if recommended by the project planning.

Additionally, the look of a design can get dirty or complicated besides the application of a pattern, in this case, it can be desirable to postpone its application to the end of a commutative sequence of pattern applications.

So, given two design patterns $dp_1$ and $dp_2$, we say that they commute if the following property written in SLAM-SL holds:

```plaintext
forall design : [Class] (  
    dp_2 ← apply(dp_1 ← apply(design)) = dp_1 ← apply(dp_2 ← apply(design))  
    if (dp_1 ← pre_apply(design) and dp_2 ← pre_apply(design))  
)
```

An example of two patterns that *commutes* are *Adapter* and *Decorator*. We omit the proof to save space, but it is straightforward. Let us discuss the influence of
this fact: Consider the example of a drawing editor, as in [54, Chapter 4], that lets you draw and manipulate graphical elements (lines, polygons, text, etc.). The interface for graphical objects is defined by an abstract \textit{Shape} class. Each elementary geometric \textit{Shape}'s subclass is able to implement a \textit{draw} method but not the \textit{TextShape} one. Meanwhile, an off-the-self user interface toolkit provides a sophisticated \textit{TextView} class for displaying and editing text. Besides, this toolkit, should let you add properties like borders or scrolling to any user interface component. In this example, we can apply two design patterns: the \textit{Adapter} one in order to define \textit{TextShape} so that it adapts the \textit{TextView} interface to \textit{Shape}'s; the \textit{Decorator} one in order to attach “decorating” responsibilities to individual objects (scroll, border, etc.) dynamically.

In the previous example, the application of the \textit{Adapter} design patterns only adds an association relation between \textit{TextView} class and \textit{TextShape} (as we said in section [9.1]). Whereas the application of \textit{Decorator} design patterns transforms the design in a more complicated one (as we said in section [9.3.2]). So in order to obtain simpler intermediate designs, \textit{Decorator} would be applied the last.

In general, it is not an usual case that two patterns directly commutes, but it is more frequent the fact that they commute after some trivial modifications (i.e. permutation of arguments, renaming of operations, etc.). As these modification can be specified in the specification language itself, more general properties can be proved.

### More General Patterns

Another interesting property might be to detect that a design pattern is an instance of a more general one. In the \textit{Pattern Languages of Program Design} meetings, it is usual that a pattern proposal is rejected with the argumentation that it is an instance of an existing one. We offer the basis for formally prove (or disagree with) such statements. However, this does not means that the concrete pattern is useless. Firstly because we are not specifying all the components of a pattern, and two operationally similar patterns can differ in the suggestions of usage, and this difference could be crucial for a software engineer. Secondly, because the general pattern could be complicated enough, or rarely used in full, and the simpler version could be more adequate for being part of the expertise of the practitioner. Nevertheless, the tool can detect that storing the concrete design pattern is not needed because it is an instance of the other pattern which specification can be used instead.

A design patterns \textit{cdp} of type \texttt{CDP<CArgs>} is an instance of a more general design pattern \textit{gdp} of type \texttt{GDP<GArgs>} (where \texttt{CArgs} is a subtype of \texttt{GArgs}) can be characterised through the following SLAM-SL formula:

\[
\forall \text{design} : \text{[Class]} (\ldots)
\]
Our specification of the design pattern *Composite* is an instance of the design pattern *Composite* presented in [54, Chapter 4]. The general version allows several Composite classes each one with its own behaviour. We can specify it in the following way:

```plaintext
class CompositeGOF
    inherits DPattern<[Class]>

public constructor
    compositeGOF ([Class])
    pre ...
    call compositeGOF(composites)
    post result ← arg = composites

public apply ([Class])
    pre ...
    call apply(leafs)
    post ...
```

and formally prove that composite is an instance of compositeGOF([]). Again we omit the proof.

**Other properties**

Other interesting examples of properties that can be easily stated for reasoning about design patterns and systems are:

**Pattern Composition.** To find out that a pattern is the composition of two patterns can be interesting. This does not preclude to exclude the composed pattern from the catalogue, but an implementation can take advantage of this feature. A design pattern \(dp\) is the composition of two design patterns \(dp_1\) and \(dp_2\) if:

```plaintext
forall design : [Class] (  
    dp ← apply(design) = dp_2 ← apply(dp_1 ← apply(design))  
    if dp ← pre_apply(design)  
)
```
Pattern Implementation. An additional usage, out of the scope of this work, is to prove that a concrete piece of software really implements a pattern. A design *design* is the result of the application of a design pattern *dp* if:

\[
\exists \text{original : [Class]} \quad (\text{dp} \leftarrow \text{post_apply(original, design)})
\]

Refactoring. Given a system design we can explore if a subsystem can be refactored by the application of any design patterns in a collection of previously specified design patterns:

\[
\text{filterQ quantifies } \quad \text{dp} \leftarrow \text{pre_apply(subsystem)}
\]

with subsystem in design←subSequences

Pattern's piece. There are designs in which we can find that a piece of a design pattern has been applied but not the whole one. So the design can be refactored applying only the remaining part. In these cases, we can find out if a design pattern *cdp* is a component (or a piece) of another design patterns *wdp*:

\[
\forall \text{design : [Class]} \quad (\text{wdp} \leftarrow \text{pre_apply}(design)
\]

and exists sub_design : [Class] (sub_design←is_in(design)

and \text{dp}←\text{pre_apply}(design) \implies \text{cdp}←\text{pre_apply}(sub_design)

and \text{dp}←\text{apply}(design) \implies \text{cdp}←\text{apply}(sub_design))

In the same way we do looking for a pattern implementation, we can find out if a piece of a pattern has been applied to a design and next, find the remaining part to be applied.

9.3.6 Application: design patterns in a development environment

Once we have the modelling of design patterns as class operation, it is relatively easy to incorporate them as a refactoring tool into development and design environments. Let us describe how to achieve this goal. An additional feature of your favourite development environment (Visual Studio, Visual Age, Rational Rose, ...) can allow the user to select a design pattern and to provide the arguments to it. Figure 9.3 shows an example in C++, where the decorator pattern has been chosen
9.3 Design patterns as class operations

to organize the responsibilities in a flexible way of three existing display classes: *Border*, *Scroll*, and *TextView*. The first two are selected as “decorators” (they just allow to display things in different ways), while the third one is classified as a concrete component (is just a concrete way to display something, in this case a text).

![DesignPatterns - Microsoft Visual C++ [design] - TextView.h](image)

Figure 9.3: A tool for using design patterns

Once the pattern is applied, the existing code is automatically modified and the new classes (if any) are generated as depicted in figure 9.4. The pattern preconditions are checked and in case of failure a message explaining the reason is displayed.

Tools for incorporating design patterns into a project has already been developed, but they depart from the idea that the designer has in mind the pattern to be used before generating any code. The tool generates a code/design skeleton, and then the user provides the particular details for each class. Obviously, our modelling can be used also for this purpose (and the tool modified accordingly with little effort), but we have preferred to focus on a refactoring point of view. Rarely the designer selects a design pattern from the very beginning, but they are inserted later when the design complexity grows. Additionally, our ideas reinforce the reusability of existing code, because the argument classes can be part of a library.
9.4 Future Trends

Although we consider our approach very promising, some additional work can be done. Our future work will address the following issues:

- Obviously, it is important to provide a formalization of a more significant collection of patterns (even if we have already described a good number of them). We also plan to reformulate the descriptions in other specification languages, more widely used, like Z or Maude. Notice that this is a simple translation (except that the reflective features need to be modelled either from the scratch – Z – or augmenting the already existing – Maude). However,
the new formulations can make easier to include design patterns in existing tools for these languages.

- One of the most promising application is those related with the development of tools. We plan to fully develop efficiently the tools described, exploring in concrete applications the real impact of our approach.

Although we have displayed how to incorporate design patterns into a development tool, it can be done in a similar way in a design tool, like Rational Rose for UML. In this case, the system generate new diagrams and OCL specifications.

In fact, we have only shown the easiest tool possible, but many extensions are possible. In particular, an additional feature could be to select some classes and then leave the system to find the pattern that can be applied to them (i.e. the preconditions are fulfilled).

- Although the reflexive features of SLAM-SL allows for many semantical treatment of specifications, it is true that it is possible to go deeper on this approach. Many interesting issues of SLAM-SL (for instance, proving that solutions implies the postconditions, or that the inheritance relation is fulfilled) needs for "hard" reasoning on formulae. This means that some non trivial mathematical proofs are needed. Either we leave them to the responsible for the specification (human), or we use some automatic theorem proving tools (computer, or mixed). We want to explore this second approach in the next future. This allows us to include more semantical conditions in our modelling of object oriented aspects. For instance, the do_nothing method just check syntactically that the postcondition is exactly the atom false while it can be checked that the postcondition is logically equivalent to false.

9.5 Conclusion

We have proposed a formalization of design patterns by viewing them as operators between classes. The idea is not new and has circulated in the design patterns community for some time[7]. However, to our best knowledge we have not found a development of the technique.

The precise definition of software design patterns is a prerequisite for allowing tool support in their implementation. Thus coherent specifications of patterns are essential not only to improve their comprehension and to reason about their properties, but also to support and automate their use.

[7] For instance, Prof. John Vlissides mentioned it in a panel at POPL'00.
If we measure our proposal following the criteria of A.H. Eden in his FAQ page on *Formal and precise software pattern representation languages* [50] we can establish that our approach is *expressive*, because conveys the abstraction observed in patterns, *concise*, at least more than other formalizations, *compact* because it is heavily focused on relevant aspects of patterns, and *descriptive* in the sense that we can apply our model to any pattern [8].

It is worth mentioning that we are not claiming that our approach is the “unique” or “the most appropriate” way to formalize design patterns. In fact, different formalizations focused on a particular aspects yield to different tools, properties to prove, aspects to understand, etc.

Our formal understanding of patterns gives support for tools that interleave with existing object oriented environments. The main difference between our tool and the one proposed in [45, 46] is that our method is adapted to ”every day” existing CASE environments instead of a totally new application, so the user of existing tools can benefit from design patterns almost for free. We can also can apply our tool to existing code, even stored in libraries. However, both kind of tools are not alternative but complementary, as the output of the [45, 46] tool could be used to instruct our tool.

In summary, our work tries to add some value to design pattern modelling: the possibility of reasoning with them, understanding, refactoring, etc.

### 9.6 Formalisation of DP in SLAM-SL

In this appendix we will show the formalization in SLAM-SL of some DP.

#### 9.6.1 Abstract Factory (Figure 9.5)

The *Abstract Factory* pattern is part of the creational series. It provides an interface for creating families of related or dependent objects without specifying the concrete classes. We can see this pattern as an operator that takes the “factories” classes as argument and produces a new abstract class `AbstractFactory` for interfacing them. The old classes are modified to inherit from this new class.

The class argument collects the methods to abstract. For simplicity we have consider it as a set of methods, although the pre and postconditions and the arguments names are not relevant.

---

[8] though for some patterns if you model the most general pattern it may lead to a less *concise* formalization than if you formalize the specific ones.
The precondition of apply basically establishes that the methods to abstract are present with the same format in all the factory classes.

```plaintext
class Abstract_Factory_DP inherits DP <{Method}>

public observer apply ([Class]) : [Class]
pre not factories.is_empty and
   forall m in arg with
      forall f in factories with
         exists cm in c.meths with not m.differs (cm)
call apply (factories)
post result = [abstract_factory] +
   map f in factories with f \ inh. insert (abstract_factory)
where
   abstract_factory =
      <slamcode>class Abstract_Factory</slamcode>
      \ meths = map m in args
         \ with m \ prec = <slamcode>false</slamcode>
         \ postc = <slamcode>true</slamcode>
```

Figure 9.5: Abstract Factory pattern specification.

### 9.6.2 Bridge (Figure 9.6)

It is one of the object structural patterns and is used to decouple an abstraction from its implementation. The operator takes a class as argument and returns two classes. One is the implementation class that is basically the original one. An abstract class is created by modifying the state of original one. It is replaced by a single attribute belonging to the implementation class. Methods are rewritten as merely calls to the correspondent ones in the implementation class.
**class** Bridge_DP inherits DP <Unit>

**public** observer apply ([Class]) : [Class]
pre  classes←length = 1
call  apply (classes)
post result =
   [impl,
    cl \ st = <slamcode>state (imp : Impl)</slamcode>
    \ meths = map m in cl←meths
    \ with m \ postc = <slamcode>result = x</slamcode>
    \ [x := imp←m←name(m←Call)]]
where impl = cl \ name = cl←name + "_Impl",
    cl = classes←first

Figure 9.6: Bridge pattern specification.

### 9.6.3 Strategy (Figure 9.7)

This object behavioural pattern can be used when we have a family of algorithms for the same purpose and we want to encapsulate each one, and make them interchangeable. So, the input classes share some methods and a new class *Strategy* is created to provide an interface for them. The old classes need to inherit from the *Strategy* class.

The *apply* precondition need to ensure that there are really common methods to abstract. Again, no argument class is really needed and the empty class is used instead.

### 9.6.4 Adapter (Figure 9.8)

The *Adapter* pattern belongs to the object structural patterns. It converts the interface of a class *adaptee* into another interface *target* clients expect. It is done by introducing a class *adapter* that inherits from both classes. Every method that needs to be adapted is specified by making a call to the corresponding *adaptee* method.
module examples.dps

class Strategy_DP inherits DP (Empty)

public observer apply ([Class]) : [Class]
let (m in common_methods equiv forall cl in leafs with m in cl.methods)
pre :- not common_methods.is_empty
call apply (classes)
post :- result = [strategy] +
    map cl in classes with cl \ inh. insert (strategy)

where
    strategy = <slamcode>class Strategy</slamcode>
    \ methods = map m in common_methods
        with m \ prec = <slamcode>false</slamcode>
        \ postc = <slamcode>false</slamcode>

Figure 9.7: Strategy pattern specification.

The apply precondition states that target is an interface and the methods to be adapted are present in both classes.

The class argument relates methods between the target and the adaptee. It is done by a couple of functions: the first one accepts a target method as parameter and returns the corresponding method in the adaptee that implements this interface, while the second one adapts the arguments between both calls. Of course, the last choice is a simplification of the problem because in many cases the way to adapt a method call to the other is not merely a reorder of the arguments.

9.6.5 Observer (Figure 9.9)

The Observer pattern is included into the behavioural pattern, and defines a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically. We assume that we already have a collection of concrete subject classes and a concrete observer. The pattern
invents a class to store observers that is a superclass of all the concrete subject and also a class Observer to abstract the update operation of the concrete observer.

The class argument simply identifies the update method in the concreteObserver class. A class Observer is created just for interfacing this method and the concreteObservers inherit from this new class. Another Subject class is invented to store observers. The classes in concreteSubjects are forced to inherit from Subject and all the constructor and modifier methods includes a call to the Notify operation of the father. The specification is a bit long but easy to follow:
9.6.6 Template Method (Figure 9.10)

The TemplateMethod (class behavioural) pattern is applied to a single class to abstract a method that can be used as an skeleton of similar algorithms. We assume that the method is already implemented in the class. The pattern abstracts this method into a new class and eliminates it from the original one.

The formalization is straightforward by inventing a new class TemplateAbstractClass with the same functionality as the original one but with empty code for those methods that are not classified as templates. The concrete class is forced to inherit from the new class and the template methods are removed.

The class argument is a boolean function which decides the methods to abstract as template methods. The precondition plays an important role because it is needed that the template method are really templates, i.e. they only use predefined and public operations of the class.

9.6.7 Decorator (Figure 9.11)

The Decorator pattern is classified as object structural and it is used to attach additional responsibility to an object dynamically. It can be seen as the following class operator: A collection of concrete components and a collection of decorators are used as arguments. They share some operations that the pattern abstracts in two steps. First of all, a new Decorator class abstracts the operation of the decorators. Then another newly created class Component abstracts the operation either for the concrete components and for the decorator.

The class argument is used to split the sequence of classes into the concrete components and the decorators. Concrete components are forced to inherit from Component, while decorators inherit from Decorator and modifies the common methods to add a call to the decorator operation. The Decorator class contains a Component in the state and offers the common methods as public. They are implemented as simple calls to the equivalent operations in the stored component. Finally, Component is merely an interface for the common methods.

9.6.8 State (Figure 9.12)

The State pattern belongs to the object behavioral classification. It can be used to allow an object to modify its behavior when its internal state changes. The object will appear to change its class. When studied as a class operator, it takes a collection of concrete state classes as argument. All these classes are present in the result, except that they inherit from the State class described below. The result adds two
classes: one to abstract the behavior of all the concrete states, called *State*, that represents an interface containing all the common methods in all the concrete states. The second one is *Context* that is designed for calling state operations. It contains a *State* as attribute and all the common methods, described as merely calls to the corresponding operation of the attribute. This class can be refined by inheritance to introduce more functionality. The complete specification can be found in figure 9.12.

### 9.6.9 Builder (Figure 9.13)

The *Builder* pattern (belonging to the *object creational* patterns) is designed to separate the construction of a complex object from its representation, so that the same construction process can create different representations.

As a class operator, the *Builder* patterns takes a collection of concrete builders as an argument. Another class *director* is part of the arguments and it is assumed that it contains the algorithm to construct objects. It is also assumed that all the concrete builders share some operations that are used to build objects. Those methods are called *builders* and need to be defined in all the concrete builders. The argument class is a boolean function *isBuilder* that is applied to methods in the concrete builders, detecting if they are builders or not. Methods classified as builders are abstracted into the *Builder* class. The concrete builders appear in the result but they are forced to inherit from *Builder*. The *director* class is modified in the following way: once an attribute belongs to one of the concrete classes it is abstracted to the *Builder* class. See figure 9.13 for the detailed description.
class Observer_DP inherits DPattern <Method → Bool>

public observer apply ([Class]): [Class]
pre exists1 m in concreteObserver ← meths
    with (isUpdate (m) and updateMethods ← sig = [])
call apply (classes)
post result = [subject, observ, concreteObserver \ inh ← insert (observer)] +
    map cl in concreteSubjects with cl \ inh ← insert (subject) and
    meths = forall m in meths with addNotify(m)
where isUpdate = arg
    concreteObserver = classes (1)
    concreteSubjects = classes ← suffix(1)
    updateMethod = select m in concreteObserver ← meths with isUpdate (m)
    subject = makeClass ("Subject", [makeDec ("observers", [observ])],
        {observ}, true, {attach, detach, create, notify })
attach = makeMethod(modifier, public, "attach",
    [makeDec("ob", Observer)],
    true, ( result = observ ← insert (ob)))
detach = makeMethod(modifier, public, "detach",
    [makeDec("ob", Observer)],
    true, ( result = observ ← remove (ob)))
create = makeMethod(constructor, public, "create",
    emptyDec, true, (result = []) )
notify = makeMethod(modifier, "notify", emptyDec, true,
    ( result = map o in observers
        with o ← makeCall(updateMethod ← name())))
observ = makeClass("Observer", emptyDec, {}, true,
    {updateMethod \ prec = false and postc = true})
addNotify (m) = if m ← kind ← isConstructor or m ← kind ← isModifier
    then m \ postc = m ← postc and self ← notify
    else m
class Template_Method_DP inherits DP <Method → Bool>

public observer apply ([Class]): [Class]
pre exists m in concreteClass.meths with isTemplate(m)
call apply ([concreteClass])
post result = [templateAbstracClass,
        concreteClass \ st = [] and
        inh. insert (templateAbstracClass) and
        meths = filter m in concreteClass.meths
        with not isTemplate(m)]
where templateMethods = filter m concreteClass.meths with not isTemplate(m)
templateAbstracClass = makeClass ("TemplateAbstractClass",
        concreteClass.st, {}, true,
        map m in concreteClass.meths
        with modify (m))
modify (m) = if m in templateMethods
        then m
        else m \ prec = false and postc = true end
isTemplate = arg

Figure 9.10: Template Method pattern specification.
class Decorator_DP inherits DPattern<Natural>

public constructor decorator (Natural)
call decorator(n)
post result ← arg = n

public observer apply ([Class]): [Class]
let common_meths = {m with cl in classes | m in cl←methods}
pre (classes←length > arg) and (not common_meths←isEmpty)
call apply (classes)
post
result = [component, decorator]
  + mapQ quantifies
    c \ inh←insert(decorator)
      \ methods = mapQ quantifies add_call(m, c←methods)
        with m in c←methods
    with c in concrete_classes
  + mapQ quantifies c \ inh←insert(component)
    with c in concrete_classes

where
concrete_classes = classes←prefix (arg);
decorators = classes←suffix (arg);
component =
mk_Class
("Component", {}, {}),
mapQ quantifies
  m \ (mapQ quantifies r \ prec = $false$
       \ postc = $true$
     with r in rules)
    with m in common_methods);
decorator =
mk_Class
("Decorator",
 {mk_State(
  [mk_Field(
    "component", component)])},
}
class State_DP inherits DP <Empty>

public observer apply ([Class]) : [Class]
let (m in common_methods equiv forall cl in leafs with m in cl←methods)
pre not concrete_states←is_empty and not common_methods←is_empty
call apply (concrete_states)
post result = [context, abs_state] +
    map c in concrete_states with c \ inh←insert (abs_state)
where
    abs_state = make_class ("State",{},{},
        map m in common_methods
            with m \ map r in rules with r \ prec =$false$
                \ postc =$true$),
    context = make_class ("Context",
        {make_state (make_attribute (make_declaration
            ("stt",abs_state)))},
        {}, map m in common_methods with transfer (m)),
    transfer (m) =
        m \ map r in rules
            with r←put_postc ($result = stt←make_call(m←name,
                m←parameters)$)

Figure 9.12: State pattern specification.
9.6 Formalisation of DP in SLAM-SL

**class** Builder_DP inherits DP <Method → Boolean>

**public** observer apply ([Class]) : [Class]

let concrete_builders = classes ← prefix (1)

(m in common_methods equiv (forall cl in concrete_builders

with m in cl ← methods)),

is_builder = arg,

builder_methods = filter m in common_methods with is_builder (m)

pre (classes ← length > 2) and (not builder_methods ← is_empty)

call apply (classes)

post result = [builder] +

[director \ states = map d in director ← states

with abstract_to_builder (d)] +

map c in concrete_builders

with c \ inheritance ← insert (builder)

where
director = classes ← prefix (1),

builder = make_class ("Builder",{},{}),

map m in builder_methods

with m \ map r in rules

with r \ prec = <slamcode>false</slamcode>

\ postc = <slamcode>true</slamcode>,

abstract_to_builder (d) = map a in d ← attributes

with if a ← type in concrete_builders

then d \ type = builder

else d end

Figure 9.13: Builder pattern specification.
Part V

Conclusion
Chapter 10

Conclusions

This chapter summarizes the conclusions of this thesis.

10.1 The Design of a Formal Object Oriented Notation

One of the motivations of this work was the study and integration of standard and relatively new object oriented concepts in a formal notation. To achieve this goal we first designed SLAM-SL [62], a large object oriented notation designed to bridge the gap between formal methods and more widespread software engineering processes ([125] [65] [64]). Clay is a lightweight evolution of SLAM-SL, a desugared version where we focus on formal aspects superficially treated in the cited works.

The most relevant aspects of the Clay syntax and its informal semantics, presented in Chapter 2 are:

- Clay is a stateless object oriented formal notation, a class-based language with a nominal type system.

- Classes are defined as algebraic types in the form of case classes: complete and disjoint subclasses of the defining class.

- Classes can be extended by subclassing.

- Methods are specified with pre and postconditions, first order formulae involving self (the recipient), parameters and result (the resulting object). Atomic formulae are equalities (=) and class membership (\(\cdot\)).
• Equality has a contextual interpretation that refinement or subclassing cannot invalidate.

• Scandinavian semantics and dynamic binding for method overriding.

• Extremely permissive overloading scheme.

According to what we know so far, Clay is the first formal notation that incorporates all these features.

### 10.2 Non-structural Type System Definition

Abadi and Cardelli recognised that subtyping based on type names, in contrast to structural subtyping, is hard to define precisely. Chapter 3 contains the definition of a type system based on names for Clay. The type system is used to discard non well-formed Clay specifications and to guide the transformation processes that is used to provide the first-order logic semantics of Clay and that is later used to generate executable prototypes.

### 10.3 First-order Formal Semantics

One of the main contributions of our work is a first-order semantics for Clay presented in Chapter 4. An important characteristic of this contribution is that the semantics has been methodologically defined via an *interlingua-based translation* where the interlingua is a first-order language of the subsorted first-order logic. The contributions of this part of the thesis are:

• An encoding of several object oriented concepts and their integration: case classes, inheritance, permissive overloading, dynamic binding and static equality.

• Using the Prover9/Mace4 input language, a concrete syntax for first order logic, we open the door to mechanise the treatment of Clay and Clay specifications by using current proving technology. Particularly relevant is the fact that some theorems about the properties of Clay have been automatically proved.
10.4 Executable Prototype Generator

We have presented a compilation scheme of Clay specifications into logic programs. The basic conclusion here is that generation of executable prototypes from an object oriented formal notation that integrates the characteristics mentioned in Section 10.1 is shown feasible, which opens the possibility of devising a method based on formal specification and early validation of requirements using those prototypes.

The main contributions related to prototype generation are the following:

- Generation of code from implicit method specifications, specially in presence of recursive definitions, something which is seldom supported by other lightweight methods and tools.

- Our current implementation combines several techniques: Lloyd-Topor transforms of first-order formulae, constructive negation and iterative deepening.

10.5 Bridging the Gap Between Formal Methods and Software Engineering

One of the main motivations for this thesis was the effective integration of formal methods, and, in particular, object oriented specification languages in the software development process. Chapters 7, 8 and 9 were devoted to prove the applicability of our approach to formal methods to some common problems in software engineering.

- We contribute some evidence that an integration of XP practices and formal methods – also declarative technology – is possible. In particular, unit testing, refactoring, and, in a more detailed way, incremental development have been studied from the prism of formal methods.

- We contribute some syntactical characterisation of specifications that enables the synthesis of readable and efficient code in such a way that the use of IRPP and formal methods is closer, cheaper, and more effective.

- We contribute a formalization of design patterns by viewing them as operators between classes. The idea is not new and has circulated in the design patterns community for some time. However, to the best of our knowledge

\[\text{Footnote: For instance, Prof. John Vlissides mentioned it in a panel at POPL'00.}\]
we have not found a development of the technique. The precise definition of software design patterns is a prerequisite for allowing tool support in their implementation. Thus coherent specifications of patterns are essential not only to improve their understanding and to reason about their properties, but also to support and automate their use.

10.6 The Clay Compiler

We contribute an implementation of a tool that reflects all the work presented in Chapters 2, 3, 4 and 5. The Clay compiler has been essential in the development of the theoretical aspects of this thesis. The compiler functionality is:

- Parsing of modularised Clay specifications.
- Type checking and annotation of Clay specifications.
- Translation of Clay specifications to first-order theories in the concrete language of Prover9.
- Synthesis of executable prototypes from Clay specifications.

Implemented in Haskell, the Clay compiler has been constructed applying a methodological approach and state-of-the-art techniques of functional programming.
Chapter 11
Future Work

In this chapter book we give our view about the future work that can follow the research presented in this book.

Advanced constructions in Clay. We have already mentioned that Clay is a lightweight evolution of SLAM-SL designed to focus on the formal aspects superficially treated with SLAM-SL. Nevertheless, in this desugaring process some interesting construction of SLAM-SL were lost. Bringing back to Clay iterators, patterns, collection comprehension is one of our lines of potential future work.

Introduction of state. Clay is stateless while most of the current modelling languages used in by software engineers considers an implicit state. The plan is to modify the language to introduce the notion of state and from our point of view the most promising tools to support this extension are evolving algebras (as known as abstract state machines) and dynamic logic.

Induction schemes for Prover9/Mace4. The data model of Clay is inductive and to prove the consistency of the specifications the automatic theorem prover needs the induction schemes of every type. Anyway, since we are working with first-order logic and Prover9 do not allow any kind of schemes we plan to study of applying such schemes by exploring the intermediate results of the deduction process in order to incorporate new axioms that are instances of such schemes.

Correctness of the Prolog generator. The results of the synthesised prototypes are not valuable if they are not correct. The most obvious reason to get incorrect results from prototypes is that the synthesised program is not correct with
respect to the formal semantics of Clay. To avoid it we will have to work on a correctness result between the formal semantics of Chapter 4 and the translation of Chapter 5.

**Improving the efficiency of prototypes.** Comparison between the Prolog code obtained from our compiler and that crafted by hand still shows room for improvement. However, there exist more mature tools (see, for instance, ProB [83, 84]) which also generate logic programs from formal specifications, that show that certain extensions of logic programming (constraints, coroutining, etc.) can help in dramatically improving the efficiency of the resulting code in an automated way. However, applying such techniques will not be enough if no native and efficient Prolog data structures like integers, lists or constraints domains are the compilation target of the most used predefined Clay classes.
Part VI

Appendices
Appendix A

Clay Notation Reference

A.1 Lexical Issues
A.2 Namespaces
A.3 Grammar
A.4 Precedence and Associativity
A.5 Semantic Basis
A.6 Types and Overloading
A.7 Language Features
Bibliography

[1] 


[41] CSK Systems Corp. [http://www.csk.com/systems/]


[54] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.


[74] K.Lano and S.Goldsack. Integrated formal and object-oriented methods: The VDM++ approach. In \textit{In proceedings of Methods Integration Workshop}. Leeds Metropolitan University, (Supported by BCS FACS), March 1996.


[87] Barry Mazur. When is one thing equal to some other thing?, 2007.


[120] Vdm portal. [http://www.vdmportal.org/]

[121] VDM Tools. [http://www.vdmtools.jp/]


