Design and Implementation of an Object-Oriented Functional Language

Lee Valentin Braine

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of the

University of London.

Department of Computer Science

University College London

September 1998

Revised July 2000

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract

A novel approach to the integration of object-oriented programming (OOP) and functional programming (FP) is investigated. This is a well-researched area and we focus on several features that have, until now, proved resistant to integration.

The search for a language which combines both functional and object-oriented features has a long and distinguished history. The aim is to integrate the formal methods benefits of FP with the software engineering benefits of both paradigms. However, we know of no language which can claim to be both purely functional and purely object-oriented (and retains complete type safety).

In this thesis, we explain some important issues concerning the integration of OOP and FP. We show why achieving this goal is difficult by identifying key differences between the two paradigms and discussing the seemingly-incompatible design issues raised by these differences. Achieving the goal thus becomes a problem of solving apparently-conflicting language design requirements.

We present a design for a new language, CLOVER, which meets almost all of these requirements by the careful integration of a number of different design criteria. The language is purely functional and almost purely object-oriented, and is also completely type safe. The remaining object-oriented aspects are resolved by supplying a new interpretation of object identity through a new visual notation.

The main contribution of CLOVER is its breadth of scope - it incorporates all the key concepts of object orientation and is purely functional. In particular, it integrates subtyping, subsumption, inheritance, method overriding, method overloading and dynamic despatch from the object-oriented paradigm with higher-order functions, curried partial applications, referential transparency, laziness and complete type safety from the functional paradigm.

We demonstrate how the language can be implemented by targeting a simple functional language. We provide a formal presentation of the implementation as a set of translation rules together with rules for code generation. This translates an abstract form of our object-oriented functional language into an abstract form of a standard functional language.

Acknowledgements

I would like to thank Chris Clack, my supervisor, for several reasons. Firstly, for providing a thought-provoking atmosphere during several years of research. Also, for much time spent improving my papers and this thesis. And finally, for the many pints of Fosters and Guinness over the years!

I would also like to thank friends and colleagues in the Department of Computer Science at University College London for providing a pleasant environment. Their support, feedback and suggestions were very much appreciated. My thanks also go to the anonymous reviewers of my papers, whose comments contributed to this thesis.

This work was supported by a research studentship from the Engineering and Physical Sciences Research Council and a CASE award from Andersen Consulting. I am particularly grateful to Andersen Consulting for their encouragement and financial support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The object-oriented (OO) paradigm, together with an appropriate methodology, has successfully delivered many large projects. OO design (OOD) is used extensively in industry since it provides good control and componentisation characteristics for structuring the design of large applications. However, OO programming (OOP) has been rather disappointing: in particular, the expected level of code reuse has not been observed [McC97]. Furthermore, OOP languages have done little to reduce testing and debugging times: current OOP languages are not *completely* type safe and require extensive run-time testing and debugging.

By contrast, functional programming (FP) is not used extensively in industry, mainly due to perceived low performance, restricted programmer skill base, and poor support for large-scale applications programming (there is typically no methodology supporting analysis and design). However, FP languages can bring major benefits to program development. Complete type safety ensures that a high percentage of all errors are detected at compile time, thereby significantly reducing the time required for run-time debugging. Furthermore, functional languages have simple syntax and semantics and the key property of referential transparency ensures that encapsulation cannot be breached and programmers can work securely at an appropriate level of abstraction.

Our goal is to provide a specification language which is both purely functional and purely object-oriented — an object-oriented functional programming (OOFP) language. Since the definition of these two terms is open to interpretation (and indeed there is no universally agreed definition of OO), we choose to define the first in terms of features that are common to most lazy functional languages and to define the second in terms

of a minimal subset that is common to most object-oriented languages and, we believe, captures the essence of object-oriented programming. We define the former as "referentially transparent with no side-effects and completely type safe, with lazy evaluation, parametric polymorphism, higher-order functions and partial applications" and the latter as "purely object-oriented (where everything is an object), using a class hierarchy with inheritance and pure encapsulation, subsumption through subtyping, method overloading, method overriding and dynamic despatch".

1.1 Goals of the Research

This thesis aims to investigate the following hypothesis:

The functional and object-oriented paradigms can be integrated, whilst retaining:

- *higher-order functions, curried partial applications, referential transparency, laziness and complete type safety from the functional paradigm;*
- *subtyping, subsumption, inheritance, method overriding, method overloading and dynamic despatch from the object-oriented paradigm.*

We will show that this hypothesis has not been demonstrated by previous research and also why achieving this goal is difficult by identifying the key research problems. We do not aim to *prove* complete type safety (in that the full construction of a type system including associated proofs is outside the scope of this thesis), but will provide a sufficiently detailed design, typing rules and type checker to give a high degree of confidence that this is achievable.

1.2 Contributions of the Work

This work contributes to the understanding of the design space of object-oriented and functional programming languages. Specifically, the contributions of this thesis are:

- a new design for completely type-safe dynamic method despatch and overloading;
- a new object-oriented semantics for partially-applied, higher-order methods;
- a new design for full overloading of methods in the presence of curried partial applications and dynamic despatch;
- a new visual notation and semantics for object state, object identity and object-oriented lazy evaluation.

1.3 Overview of the Thesis

The thesis is structured as follows: after providing a background of relevant terms and related work in Chapter 2, we establish the problem by discussing the difficult design issues in Chapter 3; we then present a new language CLOVER, its design features, syntax, type system and abstract expression semantics. In Chapter 4, we present an implementation of CLOVER as a translation from OOFP to FP. We then discuss the inappropriateness of the traditional notion of object identity for OOFP in Chapter 5 and propose an alternative notion, together with a supporting new visual programming notation. In Chapter 6, we assess this work, summarise the project status, suggest directions for further work, and conclude. Finally, in the appendices, we present a type checking algorithm and an example translation from CLOVER concrete syntax to a target functional language.

1.4 Publications

This thesis is based on the following publications (repeated in the bibliography):

[BC96] L. Braine and C. Clack. Introducing CLOVER: an Object-Oriented Functional Language. In W. Kluge, editor, *Implementation of Functional Languages, 8th International Workshop (IFL'96), Selected Papers*, Lecture Notes in Computer Science 1268, pages 1–20, Springer-Verlag, September 1996.

[BC97] L. Braine and C. Clack. An Object-Oriented Functional Approach to Information Systems Engineering. In *Proceedings of the CAiSE'97 4th Doctoral Consortium on Advanced Information Systems Engineering*, 12 pages, June 1997.

[BC97a] L. Braine and C. Clack. Object-Flow. In *Proceedings of the 13th IEEE Symposium on Visual Languages (VL'97)*, pages 422–423, September 1997.

[BC97b] L. Braine and C. Clack. The CLOVER Rewrite Rules: A Translation from OOFP to FP. In *Draft Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL'97)*, pages 467–488, September 1997.

The following paper also provides a case study of the application of several of the OOFP techniques described in this thesis within a large commercial project:

[BC98] L. Braine and C. Clack. Simulating an Object-Oriented Financial System in a Functional Language. In *Draft Proceedings of the 10th International Workshop on Implementation of Functional Languages (IFL'98)*, pages 487–496, September 1998.

Chapter 2

Background

In this chapter, we provide an overview of relevant object-oriented programming and functional programming terms. We also survey related work by focussing on relevant languages in the areas of object-oriented functional programming and visual programming.

2.1 Overview of Terms

This section briefly summarises relevant OOP and FP terms as often used in the research communities.

2.1.1 Overview of Object-Oriented Programming Terms

Basic Concepts:

A *class* comprises both *attributes* (private data items) and *methods* (interface functions). Each class has a special *constructor* function which is used to define (*instantiate*) an instance of that class with specific values for its attributes. An *object* is the instantiation of a class. An object may be manipulated by sending it a *message*, which is the name of one of the object's methods together with the actual parameters for that method.

Further Terms:

- *Binary methods* take a parameter (in addition to the distinguished object) which has the same type as the distinguished object.

- *Delegation* is the act of one object forwarding an operation to another object, to be performed on behalf of the first object [Boo94].

- A *distinguished object* (DO) is used at run-time to resolve method overriding and determine which implementation should be used for a given method application (see overloading and overriding).

- *Dynamic binding* associates a name with a value at run-time (and often a type, if this is not known statically — see dynamic typing).

- *Dynamic despatch* is a run-time feature which invokes the implementation associated with a method name according to the actual (run-time) type of the distinguished object.

- *Dynamic typing* associates a name with a type at run-time.

- *Inheritance* is a relationship among classes, wherein a class shares the attributes or methods defined in one (single inheritance) or more (multiple inheritance) other classes (see also [Boo94]).

- *Multi-methods* is a run-time feature which allows dynamic despatch to be based on more than one parameter.

- *Overloading* provides multiple implementations for a method, each with a distinct type.

- *Overriding* provides a replacement implementation for an inherited method or attribute.

- `self` is an identifier, defined in every method, that is bound dynamically to the distinguished object.

- *Subsumption* allows a function of type $\sigma_1 \rightarrow \sigma_2$ to be provided with an argument of type σ_3 iff σ_3 is a subtype of σ_1.

- *Subtyping* has been defined in many ways (see, for example, [CW85]). In general, it can be viewed as a type system with a pairwise relation on types which provides a partial ordering.

2.1.2 Overview of Functional Programming Terms

Basic Concepts:

A functional program contains a number of *function definitions*, each comprising an *expression definition* and a *type definition*. If the language supports *type inference*, the type definition may be optional. In an interpreted system, the user can choose to evaluate an arbitrary expression which may include applications of any of these functions; in a compiled system, one of these functions (normally called `main`) is chosen by the compiler as the primary expression to be evaluated. In both cases, the remaining functions provide the environment in which the primary expression is evaluated.

Further Terms:

- *Referential transparency* is the property ascribed to an expression if its evaluation always returns the same value, regardless of when it is evaluated, or how many times it is evaluated, and regardless of what may have been evaluated in the past or may be evaluated in the future.

- *Polymorphism* allows type expressions to include variable names which can be dynamically bound to any concrete type. A polymorphic function is one which contains a polymorphic type variable in its type definition.

- *Higher-order functions* either take a function as (or as part of) one of their arguments or return a function as (or as part of) their result. A value that is passed as a parameter to a function is said to be a *higher-order value* if it is a function.

- *Currying* permits greater flexibility in the passing of arguments to functions. A function of more than one argument may be defined as either: (i) a single mapping from a tuple of all the input arguments to the result value, or as (ii) a sequence of mappings between the separate input arguments and the result value. The latter style is known as Currying and effectively defines a number of anonymous intermediate functions in addition to the named function. A function that is defined using Currying is also known as a *curried function.*

- *Partial applications* permit greater flexibility in the application of functions to arguments. A function that is defined in a curried style may be partially applied:

that is, it may be applied just to its first argument, or its first two arguments, and so on including application to all of its arguments (in which case it is said to be fully applied). When a curried function is applied to its first argument, the result is another (anonymous) function which is waiting to be applied to the remaining arguments. The result can be bound to a name, thereby supporting function specialisations.

- *Laziness* is an evaluation strategy adopted by many modern functional languages. There are two general mechanisms for evaluating functional languages: (i) *applicative-order evaluation* (where arguments to functions are evaluated before the function is executed — this corresponds to call-by-value parameter passing), and (ii) *normal-order evaluation* (where arguments to functions are only evaluated if and when they are needed by the function — this corresponds to call-by-name parameter passing). If the function body contains multiple copies of an argument name, normal-order evaluation runs the risk of multiple evaluation of the same term (which is inefficient); thus, an implementation technique is used which ensures that arguments are evaluated "at most once". This implementation technique is called lazy evaluation — in practice, it also extends to include the lazy evaluation of data constructors, so that quasi-infinite data structures can be supported.

2.2 Related Work

Related work encompasses OOFP languages and the formalisation of OOP (such as a λ-calculus of objects [FHM94] and an object calculus [AC96] — the reader is referred to [CW85] for foundational work in this area). In this section, we present an overview of the relevant work in these areas.

2.2.1 Related Work on Object-Oriented Functional Programming

Since at least the early 1980s there has been considerable interest in the formalisation of OOP, most notable being attempts to integrate OOP and FP.

1980–1989

Early work such as Flavors [Can82, Moo86] and CommonLOOPS [BKK+86] involved the extension of LISP [BB64] with object-oriented features. This work culminated in CLOS [BDG+88], a set of tools for developing object-oriented programs in Common LISP [Ste84]. Significant claims have been made [BGW91] that CLOS combines both OOP and FP as it supports the OOP features of classes, inheritance and method despatch, together with the FP features of Common LISP. However, because it is based on LISP (like Flavors and CommonLOOPS), it is not referentially transparent and therefore fails to satisfy one of the criteria stated in the hypothesis.

1990–1992

In the early 1990s, interest in OOP/FP integration increased, with several newly developed languages. However, five of these languages (Leda [Bud95], Quest [CL91], Rapide [MMM91], UFO [Sar93] and O^2FDL [MCB90]) are not referentially transparent, FOOPS [Soc93] has no higher-order programming facilities, LIFE [AP93] only supports a simulation of FP, and G [HL91] and HOP [DV96] do not support full OOP. Kea [MHH91], Leda, Rapide, UFO, O^2FDL, G and HOP are briefly outlined below:

- Kea is a higher-order, polymorphic, lazy functional language supporting multi-methods and a type inference system. Unlike Smalltalk [GR83], Kea does not enforce certain aspects of OO encapsulation. In particular, Kea functions do not have to be associated with classes according to a distinguished object. Furthermore, Kea's notion of polymorphism only admits the single type variable `Any`, and [MHH91] states that Kea "is currently being extended to include higher-order and (implicitly) polymorphic functions", which implies that it does not have these features.

- Leda is an experimental language that provides an environment for multiparadigm programming. It claims to integrate imperative, functional, object-oriented and logic programming through one common language. However, this claim is too strong (for example, Leda makes no distinction between functions and procedures — [Bud95] advises that "those wishing to remain pure in the functional programming paradigm must simply employ discipline to avoid those language features

that are at odds with this technique"). Such a discipline requires the avoidance of many OOP features and Leda should, therefore, be viewed as providing *either* FP or OOP, but not integrating both simultaneously.

- Rapide extends Standard ML (SML) [MTH90] with subtyping and inheritance. Objects are modelled as structures, and SML is extended so that structures may be passed to and from functions. Unfortunately, Rapide retains SML's lack of referential transparency and, indeed, relies on it.

- United Functions and Objects (UFO) is an implicitly-parallel language that "attempts to bring [the functional and object-oriented] worlds together in a harmonious fashion" [Sar93]. A functional subset of UFO provides OOP features such as classes and inheritance, but referential transparency is lost once any of the language's *stateful* features (stateful classes, instance variables, etc.) are used. It is interesting to note that even the functional subset is not lazy, on the grounds that it "conflicts with dynamic binding".

- O^2FDL is an interactive database programming language that combines inheritance and encapsulation from OOP with an equational programming style and strong typing from FP. Although [MCB90] provides a denotational semantics based on an extended λ-calculus, referential transparency is preserved only within a given database state, not across the database lifetime.

- G is a language framework that aims to integrate algebraic, functional and object-oriented programming. The language design is closely related to Rapide [MMM91], but requires the programmer to define explicit conversion functions between types. G therefore lacks a key OOP feature of implicit subtyping.

- HOP is a functional language with object-oriented features incorporating dynamic binding and subtyping; it is also referentially transparent and lazy. Based on an extension of the λ-calculus called *label-selective λ-calculus* (also known as the λN-calculus) [AG93], HOP is an experimental language for testing the provision of OO features within FP. However, there is as yet no clear notion of "object" and

no explanation of how dynamic despatch, inheritance, subsumption, overloading and overriding would be implemented.

1993–1995

In this period there were a number of notable attempts at integration, plus extensions of previous systems. OBJ [GWM+93] is a functional language that supports multiple inheritance, exception handling and overloading but has no higher-order programming facilities. ST&T [DK94] is an extension of Smalltalk's type system bringing it closer to ML, though the result is first-order, strict, and still not referentially transparent. Uflow [SKA94] is an extension to UFO using a data-flow model for visualising execution, but is still not referentially transparent. Oz [MMR95] is a multiparadigm language which aims to encompass logical, functional and object-oriented styles; however, its use of "mutable binding of a name to a variable" results in referential transparency being maintained only "inside the objects", not across objects nor the language in general. Finally, Caml Special Light [Ler95] (later re-named Objective Caml) laid the foundations for Objective ML (see next subsection), but unfortunately these foundations are not referentially transparent.

1996–1998

The past three years have witnessed an intensifying of interest in the field with several new languages being established, including our language CLOVER. Objective ML [RV97] is implemented on top of Caml Special Light and is an extension of ML with objects, top-level classes, multiple inheritance, methods returning `self`, binary methods and parametric classes. Object ML [RR96] extends ML with objects, subtyping and heterogeneous collections. CLAIRE [CL96] is a high-level functional and object-oriented language with advanced rule processing capabilities. Object-Gofer [AS97] extends Gofer [Jon94] with classes, subtyping, inheritance and late binding, although it omits subsumption and so requires explicit type coercions to achieve subtyping. OOId [CSK+97] extends Id [NA92] with classes, inheritance and dynamic binding, but loses referential transparency with its addition of stateful objects. FOC [QM97] models a combination of concepts from OOP and FP, although it omits key features (such as overloading), is not purely object-oriented (global functions are permitted), requires explicit

casting operators and, importantly, does not consider the effects of assignment and state. Bla [Oor96] claims to unite functional and object-oriented programming through *first class environments* [GJL87]. However, none of these languages except CLOVER is referentially transparent.

$ML_{\leq}$ [BM97] is a decidable type system for higher-order object-oriented languages. It takes a polymorphic multi-methods approach (rather than the "standard" view of objects as extensible records with single-despatch methods) and can be applied directly only to multi-methods languages such as CLOS.

2.2.2 Related Work on Visual Programming

The functional paradigm is often chosen as the underlying computational model for VP because its simple semantics can be realised elegantly in standard data-flow notation. Examples of visual functional languages include:

- HI-VISUAL [MYH+84] is a visual data-flow programming language with an interactive iconic programming environment;
- viz [Hol90] is an active data-flow visual language based on the λ-calculus;
- VPL [LBF+91] is a demand-driven higher-order data-flow visual programming language for interactive image processing;
- Cantana [RW91] is a data-driven data-flow visual language component of Khoros (a general-purpose programming language);
- VISAVIS [PVM95] is a purely functional higher-order visual programming language based on the Formal FP (FFP) model [Bac78].

None of these languages, however, offer the object-oriented features we require.

The object-oriented paradigm doesn't lend itself as naturally to VP, partly because of the extra complexity that object-oriented features introduce and partly because of the tendency of object diagrams (with nodes containing mutable state) to result in complicated designs. There is a large number of visual object-oriented languages ranging from CASE tools to domain-specific languages; a well-known example is the general-purpose language Prograph [SPL89]. However, as they do not claim to be functional (i.e. they are not referentially transparent), they are not discussed in this section.

2.2.3 Summary of Related Work

It seems that, despite considerable attention from the research community, it has been impossible to combine object-oriented features such as inheritance, subsumption and dynamic method despatch with functional features such as referential transparency, higher-order functions, Currying, partial applications and lazy evaluation, into a single, completely type-safe, language. The closest attempts so far are Kea, Rapide, HOP, LIFE, Objective ML and Object ML. Furthermore, there are no visual languages which combine these functional and object-oriented features.

Chapter 3

The Design of CLOVER: An OOFP Language

In this chapter, we present the new object-oriented functional language CLOVER, its design features and syntax. In addition, we present CLOVER's type system, abstract expression semantics and also address referential transparency.

3.1 Design Issues

The most stringent criterion that we can devise for an object-oriented functional language is that it must be purely functional — that is, it must be referentially transparent with no side effects — for without this property most of the formal-methods advantages of the functional paradigm are lost (e.g. most static analysis and program manipulation techniques for FP languages assume referential transparency). As discussed in Chapter 2, attempts to create OOFP languages typically abandon referential transparency, despite its fundamental importance being emphasised repeatedly in the FP literature (e.g. "this property [referential transparency] is the hallmark of a functional language, and that under no circumstances should it be abandoned" [Sto85]).

Thus, our view is one of extending FP towards OOP rather than the other way around. Note that this requires us immediately to discard imperative notions of multiple assignment (see Chapter 5). We make two more design decisions at the outset:

1. We discard any notion of multiple inheritance because it complicates the semantics of OOP considerably (e.g. resolving attribute and method naming conflicts and upwards type coercion). If necessary, similar behaviour can be achieved us-

ing explicit object delegation.

2. We choose dynamic despatch based on a single distinguished object to avoid the complexities of the multi-methods approach and to increase encapsulation.

In the remainder of this section we present five design issues which illustrate why the integration of OOP and FP is such a difficult task. Note that these issues are purely examples and are not a comprehensive analysis.

3.1.1 Type Safety versus Dynamic Despatch

A key feature of OO languages is dynamic typing (resolving the type of an object at run-time). In particular, subtyping permits conditional statements to return different subtypes of the declared return type. Thus, in general, it is not possible to resolve method despatch statically for dynamically-typed objects.

With dynamic method despatch, a run-time check is made on the actual type of the object receiving a message; the ambiguities arising from inheritance and method overriding are then resolved and the appropriate code is executed.

Many OO languages assume and accept that this implies type errors may occur at run-time. Having realised that this is undesirable, some OO language designers have created what they claim to be "type safe" OO languages: Eiffel [Mey91], for example, makes this claim. However, Eiffel actually provides an *assignment attempt operator* which handles run-time type errors in a controlled manner by assigning a `void` value; it is assumed that the programmer will always check for the possibility of a `void` value and take appropriate action. This is not what functional programmers think of as "type safe"! The FP world requires *complete* type safety, where the type system guarantees that it is impossible for a type error to occur at run-time.

Dynamic types thus appear to compromise type safety, though recent work using run-time type checking [AWL94, AF95, AM90] partially extenuates the problem; however, a problem remains that default actions may be specified for situations where the requested method is not defined for the run-time type of the object.

3.1.2 Type Safety versus Overloading and Partial Applications

Overloading is a common feature of OO languages, allowing different definitions for the same method name. These overloadings can be distinguished by the types of the message arguments and return value: each overloading must have a unique type signature. Overloading is certainly a desirable feature that we would wish to incorporate since it allows, for example, multiple ways to set a date:

```
date (7,"August",1996)
date (7,8,1996)
```

In the above example, the two overloaded versions of `date` take the same number of arguments. However, it is also important to support overloading with different numbers of arguments, for example:

```
time (12,0,0)
time ("noon")
```

We wish to support *full overloading*: that is, overloaded methods able to vary both in the type and number of arguments declared.

Unfortunately, *it would appear to be impossible to combine dynamic despatch with curried partial applications and full overloading*. With full overloading, the number of arguments in different overloadings may vary. With partial applications, a method may be applied to only some of its arguments. If the partial application uses the curried style (rather than, for example, a tuple with dummy values for the missing arguments), then there are ambiguities which are impossible to resolve at run-time. For example, given the following overloaded definitions (using the 24-hour and the 12-hour clock):

```
time (a:int) (b:int) (c:int)
time (a:int) (b:int) (c:int) (d:string)
```

then is the application `time 6 0 0` a full application of the first overloading (meaning 6 am) or a partial application of the second overloading (which could eventually be 6 am or 6 pm)? If this cannot be resolved at run-time, then we must require all types to be known at compile-time so that overloaded functions can be resolved statically and complete type safety can be guaranteed. However, we have already established that dynamic typing implies we cannot know all actual types at compile-time.

3.1.3 Curried and Partially-Applied Methods

Currying, higher-order functions and partial applications are key features of the functional programming style, yet are absent from OOP. This is not unreasonable, since OO programmers normally perceive messages to be indivisible and it is not clear what a partially applied message would mean. For example, if the method `f` takes distinguished object `o` and normally takes three arguments `a`, `b` and `c`, then what does the message `f a b` denote? Can it be given a name? What does it mean operationally? Can it be sent as it is to the object `o` or must it be delayed until the final argument is ready? If it is sent to the object, what does the object do with it? Must it store it and wait for the final argument to arrive? The denotational and operational semantics of partial applications have not been fully addressed in the OO world.

Furthermore, as we have already seen, it is difficult to reconcile curried partial applications with dynamic despatch and full overloading.

3.1.4 Subtyping

Subtyping is central to OOP but absent from current "production" FP languages. Haskell [PHA+97] has at various times been the subject of claims that its type classes mechanism [WB89] facilitates OO programming [Ber92]; this has promulgated the mistaken assumption that Haskell's type classes provide subtyping. Unfortunately type classes do not provide the subtype relationship that we require; rather, they support a structured form of overloading.

If we are to support dynamic despatch, then the run-time method despatcher must accept an argument of many different types (the method's distinguished object); this requires either subtyping or flattening the entire type system into what is essentially a monotyped language. Similarly, subsumption requires subtyping to be applied to method arguments.

FP languages rely on advanced polymorphic type inference to ensure type correctness. The type systems of most FP languages are based on the Hindley/Milner algorithm [DM82] which does not admit subtypes. A notable exception is Mitchell's extension to the SML type system to admit inclusion polymorphism through subtyping [MMM91]. Subtyping in HOP [DV96] claims to be based largely on Mitchell's work (using recursive type constraints).

Whereas polymorphic type inference used to be considered undecidable for inclusion types, recent work [BM96] has demonstrated that decidable systems can be implemented.

3.1.5 Lazy Evaluation versus Discrete Messages and State Update

We wish to retain the powerful FP feature of lazy evaluation, yet this does not seem to have a natural meaning within the message-passing view of OOP. For example:

1. Multiple assignment semantics require strict state update;
2. State update is driven by the arrival of a message (data-driven);
3. Messages are discrete, finite and pre-evaluated;
4. Sending a message is an atomic action.

The above views seem to preclude the incorporation of any notion of lazy evaluation into OOP. However, we will show in the next section that it is in fact possible to employ lazy evaluation in a language that supports object-oriented features.

3.2 CLOVER

We now present the design for a higher-order, lazy, object-oriented, completely type-safe functional language; we call this language CLOVER. CLOVER provides:

1. a new design for completely type safe dynamic method despatch and overloading;[1]
2. a new object-oriented semantics for partially applied messages and higher-order functions;
3. a new design for full overloading of methods in the presence of curried partial applications and dynamic despatch;
4. a new programming notation and semantics for object state, object identity and object-oriented lazy evaluation.

[1]Note that Eiffel claims type-safe dynamic despatch but at the cost of losing overloading, whereas Haskell has overloading but not dynamic despatch.

CLOVER is intended to be used for application development, not low-level systems programming. We support programming at the specification level, much as functional languages can be used to write executable specifications [Tur85a].

CLOVER supports the traditional OOP features of a class hierarchy, subtyping, subsumption, inheritance, method overloading, method overriding and dynamic despatch. It also incorporates the FP features of referential transparency, single-assignment attributes, polymorphism, curried partial applications, higher-order functions and lazy evaluation. Methods are defined as expressions — they are pure functions with no side effects.

The language is completely type safe, there are no pointers and memory allocation is automatic; thus, CLOVER is a secure language which could be used, for example, to produce totally secure applets for the World-Wide Web. However, a secure CLOVER run-time system as a browser plug-in is left for future work!

In our prototype, a lazy functional programming language is used as an intermediate language: CLOVER code is first type-checked using a bespoke type-checker (see Appendix A) and then translated into a standard functional language. This allows the use of a standard compiler for final code generation.

3.2.1 Design Features

The key to CLOVER's successful support of both OOP and FP is in the careful integration of a number of different design criteria. Since there are so many design parameters (subtyping, subsumption, inheritance, overloading, overriding, genericity, partial applications, Currying, laziness, etc.), the design space is extremely large and our goal has proven to be remarkably elusive. However, as is so often the case, the solution appears quite natural in retrospect.

The key design criteria are all related to type safety, including bounded universal quantification, monotonic inheritance, and shallow subtyping. Furthermore, in order to deal with curried partial applications, we enforce an unusual ordering constraint on the implementation of message application. Finally, we support an object-oriented view of lazy evaluation through the use of a new visual notation (see Chapter 5); thus, CLOVER is a visual object-oriented functional language.

Bounded universal quantification:

It is clear that dynamic typing is incompatible with knowledge of actual types at compile time, yet we require a language which has dynamic typing and is also completely type safe.

The first step to solving this apparent conflict is to ensure that upper bounds on types are always known statically. This allows dynamic typing (in that the actual type of an expression can be any subtype of the known upper bound), whilst ensuring that all type errors (in terms of the upper bounds) can be detected at compile time. We currently require the programmer to give explicit upper bound types in all method type signatures and for all method arguments, though in future we hope to implement a subtype inference system. Inclusion polymorphism is thus implemented as bounded universal quantification [CW85].

Monotonic inheritance:

For completely safe method despatch, statically resolvable upper bounds are only part of the solution. When a message is passed to an object, the required method must also actually be defined for that object. The static knowledge of the upperbound type of the object must therefore be coupled with the restriction that inheritance be monotonic; that is, if a method or attribute exists for a given class then it will also exist (with identical type signature) for all of its subclasses.

Shallow subtyping:

In order to achieve complete type safety, it is essential that full method overloading can be resolved statically. Since only upper bound types are available at compile-time, we must therefore restrict CLOVER to shallow subtyping — that is, an inherited or reimplemented method must have the same type as its ancestor. We provide full method overloading (with different types and numbers of arguments) but insist that all overloadings are declared in the class where the method name is first defined. Thus, if an overloaded method application is valid for a given type then it will be valid for all subtypes.

Implementation of message application:

A message application to an object is often written as `o.f(a,b,c)`. The traditional way to implement this is as the function call `f(o,a,b,c)`.

For CLOVER, we wish to support curried partial applications and so it would seem that the above application could naturally be implemented as `f o a b c`. However, this causes problems for partial applications of messages. As previously discussed, there is a problem with the semantics of partial applications in an object-oriented context. Considering the above implementation technique, what would `f o a` mean, both denotationally and operationally?

We define a *partially-applied message* as a method that has not yet been applied to all its arguments and that has not yet been applied to its distinguished object. We allow a partial application to be named, to be passed as an argument to a method, and to be returned as a result from a method. However, only a fully-applied message can be sent to a distinguished object.

To implement these semantics precisely, we adopt the unusual procedure of placing the distinguished object as the *last* in the sequence of curried arguments: `f a b c o`.

Laziness:

The key to the incorporation of laziness in CLOVER is our new concept of object identity, as explained in Chapter 5.

3.2.2 Language Overview

A CLOVER program consists of three components:

1. an invocation (an expression);
2. a class hierarchy (a tree with at least one class);
3. for each class, an unordered set of attribute declarations and method definitions.

The class hierarchy is single-rooted with single monotonic inheritance for the definition of new classes as extensions of existing classes; thus, there is no sharing in the class hierarchy, and inherited attributes and methods cannot be discarded. As in the Smalltalk tradition, everything is either an object or a message — thus, the class hierarchy contains class definitions for even the most primitive types such as integer and character. A fully-applied message is a method applied to all of its arguments except the distinguished object; the distinguished object is always the last argument. Messages may be specialised through partial application. Arguments to methods and results from

methods may be objects or messages (including partial messages). Thus, CLOVER is higher-order, treating messages as first-class citizens.

The class hierarchy represents the subtype structure (see Section 3.3.1). Subclasses may inherit methods and attributes through shallow subtyping only — that is, an inherited or reimplemented method must have the same type as its ancestor. Overloaded method declaration is allowed, but only in the greatest superclass where the method is first defined; thereafter, the separate overloaded instances may be inherited and reimplemented through shallow subtyping as described above. Thus, CLOVER supports both overloading and overriding.

Each class is a subtype of its parent and subsumption allows a formal method parameter of type σ to be bound to an actual parameter of type τ as long as τ is a subtype of σ (and using the contravariant rule to establish subtypes of higher-order arguments). Method overloading is resolved at compile-time, whereas method overriding is resolved at run-time by dynamic despatch on the type of the distinguished object (if the distinguished object's class does not define or override the method, the despatcher (conceptually) searches up the inheritance hierarchy to find the least superclass which has a definition for the method — note that this cannot fail at run-time). CLOVER provides completely type-safe subsumption and dynamic despatch using bounded universal quantification of type names.

3.2.3 Language Primitives

CLOVER's strict adherence to treating everything uniformly as either an object or a message extends to all language primitives. Thus, primitive values of function type (such as + and =) are wrapped within appropriate CLOVER methods (such as `Add` and `AreEqual`). Additionally, primitive values of non-function type (such as `True`, `'x'` and `54`) are wrapped within appropriate CLOVER classes (such as `Boolean`, `Character` and `Number`). These classes can then be subclassed and extended (for example, to create `BoundedNumber` or `ComplexNumber`) and their methods, such as `Add`, can be reused or overridden.

3.2.4 Abstract Expression Syntax

The following abstract syntax for an expression is based on the typed λ-calculus (extended with *let* and *case* constructs) and with objects as constants:

$value$	::	$C^{\sigma_1 \to \cdots \to \sigma_n \to \sigma}\ x_1^{\sigma_1} \ldots x_n^{\sigma_n}$	$n \geq 0$ $(object\ constructor)$
	|	$\lambda x^{\sigma_x} . e^{\sigma_e}$	$(curried\ method\ definition)$
	|	Φ_i^{σ}	$(built-in\ method)$
	|	$literal^{\sigma}$	$(immediate\ data)$
$expr$	::	$value$	$(value)$
	|	$e^{\sigma_1 \to \sigma_2}\ x^{\sigma_x}$	$\sigma_x \preceq \sigma_1$ $(curried\ application,$ $\preceq\ is\ defined\ in\ Section\ 3.3.3)$
	|	x^{σ}	$(name)$
	|	$let\ (x_1 = e_1^{\sigma_1}) \ldots\ (x_n = e_n^{\sigma_n})\ in\ e^{\sigma_e}$	$n > 0$ $(local\ definition)$
	|	$case\ e^{\sigma}\ of\ (C_j^{\sigma}\ x_{j1}^{\sigma_{j1}} \ldots x_{jn_j}^{\sigma_{jn_j}} \to e_j^{\sigma_j})$	$j = 1 \ldots m,\ m > 0,\ n_j > 0\ \forall j$

In the above syntax, σ denotes a type. Methods, bound variables and object attributes are referenced through identifiers ("x") and key primitive methods are built-in to facilitate the implementation. λ-abstractions are only used at the top level of a method binding.

Support is provided for bindings with local scope using *let*. However, these bindings may only be local *constant applicative forms* [Pey87]; that is, they may not be parameterised λ-abstractions but they may be any other expression which returns either an object or a (partial) message. The restriction which outlaws the λ-abstraction in *let* bindings prevents the undesirable creation of new methods as local definitions, since all methods should formally be specified as part of a class interface.

The typechecker's (static) overloading resolution and the implementation of (dynamic) method despatch are illustrated in the abstract expression semantics (see later).

3.3 Type System

In this section we briefly sketch the design of the CLOVER type system. Our prototype currently supports simple type checking rather than full type inference (see Appendix A). For an introduction and review of types in object-oriented programming, see [FM95].

3.3.1 Type Syntax

A type is either an object, a message (which has function type), a bracketed message or a primitive (which is required within class definitions for primitive types). The explicit bracketing is necessary to denote a function type being returned from a method — this facilitates identification of the distinguished object (the last argument to a method). Each class has a distinct type constructor name κ (we define a one-to-one correspondence between the class name and the type constructor name). Thus, the syntax for types in CLOVER is given by:

$$\begin{aligned} \tau \;::\; & \kappa \\ & \mid \; \tau_1 \rightarrow \tau_2 \mid (\tau_1 \rightarrow \tau_2) \\ & \mid \; \texttt{'bool'} \mid \texttt{'char'} \mid \texttt{'num'} \end{aligned}$$

3.3.2 Overview of Type Semantics

We define an object's type as the set of the names and types of all its attributes and methods:

$$\begin{aligned} \mathcal{T}[\![\kappa]\!] &= \{x_i\} \cup \{m_i\}, x_i \in \mathit{Attributes}(\kappa), m_i \in \mathit{Methods}(\kappa) \\ \mathcal{T}[\![\sigma_1 \rightarrow \sigma_2]\!] &= \mathcal{T}[\![\sigma_1]\!] \rightarrow \mathcal{T}[\![\sigma_2]\!] \\ \mathcal{T}[\![(\sigma_1 \rightarrow \sigma_2)]\!] &= (\mathcal{T}[\![\sigma_1]\!] \rightarrow \mathcal{T}[\![\sigma_2]\!]) \end{aligned}$$

3.3.3 Subtyping

Although we would rather have the intermediate FP language compiler do the type checking, our preferred languages (Miranda [Tur85] and Haskell [PHA+97]) do not yet provide subtyping and are therefore unable to check inclusion polymorphism. We therefore currently implement a simple subtype matching algorithm (see Appendix A).

We take a traditional set-theoretic view of the class system [Car88]: class types are sets of attributes and methods, with subclassing equivalent to subtyping. Subtypes are ordered inversely by set-inclusion over the above semantic domain.

We define the subtype relation operator $\preceq$ as follows:

$$
\begin{aligned}
\kappa_1 &\preceq \kappa_2 \text{ iff } \mathcal{T}[[\kappa_2]] \subseteq \mathcal{T}[[\kappa_1]] \\
\sigma_1 \to \sigma_2 &\preceq \tau_1 \to \tau_2 \text{ iff } (\tau_1 \preceq \sigma_1) \wedge (\sigma_2 \preceq \tau_2) \\
(\sigma_1 \to \sigma_2) &\preceq (\tau_1 \to \tau_2) \text{ iff } (\tau_1 \preceq \sigma_1) \wedge (\sigma_2 \preceq \tau_2) \\
\kappa_i &\not\preceq \tau_1 \to \tau_2 \\
\kappa_i &\not\preceq (\tau_1 \to \tau_2) \\
\sigma_1 \to \sigma_2 &\not\preceq \kappa_i \\
(\sigma_1 \to \sigma_2) &\not\preceq \kappa_i \\
(\sigma_1 \to \sigma_2) &\not\preceq \tau_1 \to \tau_2 \\
\sigma_1 \to \sigma_2 &\not\preceq (\tau_1 \to \tau_2)
\end{aligned}
$$

3.3.4 Overloading

A method may be defined at many different types, where the types are completely unrelated. However, we require each overloaded definition to have a unique type signature that is statically-resolvable from the other overloaded definitions. Thus, given two overloaded method definitions of type $\sigma_1 \to \sigma_{dist} \to \sigma_2$ and $\tau_1 \to \sigma_{dist} \to \tau_2$ (where σ_{dist} is the type of the distinguished object), we require that either $(\sigma_1 \not\preceq \tau_1) \wedge (\tau_1 \not\preceq \sigma_1)$ or $(\sigma_2 \not\preceq \tau_2) \wedge (\tau_2 \not\preceq \sigma_2)$. This generalises to multiple arguments and overloadings with different numbers of arguments. Note that this style of overloading permits some covariant specialisation.

3.3.5 Polymorphism

Polymorphism is supported in CLOVER through bounded universal quantification [CW85] which provides both inclusion and parametric polymorphism without the need for type variables (which are replaced by subtype constraints). Our prototype does not yet support recursive types, though they are an essential element of CLOVER.

Our type matcher (see Appendix A) checks declared return types against declared argument types, for every application, to ensure that the subset relationship holds. We do not attempt type inference; type inference for inclusion polymorphism has long been considered problematic and, though we are encouraged by recent work [AW93, BM96], dynamic despatch causes problems for CLOVER type inference.

3.3.6 Typing Rules

In this section, we provide a set of typing rules for the CLOVER type system. We use $x : \sigma$ to denote an assumption (an association of type σ with variable x), $\Gamma \vdash x : \sigma$ to mean "from the set of assumptions Γ we can deduce that x has type σ", $\Gamma \cup \{x : \sigma\}$ to denote the set of assumptions formed by adding $x : \sigma$ to Γ (which does not already contain a typing for x), and $\frac{X}{Y}$ to mean "from X we can infer Y". Note that the *Let* rule does not handle recursive functions — this is left for further work.

$$\frac{}{\Gamma \cup \{x : \sigma\} \vdash x : \sigma} \quad [Var]$$

$$\frac{}{\Gamma \cup \{literal : \sigma\} \vdash literal : \sigma} \quad [Lit]$$

$$\frac{}{\Gamma \cup \{\Phi : \sigma\} \vdash \Phi : \sigma} \quad [Builtin]$$

$$\frac{\Gamma \vdash C : \sigma_1 \to \ldots \to \sigma_n \to \sigma \quad \Gamma \vdash (x_1 : \tau_1) \ldots (x_n : \tau_n)}{\Gamma \vdash (C\ x_1\ \ldots\ x_n) : \sigma}, \ \tau_i \preceq \sigma_i \quad [Constr]$$

$$\frac{\Gamma \vdash e : \sigma_1 \to \sigma_2 \quad \Gamma \vdash x : \tau}{\Gamma \vdash (e\ x) : \sigma_2}, \ \tau \preceq \sigma_1 \quad [App]$$

$$\frac{\Gamma \cup \{x : \sigma\} \vdash e : \tau}{\Gamma \vdash (\lambda x.e) : \sigma \to \tau} \quad [Abstr]$$

$$\frac{\Gamma \cup \{x_i : \sigma_i\} \vdash e : \sigma \quad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash (let\ (x_1 = e_1) \ldots (x_n = e_n)\ in\ e) : \sigma}, \ \tau_i \preceq \sigma_i, \ 1 \leq i \leq n \quad [Let]$$

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma_i \vdash (C\ x_{i_1}\ \ldots\ x_{i_n}) : \sigma_i \quad \Gamma_j \vdash e_j : \sigma_j}{\Gamma, \Gamma_i, \Gamma_j \vdash (case\ e\ of\ (C_k\ x_{k_1}\ \ldots\ x_{k_{n_k}} \to e_k)) : \sigma_k}, \ 1 \leq i \leq k < j \leq 2k \quad [Case]$$

3.3.7 Type Safety

We distinguish between languages that check types at compile-time and those that check type tags at runtime. CLOVER checks all types at compile-time and catches *all* type errors statically, yet still supports dynamic despatch. Thus, no type errors can occur at run-time — this is our definition of type safety. However, our system does not, of course, catch all possible errors (e.g. division by zero and non-termination).

In particular, we contend that our system will never send a message to an object that does not have a corresponding method implementation for that message. Briefly, this property can be argued for CLOVER as follows:

- if a method exists for a class, it is defined to exist for all subclasses (due to the property of monotonic inheritance introduced previously);

- an object supports exactly those methods that are defined for its class;

- each method application is statically type-checked using the explicitly-annotated *upper bound* types of the arguments and the distinguished object;

- if the type-checker proves the existence of the method for the upper bound type of the distinguished object, and if at runtime the distinguished object must be either of that upper bound type or of a subclass, then the method must exist for the actual object present at runtime.

Proofs of soundness and completeness are of course required for the CLOVER type system in order to demonstrate the type safety of all CLOVER language constructs, including message sending, object construction and application of language primitives. In particular, we will require a subject reduction theorem (e.g. see [Mil78]) to demonstrate that types are preserved under computations. However, such proofs are beyond the scope of this thesis and are therefore left for future work.

3.4 Concrete Syntax

The complete concrete syntax for CLOVER is specified below. This includes constructs for defining a hierarchy with classes containing attributes and methods, constructors, type declarations and all language keywords.

$$
\begin{aligned}
\mathit{Program} &::= \mathit{Node}\ \mathit{Invocation} \\
\mathit{Invocation} &::= \texttt{'invocation'}\ \mathit{Expression} \\
\mathit{Node} &::= \mathit{Class}\ \mathit{Subclasses}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Class} &::= \texttt{'class'}\ \mathit{Identifier}\ \mathit{Attributes}\ \mathit{Methods} \\
\mathit{Attributes} &::= \texttt{'attributes'}\ \texttt{'\{'}\ \mathit{Attribute}^{*}\ \texttt{'\}'} \\
\mathit{Attribute} &::= \mathit{Identifier}\ \mathit{Typing}\ \texttt{';'} \\
\mathit{Methods} &::= \texttt{'methods'}\ \texttt{'\{'}\ \mathit{Method}^{*}\ \texttt{'\}'} \\
\mathit{Method} &::= \mathit{Identifier}\ \mathit{Typing}\ \texttt{'\{'}\ \mathit{MethodDef}\ \texttt{'\}'} \\
\mathit{MethodDef} &::= \mathit{Identifier}^{+}\ \texttt{'='}\ \mathit{Expression} \\
\mathit{Subclasses} &::= \texttt{'subclasses'}\ \texttt{'\{'}\ \mathit{Node}^{*}\ \texttt{'\}'}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Expression} &::= \mathit{Identifier}\ \mathit{Typing} \\
&\mid \mathit{Expression}\ \mathit{Expression}\ \mathit{Typing} \\
&\mid \texttt{'('}\ \mathit{Expression}\ \texttt{')'}\ \mathit{Typing} \\
&\mid \texttt{'let'}\ \texttt{'\{'}\ \mathit{Binding}^{*}\ \texttt{'\}'}\ \texttt{'in'}\ \mathit{Expression}\ \mathit{Typing} \\
&\mid \texttt{'new'}\ \mathit{Identifier}\ \mathit{Constrs}\ \mathit{Typing} \\
&\mid \mathit{Literal}\ \mathit{Typing} \\
\mathit{Binding} &::= \mathit{Identifier}\ \texttt{'='}\ \mathit{Expression}\ \texttt{';'} \\
\mathit{Constrs} &::= \texttt{'()'} \mid \texttt{'('}\ \mathit{ConstrArgs}\ \texttt{')'} \\
\mathit{ConstrArgs} &::= \mathit{Expression} \mid \mathit{Expression}\ \texttt{';'}\ \mathit{ConstrArgs} \\
\mathit{Literal} &::= \texttt{'True'} \mid \texttt{'False'} \mid \texttt{CHARACTER} \mid \texttt{NUMBER}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Typing} &::= \texttt{'::'}\ \mathit{Type} \\
\mathit{Type} &::= \mathit{Identifier} \\
&\mid \mathit{Type}\ \texttt{'->'}\ \mathit{Type} \\
&\mid \texttt{'('}\ \mathit{Type}\ \texttt{'->'}\ \mathit{Type}\ \texttt{')'} \\
&\mid \texttt{'bool'} \mid \texttt{'char'} \mid \texttt{'num'}
\end{aligned}
$$

3.5 Abstract Expression Semantics

In this section, we provide a full definition of CLOVER's abstract expression semantics. This includes the key issues of subsumption, (static) overloading resolution and (dynamic) method despatch. Note that the complete translation of CLOVER into a standard functional language is defined in Chapter 4. We use the semantic functions $\mathcal{E}$ and $\mathcal{K}$ to map from syntactic to semantic domains:

$$\mathcal{E} :: Expressions \rightarrow Environments \rightarrow Semantic\,Values$$
$$\mathcal{K} :: Constants \rightarrow Semantic\,Values$$

$\mathcal{K}$ is pre-loaded with the semantic definitions of the primitive methods, literals and constructors. Similarly, the syntactic definitions of the user-defined methods are pre-loaded into the syntactic function *select*:

$$select :: Expressions \rightarrow Methods \rightarrow Expressions$$

In the following equations, $(select \quad e^{\sigma_e} \quad m^{\sigma_{x_1} \rightarrow \sigma_{x_2} \rightarrow \ldots \rightarrow \sigma_{x_n} \rightarrow \kappa})$ dynamically despatches (returns the lambda abstraction associated with) method m from class σ_e at overloaded type $\sigma_{x_1} \rightarrow \sigma_{x_2} \rightarrow \ldots \rightarrow \sigma_{x_n} \rightarrow \kappa$. In essence, the equation containing *select* illustrates how the last argument of a method application is given special status as the distinguished object and the method definition is determined by reference to the type of that distinguished object. Note that the special bracketing syntax for types is required so that we may detect the distinguished object for a method which returns a message. Also, subtype constraints are checked statically based on upper bound types.

Whereas the method definitions are pre-loaded statically into the semantic function $\mathcal{K}$ and the syntactic function *select*, the binding of variable names to values is achieved dynamically with the environment ρ; this environment maps names to syntactic values. For convenience, syntactic and semantic names are drawn from the same set of identifiers. The environment uses push-semantics and is passed by value into a local context; thus, on return from the local context, the previous version of the environment is available and so nested definitions are permitted. When the environment is searched, a LIFO search is used for the first occurrence of a binding for the given name (thereby avoiding any conflicting semantics).

$$\mathcal{E}\,[\![C^{\sigma_1\to\ldots\to\sigma_n\to\sigma}\ x_1^{\sigma_1}\ \ldots\ x_n^{\sigma_n}]\!]\,\rho \;=\; (\mathcal{K}\,[\![C^{\sigma_1\to\ldots\to\sigma_n\to\sigma}]\!])\,\varepsilon_1\ \ldots\ \varepsilon_n$$
$$\text{where}\ \ \varepsilon_i \;=\; \mathcal{E}\,[\![x_i^{\sigma_i}]\!]\,\rho$$

$$\mathcal{E}\,[\![\Phi_i^{\sigma}]\!]\,\rho \;=\; \mathcal{K}\,[\![\Phi_i^{\sigma}]\!]$$

$$\mathcal{E}\,[\![literal^{\sigma}]\!]\,\rho \;=\; \mathcal{K}\,[\![literal^{\sigma}]\!]$$

$$\mathcal{E}\,[\![x^{\sigma}]\!]\,\rho \;=\; \mathcal{E}\,[\![\rho\,(\,x^{\sigma}\,)]\!]$$

$$\mathcal{E}\,[\![m^{\sigma_{x_1}\to\sigma_{x_2}\to\ldots\to\sigma_{x_n}\to\tau}\ e_1^{\sigma_{e_1}}\ e_2^{\sigma_{e_2}}\ \ldots\ e_n^{\sigma_{e_n}}]\!]\,\rho$$
$$=\; \mathcal{E}\,[\![(\lambda x\,.\,e)^{\sigma_{x_1}\to\sigma_{x_2}\to\ldots\to\sigma_{x_n}\to\tau}\ e_1^{\sigma_{e_1}}\ e_2^{\sigma_{e_2}}\ \ldots\ e_n^{\sigma_{e_n}}]\!]\,\rho,$$
$$n\geq 1,\ \tau=(\kappa\ \text{or}\ (\sigma_1\to\sigma_2))$$
$$\text{where}\ \ (\lambda x\,.\,e)\;=\;select\ e_n^{\sigma_{e_n}}\ m^{\sigma_{x_1}\to\sigma_{x_2}\to\ldots\to\sigma_{x_n}\to\tau}$$

$$\mathcal{E}\,[\![(\lambda x\,.\,e)^{\sigma_{x_1}\to\sigma_{x_2}\to\ldots\to\sigma_{x_n}\to\tau}\ e_1^{\sigma_{e_1}}\ e_2^{\sigma_{e_2}}\ \ldots\ e_n^{\sigma_{e_n}}]\!]\,\rho$$
$$=\; \mathcal{E}\,[\![e^{\sigma_{x_2}\to\ldots\to\sigma_{x_n}\to\tau}\ e_2^{\sigma_{e_2}}\ \ldots\ e_n^{\sigma_{e_n}}]\!]\,\rho(x:=e_1),$$
$$n\geq 1,\ \ \sigma_{e_1}\preceq\sigma_{x_1},\ \tau=(\kappa\ \text{or}\ (\sigma_1\to\sigma_2))$$

$$\mathcal{E}\,[\![m^{\sigma_1\to\ldots\to\sigma_n\to\tau}]\!]\,\rho \;=\; \lambda x_1\ldots x_n\,.\,(\mathcal{E}[\![m^{\sigma_1\to\ldots\to\sigma_n\to\tau}\ x_1\ldots x_n]\!]\,\rho),$$
$$n\geq 1,\ \tau=(\kappa\ \text{or}\ (\sigma_x\to\sigma_y))$$

$$\mathcal{E}\,[\![(\lambda x\,.\,e)^{\sigma_1\to\sigma_2}]\!]\,\rho \;=\; \lambda x\,.\,(\mathcal{E}\,[\![e^{\sigma_2}]\!]\rho)$$

$$\mathcal{E}\,[\![e^{\sigma_1\to\sigma_2}\ x^{\sigma_x}]\!]\,\rho \;=\; (\mathcal{E}\,[\![e^{\sigma_1\to\sigma_2}]\!]\,\rho)\,(\mathcal{E}\,[\![x^{\sigma_x}]\!]\,\rho),\ \ \sigma_x\preceq\sigma_1$$

$$\mathcal{E}\,[\![let\,(x_1=e_1^{\sigma_1})\ldots(x_n=e_n^{\sigma_n})\ \ in\ e^{\sigma_e}]\!]\,\rho \;=\; \mathcal{E}\,[\![e^{\sigma_e}]\!]\,\rho(x_1:=e_1^{\sigma_1},\ \ldots\ ,x_n:=e_n^{\sigma_n})$$

$$\mathcal{E}\,[\![case\,e^{\sigma}\ of\,(C_1^{\sigma_1}\ x_{11}^{\sigma_{11}}\ \ldots\ x_{1n_1}^{\sigma_{1n_1}}\ \to\ e_1^{\sigma_1}\ ;$$
$$C_j^{\sigma_j}\ x_{j1}^{\sigma_{j1}}\ \ldots\ x_{jn_j}^{\sigma_{jn_j}}\ \to\ e_j^{\sigma_j}\ ;$$
$$C_m^{\sigma_m}\ x_{m1}^{\sigma_{m1}}\ \ldots\ x_{mn_m}^{\sigma_{mn_m}}\ \to\ e_m^{\sigma_m})]\!]\,\rho$$
$$=\; \mathcal{E}\,[\![e_j^{\sigma_j}]\!]\,\rho'$$
$$\text{where}\ \ e^{\sigma}=\ C_j^{\sigma_j}\ e_{j1}^{\sigma_{j1}}\ e_{jn_j}^{\sigma_{jn_j}}$$
$$\rho'\ =\ \rho(x_{j1}:=e_{j1}^{\sigma_{j1}},\ \ldots\ ,x_{jn_j}:=e_{jn_j}^{\sigma_{jn_j}})$$
$$C_1\neq C_j,\ \ldots\ ,C_{j-1}\neq C_j$$

3.6 Referential Transparency

The term "referential transparency" was first used in [WR13] to describe an aspect of syllogistic logic when only the content of a statement is required, not its circumstances. The term was subsequently introduced into computing in [Lan64] and has become an important property of functional languages.

We adopt the following definition, taken from [Dil88], where a language is referentially transparent if it satisfies the following two conditions:

- Different occurrences of the same expression in a single scope have the same meaning;
- Two different expressions which have the same meaning anywhere in a single scope have the same meaning throughout that scope and, hence, can be substituted for each other anywhere in that scope.

We argue informally about CLOVER's property of referential transparency by referring to the abstract expression semantics in the previous section. The above concept of a "single scope" is defined for our purposes as a constant value for the environment ρ.

Values of base-case expressions are computed by applying either: (i) the semantic function $\mathcal{K}$ (to literals or primitives), or (ii) the semantic function $\mathcal{E}$ (to variables and the environment ρ). In the former case, $\mathcal{K}$ is pre-loaded with bindings that never change, whereas in the latter case selection of a binding from the environment is achieved using a LIFO search and therefore, for a given environment and name, will always return the same value. Furthermore, the environment ρ is only modified as the result of a let or λ expression, and in these two situations the environment is *extended*; existing bindings in the environment are not modified.

Some care must be taken with the operation of the $select$ function, which determines the value of a method name for a fully-saturated application. It suffices to note that $select$ always returns the same result when passed the same arguments.

A formal proof of referential transparency for CLOVER (proceeding via structural induction on the full program semantics) is beyond the scope of this thesis and is therefore left for future work. However, the preceding informal argument (coupled with the abstract expression semantics) provides strong support for our contention that CLOVER is referentially transparent.

3.7 Summary

In this chapter, we have presented a new language CLOVER, its design features, concrete and abstract syntax syntax, type system (including typing rules), abstract expression semantics and addressed referential transparency. This language meets almost all of our requirements by the careful integration of a number of different design criteria. The remaining requirement, object identity, represents a core incompatibility that is resolved in Chapter 5 by supplying new interpretations using a new visual notation.

Chapter 4

The Implementation of CLOVER: A Translation from OOFP to FP

In this chapter, we present the key stages of an implementation of CLOVER as a set of rewrite rules. Many implementation routes are possible, but we choose a functional language as our target code. This permits the exploitation of automatic memory management, lazy evaluation and Currying (all of which become available at zero implementation cost). Furthermore, we can generate concise understandable code (which is more important than execution speed for our prototype implementation).

In order to provide a focussed discussion, we ignore all issues related to type checking and assume that this has already been performed (see Appendix A for type checker). We similarly assume that the straightforward translation from CLOVER's visual notation, Object-Flow, to a textual code has already been performed. The goal is to describe the translation from an abstract form of this code, Abstract Program Code (APC_{IO}), into an abstract form of a simple functional code, Target Code (TC). We give detailed algorithms, to provide a basis for others to explore and further develop these ideas. We also provide the first demonstration of how to support completely type-safe dynamic despatch in the presence of (partially-applied) higher-order methods.

4.1 Overview of the Transformation Rules

In this section, we present an overview of the CLOVER transformation rules and introduce our notations. A CLOVER program is transformed into a standard functional program, then compiled and executed. We express the program transformation as a set

of high-level translations, as illustrated in Table 4.1 (the terms are expanded in Table 4.2 and explained fully in subsequent sections).

Table 4.1: Summary of Main CLOVER Translations

Translation	Source	Target	Description
$\mathcal{P}$	Object-Flow	APC_{IO}	Parses visual language into abstract code
$\mathcal{R}$	APC_{IO}	APC_{I}	Resolves overloaded methods
$\mathcal{I}$	APC_{I}	APC	Expands inherited attributes and methods
$\mathcal{F}$	APC	IDS	Flattens and inverts program structure
$\mathcal{G}$	IDS	TC	Generates target code

Table 4.2: Terms Used During Translations

Term	Description
APC_{IO}	Abstract Program Code with Inheritance and Overloading
APC_{I}	Abstract Program Code with Inheritance
APC	Abstract Program Code
IDS	Intermediate Data Structure
TC	Target Code

We define the translation $\mathcal{T}$ of Object-Flow into TC as the complete transformation, composing the other translations as shown below:

Translation of Object-Flow into TC

$$\mathcal{T} :: Program_{OF} \rightarrow Program_{TC}$$
$$\mathcal{T}[\,of\,] \;=\; \mathcal{G} \circ \mathcal{F} \circ \mathcal{I} \circ \mathcal{R} \circ \mathcal{P}[\,of\,]$$

As discussed above, presentation of the first translation, $\mathcal{P}$, is omitted from this thesis. The second translation, $\mathcal{R}$, resolves method overloading by transforming APC_{IO} into APC_{I}. The third translation, $\mathcal{I}$, expands inherited attributes and methods by

transforming APC_I into APC. The fourth translation, $\mathcal{F}$, flattens and inverts the code to create an intermediate data structure (that is more amenable to later functional code generation) by transforming APC into IDS. Finally, the translation $\mathcal{G}$ generates target code in a simple abstract functional language.

4.1.1 Notation

We use two main notations within this chapter: one to define the abstract syntax of our languages and data structures, and the other to define the transformation rules which operate on them.

We define abstract syntax by induction on syntactic structure, using Extended-BNF notation. We use ::= to define syntactic categories, ( ) for grouping, [ ] to indicate zero-or-one, * to indicate zero-or-more, $^+$ to indicate one-or-more, | to separate alternatives and ' ' for terminals.

We define transformation rules to manipulate syntactic categories directly. The rules are higher-order and exploit curried notation, with rule names not italicised so that curried rule applications, e.g. "(expAttrs ns $asInh$)", can be differentiated from sequences of syntactic categories, e.g. "(cId $attrs$ ms)". There is no special bracketing notation for lists; instead, we define the following operators and rules for direct manipulation of syntactic categories.

Operators and transformation rules for list functionality

$$
\begin{array}{lcl}
x \oplus (y_1 \ldots y_n) & = & (x\ y_1 \ldots y_n) \\
(x_1 \ldots x_n) \otimes (y_1 \ldots y_n) & = & (x_1 \ldots x_n\ y_1 \ldots y_n) \\
\text{hd}\ (x_1 \ldots x_n) & = & x_1 \\
\text{tl}\ (x_1\ x_2 \ldots x_n) & = & (x_2 \ldots x_n) \\
\text{map}\ \text{f}\ () & = & () \\
\text{map}\ \text{f}\ (x \oplus xs) & = & (\text{f}\ x) \oplus (\text{map}\ \text{f}\ xs) \\
\text{concat}\ () & = & () \\
\text{concat}\ (x \oplus xs) & = & x \otimes (\text{concat}\ xs) \\
\text{mapconcat}\ \text{f}\ xs & = & \text{concat}\ (\text{map}\ \text{f}\ xs)
\end{array}
$$

4.2 Abstract Program Code (APC)

We use "Abstract Program Code" to describe three language forms that are syntactically very similar: (i) Abstract Program Code with Inheritance and Overloading (APC_{IO}), (ii) Abstract Program Code with Inheritance (APC_{I}), and (iii) Abstract Program Code (APC).

APC_{IO} provides all the object-oriented features discussed in Chapter 1. After an Object-Flow program has been parsed into APC_{IO}, it is first translated into APC_{I} (which has no method overloading) and then into APC (which has explicit, rather than implicit, inherited attributes and methods). The syntax of APC_{IO} and APC_{I} are identical, and are almost identical to APC. The semantics of these language forms are, however, considerably different and the translations must, of course, preserve the original program semantics.

4.2.1 Syntax of Abstract Program Code

APC is a pair consisting of a class hierarchy and a program invocation definition (see below). The class hierarchy is a recursively-defined tree structure with each node consisting of a class definition and its subclasses. A class definition contains the class name, a list of attributes and a list of methods. An attribute consists simply of an attribute name and its type, and a method consists of a method name, its type and an expression representing the method implementation. Similarly, the program invocation definition is an expression (except that the semantics prohibit it from containing λ-abstractions).

The expression syntax is based on the typed λ-calculus, with the addition of object constructors which require the type (i.e. the class name) of the object to be constructed together with a list of arguments. The constructor definitions themselves have the same syntax as method definitions, although the semantics are somewhat different (see Section 4.4.2). Note that all expressions are annotated with an explicit type tag.

APC's type system is purely object-oriented. Values can be objects (of class type), messages (of function type) or literals (of primitive type). Note the bracketed function types; these are necessary for higher-order messages so that we may identify the distinguished object — see Chapter 3 for full details.

Syntax of APC_{IO} and APC_{I}

$$
\begin{array}{rcl}
Program_{APC} & ::= & Node\ PrInvDef \\
PrInvDef & ::= & Expr \\
Node & ::= & ClassDef\ Node^{*} \\
\\
ClassDef & ::= & ClassName\ Attribute^{*}\ Method^{*} \\
Attribute & ::= & AttributeName\ Type \\
Method & ::= & MethodName\ Type\ Expr \\
\\
Expr & ::= & \texttt{'Object'}\ ClassName\ Expr^{*}\ Type \\
 & | & \texttt{'Variable'}\ VariableName\ Type \\
 & | & \texttt{'Variable'}\ MethodName\ Type \\
 & | & \texttt{'Builtin'}\ PrimFunction\ Type \\
 & | & \texttt{'Apply'}\ Expr\ Expr\ Type \\
 & | & \texttt{'Lambda'}\ VariableName\ Expr\ Type \\
 & | & \texttt{'Let'}\ Binding^{*}\ Expr\ Type \\
 & | & \texttt{'Literal'}\ PrimValue\ Type \\
Binding & ::= & VariableName\ Expr \\
PrimValue & ::= & \texttt{'BoolVal' bool} \mid \texttt{'CharVal' char} \mid \texttt{'NumVal' num} \\
PrimFunction & ::= & \texttt{'Plus'} \mid \texttt{'Minus'} \mid \texttt{'Multiply'} \mid \texttt{'Divide'} \mid \ldots \\
 & | & \texttt{'If'} \mid \texttt{'And'} \mid \texttt{'Or'} \mid \texttt{'Not'} \mid \ldots \\
 & | & \texttt{'Equal'} \mid \texttt{'GreaterThan'} \mid \texttt{'LessThan'} \mid \ldots \\
\\
Type & ::= & \texttt{'ClassType'}\ ClassName \\
 & | & \texttt{'FunctionType'}\ Type\ Type \\
 & | & \texttt{'BrFunctionType'}\ Type\ Type \\
 & | & \texttt{'PrimitiveType'}\ PrimType \\
PrimType & ::= & \texttt{'bool'} \mid \texttt{'char'} \mid \texttt{'num'}
\end{array}
$$

$$ClassName, AttributeName, MethodName, VariableName \in \{Identifiers\}$$

The syntax of APC is defined by extending APC_{IO} and APC_{I} with `Reused` expressions. Such expressions are created during the expansion of inherited methods (see later trans-

lation) and indicate explicitly that the implementation is inherited (but not overridden) and should be reused from a superclass.

Syntax of APC

As for APC_{IO} and APC_{I}, except $Expr$ is extended as follows:

$$Expr \quad ::= \quad ... \mid \texttt{'Reused'}\ ClassName$$

4.2.2 Translation of APC_{IO} into APC_{I}

We translate APC_{IO} into APC_{I} by resolving method overloading. This involves generating new names for each separate type-instance of a method definition: these new method names must be used appropriately wherever the overloaded method name is applied, so the entire program must be transformed and the type of every application must be checked.

Overloading resolution proceeds in two phases:

1. Scan all classes and, where overloaded method definitions are found, generate new names and add them to a translation table. Each entry in the table is a triple comprising the overloaded method name, the type, and the resolved method name.

2. Use the translation table from the first phase to tranform the program (i.e. both the left-hand-side and right-hand-side of all method definitions, together with the invocation), so that all occurrences of overloaded method names are translated according to the types at which the methods are applied.

Note that the techniques for resolving method overloading have been implemented in many compilers (Haskell's type classes provide a purely functional example and Smalltalk provides a purely object-oriented example); we, therefore, do not present the algorithm in detail here. However, for a CLOVER example, see the translation of overloaded method `minus` into `minus_1` and `minus_2` in Appendix B.3.

4.2.3 Translation of APC_I into APC

We translate APC_I into APC by making implicit method and attribute inheritance explicit, as shown in translation $\mathcal{I}$ below. The implementations for inherited (i.e. not overridden) methods are not duplicated; instead we make the methods explicit by creating a `Reused` tag as each one's implementation. This results in much shorter target code, but complicates the later generation of method despatchers. The transformation defines the inheritance semantics.

Translation of APC_I into APC

$$\mathcal{I} :: Program_{APC_I} \rightarrow Program_{APC}$$

$$\mathcal{I}[\ apci\ ] \ = \ (h''\ \ pid)$$

$$\textbf{where} \quad h'' = \text{expMeths}\ (\ )\ h'$$

$$h' = \text{expAttrs}\ (\ )\ h$$

$$(h\ \ pid) = apci$$

The rule "expAttrs" takes an (initially empty) list of attributes and an (initially complete) class hierarchy, and expands the inherited attributes. The resulting class hierarchy contains explicit definitions for every class's attributes, including inherited ones. The supporting function "comAttrs" combines inherited and introduced attributes to produce a complete set of attributes for a particular class. Note that, unlike traditional OOP languages, CLOVER does not give special status to built-in (primitive) classes such as `Number`: in CLOVER, it is possible to define subclasses of the primitive classes (for example, `BoundedNumber`). However, this requires that if a primitive class (or a subclass of a primitive class) is being extended for the first time, then the inherited attribute of primitive type (i.e. a type that is built-in, rather than being associated with a class) must be coerced to the appropriate class type.

For a CLOVER example, see the inherited primitive attribute `prim` and the introduced attributes `upperbound` and `lowerbound` in class `BoundedNumber` in Appendix B.3.

Rule for expanding inherited attributes

$$
\begin{array}{l}
\text{expAttrs} :: \mathit{Attribute}^* \rightarrow \mathit{Node} \rightarrow \mathit{Node} \\
\text{expAttrs}\ \mathit{asInh}\ (\mathit{cDef}\ \mathit{ns}) \ = \ (\mathit{cDef}'\ \mathit{ns}') \\
\qquad\qquad\qquad\qquad \textbf{where}\ \ \mathit{cDef}' = (\mathit{cId}\ \mathit{as}'\ \mathit{ms}) \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathit{as}' = \text{comAttrs}\ \mathit{asInh}\ \mathit{as} \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathit{ns}' = \textbf{map}\ (\text{expAttrs}\ \mathit{as}')\ \mathit{ns} \\
\qquad\qquad\qquad\qquad\qquad\ \ (\mathit{cId}\ \mathit{as}\ \mathit{ms}) = \mathit{cDef}
\end{array}
$$

$$
\begin{array}{lll}
\text{comAttrs} :: \mathit{Attribute}^* \rightarrow \mathit{Attribute}^* \rightarrow \mathit{Attribute}^* & & \\
\text{comAttrs}\ \ () & \mathit{asInt} \ = & \mathit{asInt} \\
\text{comAttrs}\ \ \mathit{asInh} & () \ = & \mathit{asInh} \\
\text{comAttrs}\ \ ((n\ (\texttt{'PrimType'}\ p)) \oplus \mathit{asInh}) & \mathit{asInt} \ = & (n\ (\texttt{'PrimType'}\ p)) \oplus \\
 & & (\mathit{asInh} \otimes \mathit{asInt}) \\
\text{comAttrs}\ \ \mathit{asInh} & \mathit{asInt} \ = & \mathit{asInh} \otimes \mathit{asInt}
\end{array}
$$

The rule "expMethods" takes an (initially empty) list of methods and an (initially complete) class hierarchy, and expands the inherited (i.e. not overridden) methods. The resulting class hierarchy contains explicit method definitions for every class's methods, including inherited ones. We merely create a `Reused` tag for every inherited method's implementation, providing an indirection to a superclass containing the most recent implementation. For a CLOVER example, see the inherited implementation for method `dec` in class `BoundedNumber` in Appendix B.3.

Rule for expanding inherited methods

$$
\begin{array}{l}
\text{expMeths} :: \mathit{Method}^* \rightarrow \mathit{Node} \rightarrow \mathit{Node} \\
\text{expMeths}\ \mathit{msInh}\ (\mathit{cDef}\ \mathit{ns}) \ = \ (\mathit{cDef}'\ \mathit{ns}') \\
\qquad\qquad\qquad\qquad \textbf{where}\ \ \mathit{cDef}' = (\mathit{cId}\ \mathit{as}\ \mathit{ms}') \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathit{ms}' = \text{expMs}\ \mathit{ms}\ \mathit{msInh} \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathit{ns}' = \textbf{map}\ (\text{expMeths}\ \mathit{msInh}')\ \mathit{ns} \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathit{msInh}' = \text{upMs}\ \mathit{msInh}\ \mathit{ms}\ \mathit{cId} \\
\qquad\qquad\qquad\qquad\qquad\ \ (\mathit{cId}\ \mathit{as}\ \mathit{ms}) = \mathit{cDef}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{expMs} :: Method^* \to Method^* \to Method^* & & \\
\textbf{expMs} \;\; ms \;\; () & = & ms \\
\textbf{expMs} \;\; ms \;\; (mInh \oplus msInh) & = & \textbf{expMs} \;\; ms' \;\; msInh \\
 & & \textbf{where} \; ms' = \textbf{expM} \;\; ms \;\; mInh
\end{array}
$$

$$
\begin{array}{lll}
\textbf{expM} :: Method^* \to Method \to Method^* & & \\
\textbf{expM} \;\; () \;\; mInh & = & (mInh) \\
\textbf{expM} \;\; (m \oplus ms) \;\; mInh & = & m \oplus ms, \; \textbf{if} \; mId = mInhId \\
 & = & m \oplus (\textbf{expM} \;\; ms \;\; mInh), \; \textbf{otherwise} \\
 & & \textbf{where} \;\; (mIdInh \;\; mTInh \;\; mEInh) = mInh \\
 & & \quad (mId \;\; mT \;\; mE) = m
\end{array}
$$

$$
\begin{array}{lll}
\textbf{upMs} :: Method^* \to Methods^* \to Name \to Method^* & & \\
\textbf{upMs} \;\; msInh \;\; () \;\; cId & = & msInh \\
\textbf{upMs} \;\; msInh \;\; (m \oplus ms) \;\; cId & = & \textbf{upMs} \;\; msInh' \;\; ms \;\; cId \\
 & & \textbf{where} \; msInh' = \textbf{upM} \;\; msInh \;\; m \;\; cId
\end{array}
$$

$$
\begin{array}{lll}
\textbf{upM} :: Method^* \to Method \to ClassName \to Method^* & & \\
\textbf{upM} \;\; methsInh \;\; m \;\; cId & = & (mInh'), \; \textbf{if} \; methsInh = () \\
 & = & mInh' \oplus msInh, \; \textbf{if} \; mIdInh = mId \\
 & = & mInh \oplus (\textbf{upM} \;\; msInh \;\; m \;\; cId), \; \textbf{otherwise} \\
 & & \textbf{where} \;\; mInh' = (mId \;\; mT \;\; (\texttt{'Reused'} \;\; cId)) \\
 & & \quad (mIdInh \;\; mTInh \;\; mEInh) = mInh \\
 & & \quad (mId \;\; mT \;\; mE) = m \\
 & & \quad (mInh \oplus msInh) = methsInh
\end{array}
$$

4.3 Intermediate Data Structure (IDS)

The compilation of APC into TC is a complex transformation comprising two main tasks: (i) reorganising the program structure by flattening the hierarchy and inverting the class-method containment relationship, and (ii) generating functional target code. We use an intermediate data structure (IDS) to pass the results from the first task (translation $\mathcal{F}$) to the second task (translation $\mathcal{G}$). This data structure has been designed to organise object-oriented functional programs such that subsequent generation of standard functional code is simplified.

4.3.1 Syntax of Intermediate Data Structure

The intermediate data structure is a triple consisting of a list of IDS class definitions, a list of IDS method definitions and a program invocation definition. Being a "flattened" form of APC (i.e. without an explicit class hierarchy), the class definitions and method definitions must now be self-contained. We have thus changed from a "hierarchy of classes (each class containing its methods and attributes)" representation to a "classes (containing attributes) and methods (containing associations of implementations to classes)" representation, which is equivalent to flattening and inverting.

Each IDS class definition is simply a class name and a list of attributes. IDS method definitions are more complex as each one must encapsulate all the functionality for a particular method, including the information required to perform appropriate dynamic despatch (based on method reuse and overriding). Each method definition therefore consists of a method name, a type and a list of implementation definitions. Each implementation definition contains a method implementation (i.e. an expression) and a list of class names for which this implementation is defined.

Note that an implementation is defined for the class in which it is introduced and the subclasses in which it is reused, but not for any subclasses in which it is overridden. Also note that IDS shares the syntax of expressions, attributes, invocation and types with APC (defined in Section 4.2.1).

Syntax of IDS

$$
\begin{aligned}
Program_{IDS} &::= ClDef^{*}\ MethDef^{*}\ PrInvDef \\
ClDef &::= ClassName\ Attribute^{*} \\
MethDef &::= MethodName\ Type\ ImpDef^{+} \\
ImpDef &::= Expr\ ClassName^{+}
\end{aligned}
$$

4.3.2 Translation of APC into IDS

We translate APC into IDS by flattening and inverting the class hierarchy. This creates IDS class definitions and IDS method definitions (and also retains the program invocation definition), as shown in $\mathcal{F}$ below.

Translation of APC into IDS

$$
\begin{aligned}
&\mathcal{F} :: Program_{APC} \rightarrow Program_{IDS} \\
&\mathcal{F}[\,apc\,] \;=\; (clDefs\ methDefs\ pid) \\
&\qquad\qquad \textbf{where}\ \ clDefs = \text{createClDefs}\ h \\
&\qquad\qquad\qquad\quad\ \ methDefs = \text{createMethDefs}\ h\ (\,) \\
&\qquad\qquad\qquad\quad\ \ (h\ pid) = apc
\end{aligned}
$$

The rule "createClDefs" takes an (initially complete) class hierarchy and creates the IDS class definitions by simply extracting the class name and attributes from each class and flattening the hierarchy, as shown below. For a CLOVER example, see the definition for class `BoundedNumber` in Appendix B.4.

Rule for creating IDS class definitions

$$
\begin{aligned}
&\text{createClDefs} :: Node \rightarrow ClDef^{*} \\
&\text{createClDefs}\ (cDef\ ns) \;=\; (cId\ as) \oplus (\text{mapconcat createClDefs}\ ns) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\ (cId\ as\ ms) = cDef
\end{aligned}
$$

The rule "createMethDefs" takes an (initially complete) class hierarchy and an (initially empty) list of method definitions and creates IDS method definitions. The rule is more complex than "createClDef" as we must encapsulate all the functionality for each method (such as implementation reuse implicitly defined by subclassing — later required for determining dynamic despatch) as we flatten the hierarchy and invert the class-method containment relationship. This requires us to maintain a list of the method implementations, overridings and reuses defined so far for each method when descending the hierarchy.

The core of the rule is the insertion of method implementations in the supporting rule "insImp". This handles the insertion of a new method (by creating a new method definition), an overriding (by adding a new implementation definition in the appropriate method definition) and a reuse (by adding a new class name in the appropriate implementation definition).

For CLOVER examples, see Appendix B.4: (i) the single method implementation for `fac` associated to the two classes `Number` and `BoundedNumber`, and (ii) the two method implementations for `minus_1`, one associated with class `Number` and the other with class `BoundedNumber`.

Rule for creating IDS method definitions

$$\text{createMethDefs} :: Node \rightarrow MethDef^* \rightarrow MethDef^*$$
$$\text{createMethDefs}\ ((cId\ as\ ms)\ ns)\ msDef$$
$$= \text{crMethDefs}\ ns\ (\text{insImps}\ msDef\ ms\ cId)$$

$$\text{crMethDefs} :: Node^* \rightarrow MethDef^* \rightarrow MethDef^*$$
$$\text{crMethDefs}\ ()\ msDef = msDef$$
$$\text{crMethDefs}\ (n \oplus ns)\ msDef = \text{crMethDefs}\ ns\ (\text{createMethDefs}\ n\ msDef)$$

$$\text{insImps} :: MethDef^* \rightarrow Method^* \rightarrow ClassName \rightarrow MethDef^*$$
$$\text{insImps}\ msDef\ ()\ cId = msDef$$
$$\text{insImps}\ msDef\ (m \oplus ms)\ cId = \text{insImps}\ (\text{insImp}\ msDef\ m\ cId)\ ms\ cId$$

$$
\begin{array}{l}
\text{insImp} :: MethDef^* \rightarrow Method \rightarrow ClassName \rightarrow MethDef^* \\
\text{insImp}\ msDef\ (mId\ mT\ mE)\ cId \\
\quad = imp' \oplus imps \\
\qquad \textbf{where}\ \ imp' = (mId\ mT\ ((mE\ (cId)))),\ \textbf{if}\ imp = (\,) \\
\qquad\qquad\qquad\ = (mIdI\ mTI\ (\text{appImp}\ impsI\ cId)),\ \textbf{if reused}\ mE \\
\qquad\qquad\qquad\ = (mIdI\ mTI\ (impsI \otimes (mE\ (cId)))),\ \textbf{otherwise} \\
\qquad\qquad (mIdI\ mTI\ impsI) = \text{hd}\ imp \\
\qquad\qquad (imp\ imps) = \text{spDef}\ msDef\ mId\ (\,) \\
\qquad\qquad \text{reused}\ (\texttt{'Reused'}\ cIdI) = \textbf{True} \\
\qquad\qquad \text{reused}\ e = \textbf{False}
\end{array}
$$

$$
\begin{array}{lllll}
\text{spDef} :: MethDef^* \rightarrow MethodName \rightarrow MethDef^* \rightarrow (MethDef^*\ MethDef^*) & & & & \\
\text{spDef}\ \ (\,) & mId & accDef & = & ((\,)\ accDef) \\
\text{spDef}\ \ (mDef \oplus msDef) & mId & accDef & = & ((mDef)\ accDef \otimes msDef), \\
 & & & & \textbf{if}\ mId = mIdI \\
 & & & = & \text{spDef}\ msDef\ mId \\
 & & & & (accDef \otimes (mDef)),\ \textbf{otherwise} \\
 & & & & \textbf{where}\ (mIdI\ mTI\ impsI) = mDef
\end{array}
$$

$$
\begin{array}{lllll}
\text{appImp} :: ImpDef^* \rightarrow ClassName \rightarrow ImpDef^* & & & & \\
\text{appImp}\ \ ((e\ cIds) \oplus (\,)) & cId & = & (e\ (cIds \otimes (cId))) & \\
\text{appImp}\ \ (i \oplus is) & cId & = & i \oplus (\text{appImp}\ is\ cId) &
\end{array}
$$

4.4 Target Code (TC)

In our prototype implementation, we generate Miranda code but, for the purposes of this thesis, we define a simple abstract functional language, Target Code (TC), which captures the features we require. We assume the properties of a typical modern functional language, including lazy evaluation, curried partial applications and automatic garbage collection.

4.4.1 Syntax of Target Code

TC consists of a single type definition and a list of function definitions. The type definition is an algebraic data type containing a unique constructor and underlying type for every class and message type. All other generated code is in the form of standard function definitions comprising a function name, function arguments and function body (with local definitions in `where` blocks).

Syntax of TC

$$
\begin{array}{lcl}
Program_{TC} & ::= & TarTypeDef\ TarDef^* \\
\\
TarDef & ::= & TarVarName\ TarExpr^*\ '='\ TarExpr \\
TarExpr & ::= & TarVarName \mid TarExpr\ TarExpr \\
 & \mid & '['\ [TarExpr\ (','\ TarExpr)^*]\ ']' \\
 & \mid & '('\ TarConsName\ TarExpr^*\ ')' \\
 & \mid & TarExpr\ \texttt{'where'}\ TarDef^+ \\
 & \mid & '('\ TarExpr\ ')' \mid TarPrimFunc \mid Literal \\
TarPrimFunc & ::= & '+' \mid '-' \mid '*' \mid '/' \mid '\neq' \mid '<' \mid \ldots \\
\\
TarTypeDef & ::= & TarTypeName\ '::='\ TarConsType\ TarConsAlt^* \\
TarConsAlt & ::= & '|'\ TarConsType \\
TarConsType & ::= & TarConsName\ TarType^* \\
TarType & ::= & TarTypeName \mid '['\ TarType\ ']' \\
 & \mid & TarType\ '\rightarrow'\ TarType \\
 & \mid & '('\ TarType\ '\rightarrow'\ TarType\ ')' \\
 & \mid & \texttt{'bool'} \mid \texttt{'char'} \mid \texttt{'num'}
\end{array}
$$

$$TarVarName, TarConsName, TarTypeName \in \{Identifiers\}$$

We define no new type synonyms, do not require type variables and the only aggregate type we use is a list to collect together the attributes for a given class. Note that, to reduce complexity in the transformation rules, primitive functions are prefix and function type signatures are omitted (all types can be deduced in the target language compiler by standard type inference). For an example of our target code, see the generated programs in Section 4.5.

4.4.2 Translation of IDS into TC

We translate IDS into TC by generating the components of the target program, as shown in $\mathcal{G}$ below. The components include, *inter alia*, a program invocation definition, a meta type definition, method despatchers, wrapped method despatchers, method implementations and message appliers.

Translation of IDS into TC

$$
\begin{array}{l}
\mathcal{G} :: Program_{IDS} \rightarrow Program_{TC} \\
\mathcal{G}[\,ic\,] \;=\; (metaDef\;\; (iDef \otimes mWraps \otimes mDesps \otimes mImps \otimes dObjs \otimes apps)) \\
\qquad\qquad \textbf{where}\;\; metaDef = \text{genMetaDef}\;\; clDefs\;\; maxAr \\
\qquad\qquad\qquad\qquad iDef = \text{genInvDef}\;\; pid \\
\qquad\qquad\qquad\qquad mWraps = \text{map}\;\; \text{genMethWrap}\;\; methDefs \\
\qquad\qquad\qquad\qquad mDesps = \text{mapconcat}\;\; \text{genMethDesp}\;\; methDefs \\
\qquad\qquad\qquad\qquad mImps = \text{map}\;\; \text{genMethImps}\;\; methDefs \\
\qquad\qquad\qquad\qquad dObjs = \text{genDefObjs}\;\; clDefs \\
\qquad\qquad\qquad\qquad apps = \text{genAppliers}\;\; maxAr \\
\qquad\qquad\qquad\qquad maxAr = \text{max}\;\; (\text{map}\;\; \text{arity}\;\; methDefs) \\
\qquad\qquad\qquad\qquad (clDefs\;\; methDefs\;\; pid) = ic
\end{array}
$$

In the remainder of this section, we divide the target code generation rules into six categories, corresponding to the component generated. However, we first define a subsidiary rule, "arity", which determines the arity of an IDS method definition.

Rules for determining arities

$$
\begin{array}{lll}
\text{arity} :: MethDef \rightarrow Number & & \\
\text{arity}\;\; (mId\;\; mT\;\; imps) \;=\; \text{ar}\;\; mT & & \\
\qquad\qquad\qquad\qquad \textbf{where}\;\; \text{ar}\;\; (\texttt{'FunctionType'}\;\; t1\;\; t2) & = & 1 + (\text{ar}\;\; t2) \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ar}\;\; t & = & 0
\end{array}
$$

Invocation definition

The rule "genInvDef" takes a program invocation definition and generates the highest-level function (arbitrarily named `main`) in the target code. The function body is generated using the rule "genExpr" (part of method implementation generation). For a CLOVER example, see `main` in Appendix B.5.

Rule for generating a program invocation

$$\text{genInvDef} :: PrInvDef \rightarrow TarDef$$
$$\text{genInvDef} \quad pid \quad = \quad (\text{'main'} \; \text{'='} \; (\text{genExpr} \; pid))$$

Meta type definition

We implement dynamic typing by attaching type information to every object. These "type tags" (corresponding to class names) are constructors for an algebraic type — objects are encoded as the constructor plus a list of attributes or a primitive value. Messages are also represented as a type tag constructor together with the actual message, thereby supporting partially-applied, higher-order functions (the constructor provides arity data that is used at run-time to determine whether the message is fully applied and should be despatched: the algebraic type therefore requires constructors for every message type in the program).

We wish to implement subtyping in the single-rooted class hierarchy via subsumption rather than via coercion, since coercion to a supertype loses information. Given a target functional language that does not support subtyping, and given that in the limit the root class subsumes all other classes, the only way to do this is to utilise a single algebraic type with constructors for every class. Parametric polymorphism (where an argument can be either an object or a message) is implemented by combining the type tags for both objects and messages into a single monolithic algebraic type.

The result is an elegant and flexible meta type representation, generated from the list of IDS class definitions and the maximum method arity by the rule "genMetaDef". For a CLOVER example, see `metatype` in Appendix B.5.

Rule for generating a meta type

```
genMetaDef :: ClDef* → Number → TarTypeDef
genMetaDef clDefs maxAr = ('metatype' '::=' (metaCl ⊗ metaFunc))
                          where metaCl = map genMetaCl clDefs
                                metaFunc = genMetaFunc maxAr

    genMetaCl :: ClDef → TarConsAlt
    genMetaCl (cId ())                          = ('|' cId ())
    genMetaCl (cId (n ('PrimitiveType' p)))     = ('|' cId p)
    genMetaCl (cId as)                          = ('|' cId '[ metatype ]')

genMetaFunc :: Number → TarConsAlt*
genMetaFunc 0  = ()
genMetaFunc ar = ('|' ('Message' ar) type) ⊕ (genMetaFunc (ar − 1))
                 where type = ('(' 'metatype' (rep ar '→ metatype') ')')
                       rep 0 x = ()
                       rep n x = x ⊕ (repeat (n − 1) x)
```

Method despatchers

The provision of dynamic method despatch requires a mechanism that selects a particular method implementation at run-time from a list of candidates (comprising the method's initial implementation and any overridings in subclasses) according to the actual type of a distinguished object. We wish to avoid code duplication for reused (i.e. inherited but not overridden) implementations and therefore require the mechanism to support many-to-one mappings between a distinguished object type and the appropriate implementation. We also require the method despatchers to support standard FP features, such as curried and partially-applied methods (see Chapter 3 for details of the design of CLOVER's curried message application).

We provide this functionality using explicit despatchers, one for each method. Each despatcher adopts the same arity as the method and performs an explicit indirection to the appropriate implementation, with selection occurring by pattern matching on the type tag of the distinguished object. This allows us to apply despatchers (rather than

the actual method implementations) to arguments, permitting curried dynamic despatchers that can be partially-applied. The complete rule for generating a method despatcher from an IDS method definition is shown below.

For CLOVER examples, see Appendix B.5: (i) the two despatchers in `d_fac` which select the same implementation `i_fac_Number`, and (ii) the two despatchers in `d_minus_1` which select either the implementation `i_minus_1_Number` or `i_minus_1_BoundedNumber`.

Rule for generating a method despatcher

genMethDesp :: $MethDef \rightarrow TarDef^*$

genMethDesp $mDef$ = genMDesp $imps\ mId\ args$

where $args$ = genArgs (arity $mDef$)

genArgs $1 = ()$

genArgs $(n+1) =$ (genArgs n) $\otimes$ ('a' n)

$(mId\ mT\ imps) = mDef$

genMDesp :: $ImpDef^* \rightarrow MethodName \rightarrow TarExpr^* \rightarrow TarDef^*$

genMDesp $()$ mId $args$ = $()$

genMDesp $(i \oplus is)$ mId $args$ = (genMDImp $cIds\ cIdImp\ mId\ args$) $\otimes$

(genMDesp $is\ mId\ args$)

where $(cIdImp \oplus cIdReuses) = cIds$

$(mE\ cIds) = i$

genMDImp :: $ClassName^* \rightarrow ClassName \rightarrow MethodName \rightarrow TarExpr^* \rightarrow$

$TarDef^*$

genMDImp $cIds\ cIdImp\ mId\ args$

= $()$, **if** $cIds = ()$

= $mDesp \oplus$ (genMDImp $cIds'\ cIdImp\ mId\ args$), **otherwise**

where $mDesp = (('$d_$'\ mId)\ parms\ '='\ body)$

$body = (('$i_$'\ mId\ '_'\ cIdImp)\ parms)$

$parms = args \oplus distObj$

$distObj = ('('\ cId\ '$as$'\ ')')$

$(cId \oplus cIds') = cIds$

Wrapped method despatchers

CLOVER is higher-order and we wish to use messages as first-class values. This involves encoding messages in the meta type and also using encoded versions of the method despatchers (which we call *wrapped*). The rule "genMethWrap" takes an IDS method definition and generates a wrapped version of the method despatcher. Generated code always uses and refers to these wrapped versions instead of the original despatchers. For a CLOVER example, see the wrapped method `w_fac` (and its use in `main`) in Appendix B.5.

Rule for generating wrapped method despatchers

$$
\begin{array}{l}
\text{genMethWrap} :: MethDef \rightarrow TarDef \\
\text{genMethWrap}\ \ mDef \ = \ (('\texttt{w\_}'\ mId)\ '='\ body) \\
\qquad\qquad\qquad\qquad \textbf{where}\ \ body = ('('\ ('\texttt{Message}'\ ar)\ ('\texttt{d\_}'\ mId)\ ')') \\
\qquad\qquad\qquad\qquad\qquad\ \ ar = \text{arity}\ \ mDef \\
\qquad\qquad\qquad\qquad\qquad\ \ (mId\ \ mT\ \ imps) = mDef
\end{array}
$$

Method implementations

A key part of the target code is the method implementations themselves. Each method may have several implementations (an initial implementation and overridings in subclasses). For each implementation definition in IDS, we need to know in which class it was first implemented (i.e. the head of the list of class names) but not in which subclasses it was reused (because our method despatchers provide the appropriate many-to-one mappings to implementations).

All λ-abstractions are at the top level of a CLOVER method body (having outlawed the undesirable creation of new methods as local definitions) and are handled by the rule "genMIArgs", which generates appropriate method arguments. The rule "genExpr" is called on a method body and recursively generates corresponding target code for all possible expression types. A message application generates a call to an explicit message applier (which one depends on the the number of arguments to be applied — see Message Appliers later).

CLOVER handles object construction differently, both syntactically and semantically, to message sending. Constructors do not have a distinguished object (and so do not undergo dynamic despatch) and are always applied to their arguments *en masse* (and so we never need manipulate unapplied or partially-applied constructors). The expressive power of CLOVER's message application mechanism is therefore not required for object constructors; they are simply translated into standard function definitions that aggregate their arguments (e.g. `i_new_Account a1 a2 a3 = (Account [a1, a2, a3])` and are invoked by standard function calls. However, constructor definitions for primitive classes (`Number`, `Character` and `Boolean`) are a special case as no aggregation is required (e.g. `i_new_Number a1 = (Number a1)`).

Note that the rules "genPVal" and "genPFunc" merely map primitive values and primitive functions in APC onto their wrapped counterparts in TC. For example, the rule to map a primitive number is: genPVal $('\texttt{NumVal}'\ n) = '(\ '\texttt{Number}'\ n\ ')'$.

The above functionality is provided by the rule "genMethImps", which takes an IDS method definition and generates the target code for each of its implementations. For CLOVER examples, see the method implementations `i_fac_Number`, `i_minus_1_Number` and `i_minus_1_BoundedNumber` in Appendix B.5.

Rule for generating method implementations

$$
\begin{array}{lcl}
\text{genMethImps} :: MethDef \rightarrow TarDef^* & & \\
\text{genMethImps}\ \ (mId\ \ mT\ \ ()) & = & () \\
\text{genMethImps}\ \ (mId\ \ mT\ \ (i \oplus is)) & = & mImp \otimes (\text{genMethImps}\ \ (mId\ \ mT\ \ is)) \\
 & & \textbf{where}\ \ mImp = (fnId\ \ args\ \ '='\ \ body) \\
 & & \qquad fnId = ('\texttt{i\_}'\ \ mId\ \ '\_'\ \ cId) \\
 & & \qquad body = \text{genExpr}\ \ mE' \\
 & & \qquad (mE'\ \ args) = \text{genMIArgs}\ \ mE\ \ () \\
 & & \qquad (mE\ \ (cId \oplus cIds)) = i
\end{array}
$$

```
genMIArgs :: Expr → TarExpr* → TarExpr*
genMIArgs ('Lambda' aId e t) args = genMIArgs e (args ⊗ ('w_' aId))
genMIArgs e                  args = (e args)
```

```
genExpr :: Expr → TarExpr
genExpr ('Object' cId es t)  = ('(' ('i_new_' cId) (map genExpr es) ')')
genExpr ('Variable' vId t)   = ('w_' vId)
genExpr ('Apply' e1 e2 t)    = genApp ('Apply' e1 e2 t)
genExpr ('Let' bs e t)       = ((genExpr e) 'where' (genBinds bs))
genExpr ('Literal' pVal t)   = genPVal pVal
genExpr ('Builtin' pFun t)   = genPFunc pFun
```

```
genApp :: Expr → TarExpr
genApp e
  = (('apply' n) (genExpr e') args)
    where (e' n args) = gA e 0 ()
          gA ('Apply' e1 e2 t) n as = gA e1 (n + 1) ((genExpr e2) ⊕ as)
          gA e                 n as = (e n as)
```

```
genBinds :: Binding* → TarExpr*
genBinds ()           = ()
genBinds ((vId e) ⊕ bs) = lDef ⊕ (genBinds bs)
                          where lDef = (('w_' cId) '=' (genExpr e))
```

Message appliers

Message sending is implemented in standard OOP by applying a method to all its arguments and a distinguished object. We must also support the FP features of curried arguments, partially-applied functions and higher-order functions. We observe that all run-time values, including messages, are wrapped in a meta type; this requires message application to unwrap a message, apply it to arguments and (if a partial application) wrap it again. Standard λ-calculus function application is therefore insufficient. The desired abstract semantics for message application are as follows (where $(M_n\ e)$ is a wrapped message of arity n):

$$\begin{aligned}
\mathcal{E}[\![(M_1\ e^{\sigma_1\to\sigma_2})\ x^{\sigma_x}]\!] &= \mathcal{E}[\![e^{\sigma_1\to\sigma_2}]\!]\ \mathcal{E}[\![x^{\sigma_x}]\!] \\
\mathcal{E}[\![(M_n\ e^{\sigma_1\to\cdots\to\sigma_{n+1}})\ x^{\sigma_x}]\!] &= \mathcal{E}[\![(M_{n-1}\ (e^{\sigma_1\to\cdots\to\sigma_{n+1}}\ x^{\sigma_x}))]\!] \\
&\quad \textbf{where}\ \sigma_x \preceq \sigma_1
\end{aligned}$$

The first equation represents a standard message send: having already been applied to all its message arguments, the method is applied to the distinguished object (its final parameter). The second equation represents a curried partial application, with the result being a wrapped message.

We note that much of the repeated unwrapping and wrapping resulting from a list of curried arguments can be eliminated by applying arguments *en masse* whenever available. This corresponds to the following optimised semantics (where m is the statically-determined number of curried arguments to be applied).

$$\begin{aligned}
\mathcal{E}[\![(M_m\ e^{\sigma_1\to\cdots\to\sigma_{m+1}})\ x_1^{\sigma_{x_1}}\ldots x_m^{\sigma_{x_m}}]\!] &= \mathcal{E}[\![e^{\sigma_1\to\cdots\to\sigma_{m+1}}\ x_1^{\sigma_{x_1}}\ldots x_m^{\sigma_{x_m}}]\!] \\
\mathcal{E}[\![(M_n\ e^{\sigma_1\to\cdots\to\sigma_{n+1}})\ x_1^{\sigma_{x_1}}\ldots x_m^{\sigma_{x_m}}]\!] &= \mathcal{E}[\![(M_{n-m}\ (e^{\sigma_1\to\cdots\to\sigma_{n+1}}\ x_1^{\sigma_{x_1}}\ldots x_m^{\sigma_{x_m}}))]\!] \\
&\quad \textbf{where}\ \sigma_{x_i} \preceq \sigma_i,\ i = 1...m,\ m < n
\end{aligned}$$

Note that, for each message arity, we require a set of appliers — one for each number of arguments to be applied. We use the above semantics to define the following rule to generate explicit message appliers from the maximum method arity. For CLOVER examples, see `apply1` ... `apply3` in Appendix B.5.

Rule for generating message appliers

$$
\begin{array}{l}
\text{genAppliers} :: Number \rightarrow TarDef^* \\
\text{genAppliers} \;\; maxAr \;\; = \;\; \text{genApps} \;\; maxAr \;\; maxAr \\
\\
\text{genApps} :: Number \rightarrow Number \rightarrow TarDef^* \\
\text{genApps} \;\; 0 \;\; 0 \;\; = \;\; () \\
\text{genApps} \;\; n \;\; 0 \;\; = \;\; \text{genApps} \;\; (n-1) \;\; (n-1) \\
\text{genApps} \;\; n \;\; m \;\; = \;\; app \oplus (\text{genApps} \;\; n \;\; (m-1)) \\
\qquad \textbf{where} \;\; app = (('\texttt{apply}' \;\; m) \;\; args \;\; '=' \;\; body) \\
\qquad\qquad args = ('(' \;\; ('\texttt{Message}' \;\; n) \;\; '\texttt{f}' \;\; ')') \oplus xs \\
\qquad\qquad body \;\; = ('\texttt{f}' \;\; xs), \; \textbf{if} \; m = n \\
\qquad\qquad\qquad = ('(' \;\; ('\texttt{Message}' \;\; (n-m)) \;\; ('(' \;\; '\texttt{f}' \;\; xs \;\; ')') \;\; ')'), \\
\qquad\qquad\qquad\quad \textbf{otherwise} \\
\qquad\qquad xs = \text{genXs} \;\; 1 \\
\qquad\qquad \text{genXs} \;\; x = (), \; \textbf{if} \; x > m \\
\qquad\qquad \text{genXs} \;\; x = ('\texttt{a}' \;\; x) \oplus (\text{genXs} \;\; (x+1))), \; \textbf{otherwise}
\end{array}
$$

Other target code components

In this section, we have presented the key components of generated target code. There are several other subsidiary components which have been omitted. These include record operations, input/output and error handling. Note that standard record operations have been presented many times in the literature — illustrative examples include: (i) labelled field selection and update in [AG93]; and (ii) record calculus encoding with extensible tuple values, selectors and updators in [CL91].

4.5 Summary

We have presented the implementation of CLOVER as a translation from OOFP to FP. In doing this, we have demonstrated how to support completely type-safe dynamic despatch in the presence of (partially-applied) higher-order methods. We have also demonstrated how objects and messages may be encoded, and subsumption supported. The CLOVER transformation rules incorporate all of the key features of OOP and are

purely functional. They include a set of high-level translations which resolve method overloading, expand inherited attributes and methods, flatten and invert the code to create an intermediate data structure and, finally, generate target code. The notion of object identity is the only aspect where CLOVER departs from mainstream OOP, as discussed in the next chapter.

Chapter 5

Object-Flow

In this chapter, we discuss the inappropriateness of the traditional OO notion of object identity for OOFP and propose an alternative notion of object identity that could be adopted for OOFP.

There have been many attempts to integrate OOP with FP [BC96], VP with OOP [BGL95], and VP with FP [Hil92]. However, to date we know of no language that integrates OOP and FP with VP, and yet retains the key features of both the functional and object-oriented paradigms. Existing attempts, such as object-oriented dataflow [Kim95], typically sacrifice important features from either OOP or FP. We present the novel visual aspects of *object-flow*, a visual OOFP notation that is purely functional and also object-oriented. The key contributions of this notation include: (i) an application of VP to the integration of OOP and FP, giving a visual representation for OOFP, and (ii) a visual representation of type-safe, curried, higher-order method sending.

5.1 Inappropriateness of Traditional Object Identity

Whilst encapsulation is an extremely important concept for software engineering, it is not clear that mutable internal state is as important. Indeed, it is not clear that object identity should be enforced in OOFP in the same manner as in OOP.

We propose that a finer level of identity should be assumed in OOFP, based on the concept of object behaviour (similar to that used in the Actor model [AH87] for distributed OOP). For example, a digital watch has entirely different behaviour according to whether it has or does not have a new battery installed. These two behaviours can be viewed as different "incarnations" of the watch, and given different names. A watch

may go through several phases of having or not having a fully charged battery, with each incarnation having a separate identity.

In this way, we provide a history of identities which provide immediate "hooks" back into the past. For example, a student might have different option choices in different years: by providing different identities to the different stages of each student, it is possible to ask questions about (send messages to) the different stages of the student's academic study. This also provides obvious benefits for searching algorithms which use backtracking; previous incarnations of an object are immediately accessible. The idea is not new: CLAIRE [CL96] provides versioning for the entire object database, however we provide versioning on a per-object basis.

As a final example of the inappropriateness of the traditional notion of object identity in OOFP, we note that the message-passing view of OOP requires discrete messages to be passed as an atomic action to an object: not only is there no explicit declaration of the behaviour that is expected from that object, but there is also no way in which lazy evaluation could be given meaning in such a system. By contrast, if we view objects as having identities which explicitly change as their behaviour changes, then expected behaviour is made explicit and sequencing of behaviour change is made explicit; this latter change opens the way to the incorporation of lazy evaluation (as shown below).

From the foregoing discussion, it can be seen that we align ourselves with the Actor model used for distributed OOP based on sequences of behaviours rather than state changes. This is also similar to the continuation-passing style often used by FP programmers.

This notion of separating identities by behaviour requires a new object identifier to be created for each change in the internal state of an object, and has two main consequences. Firstly, it affects the utility of the resulting language and, in particular, the patterns of programming that are supported most naturally. Typical functional patterns, such as mathematical algorithms exploiting laziness, are captured well (and also benefit from the addition of OO features). OO patterns exploiting mutable state, such as network simulations, are captured less naturally and require additional "plumbing". Secondly, it affects the execution efficiency by requiring what is essentially copy semantics: to update the state of an object, the old object is copied and given a new state

and identity. Fortunately, it is possible to implement this procedure with low overhead and without name proliferation, as illustrated below.

5.2 Object-Flow: a New Visual Notation

CLOVER provides a visual programming interface. Methods are defined using nodes and arcs to build up a representation of a CLOVER expression.

The choice of a visual notation should be straightforward, yet it is not. This choice is of paramount importance and yet there is no existing suitable notation. Control-flow diagrams are clearly inappropriate for a single-assignment, expression-based language, and *dataflow graphs* [DK82] provide no semantics for OO notions of object identity (with or without behaviours), subsumption, dynamic despatch, etc. A common OO notation is the *object diagram* [Boo94], otherwise known as message-passing or message-flow notation; unfortunately, this notation relies on multiple assignment and does not support the concept of laziness.

The object-oriented message send `o.f(x)` can be represented visually using an object diagram, as in Figure 5.1(a). The functional definition `a = f(x,y)` can be represented visually using a dataflow graph, as in Figure 5.1(b).

(a) Object diagram (b) Dataflow graph

Figure 5.1: Standard Visual Representations

Our goal is to provide a visual notation that integrates the semantics of both the object-oriented and functional representations, despite their apparently-conflicting requirements. In particular, we wish to integrate object identity with referential transparency, and support higher-order methods, curried partial applications and lazy evaluation. Our solution is to use a notation that is almost the dual of object diagram notation, and is similar to Uflow notation [SKA94]. Instead of nodes representing objects and messages flowing along arcs, in our notation the nodes represent the application of methods to their arguments and objects flow along the arcs — this makes the changing state of an

object explicit. When we include higher-order methods, we allow both objects and messages (which may be partial) to flow along the arcs. We call this notation *object-flow*.

We follow CLOVER's approach of extending FP towards OOP, rather than the other way around. This requires us to build upon a referentially transparent dataflow base. We first note that standard dataflow semantics do not provide support for key object-oriented notions such as dynamic despatch. We thus provide extra semantics to facilitate dynamic despatch by identifying the final parameter to be applied as the distinguished object.

Our next step is a notation change so that higher-order methods can be handled naturally. In the traditional functional dataflow model, each node contains a function name, and this function is applied to its incoming arguments. In order to permit the function itself to flow into a node, it is necessary to make each node an *application site* (see Figure 5.2) that receives a method, its arguments and a distinguished object.

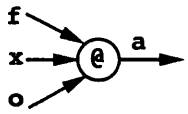


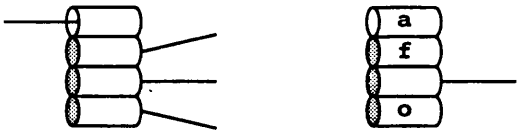
Figure 5.2: Application Site

Object-flow does not permit fan-out; we use aggregate types instead, with explicit selection. This results in an equally expressive and powerful, but less concise, notation. However, a pleasant consequence is that we can eliminate arrows indicating flow direction; we merely work backwards from the result.

The semantics of lazy evaluation are captured and visualised in object-flow by the use of a mechanical *winder* (see Figure 5.3) that "pulls" wires through application nodes from the left. Each method definition contains one winder — the result that is returned. Being demand driven, an object or message is only pulled along an arc (evaluated) if and when it is required. Shared demand for any object will be evaluated by whichever method issues the first demand. All incarnations of an object are preserved for as long as the run-time system can determine that they may be required: as soon as there are no remaining links to an incarnation, it is automatically garbage-collected.

Figure 5.3: Object-Flow Winder

This visual metaphor can be extended with a node represented as a stack of tubes, each tube open at the appropriate end depending on whether it produces or consumes. Object-flow places the result at the top, followed by the method, its arguments and finally the distinguished object to give nodes with structures like Figure 5.4(a). We can also reduce visual clutter due to a plethora of arcs by allowing named values, as in Figure 5.4(b).



(a) Basic node (b) Node with names

Figure 5.4: Object-Flow Nodes

Object-flow is naturally curried — adding another pipe to an application node adds another argument. To represent partial applications, we omit one or more pipes. To aid identification of partial applications, the editor automatically adds an exposed connector to the bottom-most tube, indicating that further pipes are required for full application. For example, we can partially apply + to create the local definition `inc` (see Figure 5.5) which increments a number.

Figure 5.5: Local Definition for `inc`

We can represent the function definition `incList self = map (+ 1) self`, which increments every element in a list by employing the higher-order function `map` to apply `(+ 1)` to each element, as the object-flow method in Figure 5.6.

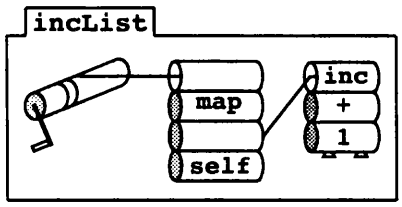


Figure 5.6: Method Definition for `incList`

In object-flow, each arc carries a single atomic object or message, not a stream of objects or messages. Also, recursion is supported by simply naming a method within its object-flow definition.

5.3 Examples of Object-Flow Notation

Figure 5.7 illustrates an `Account` object flowing into a node which represents the application of the `deposit` method to the argument `100.00`. The node comprises a sequence of boxes — the top box represents the result, the box under that represents the method to be applied, the next box represents the parameter and the last box represents the distinguished object. The distinguished object flows into the node from the bottom right and flows out as the result from the top left; this is a new incarnation of the object, which has a different state. Thus, state changes are explicit and object-flow notation provides a timeline for the life of the object. Each stage of the object's life is accessible, providing a versioning feature which supports easy exploration of search spaces through backtracking.

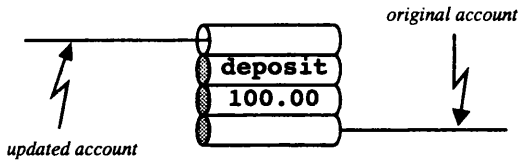


Figure 5.7: Using the Method `deposit`

Figure 5.8 illustrates the definition of the method `deposit`. The arguments (the credit amount and the account) flow in from the right and are given names (the account is called `self` because it is the distinguished object). The method updates[1] `self` with a new balance, which is calculated by adding the credit amount to the existing balance. The result of a node can either be transmitted via an arc or it can be given a name and referenced elsewhere (see `newBalance`).

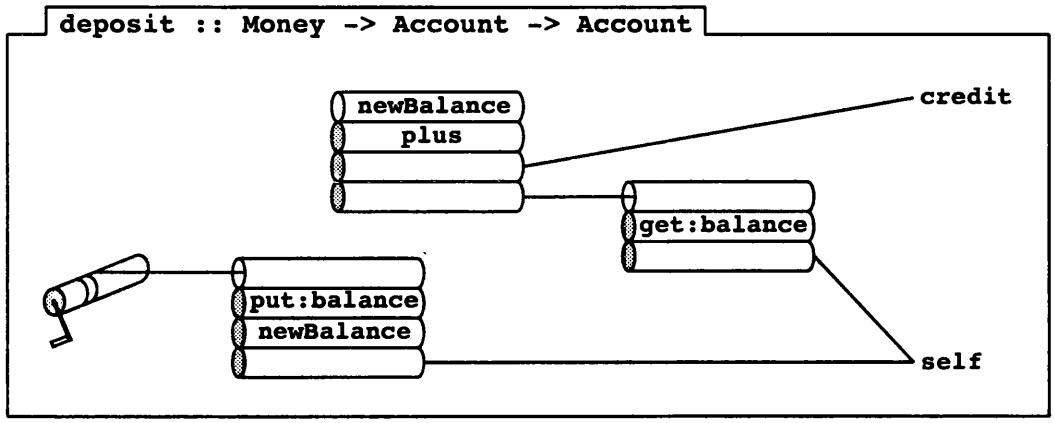


Figure 5.8: Method Definition for `deposit`

Our final examples illustrate method overriding, dynamic despatch and higher-order functions. We assume that the method `charge` is defined in class `Account` and inherited in the two subclasses `CorpAccount` and `PersAccount`. This method calculates the bank charges (of type `Money`) if an account balance is too low. It is overridden in both of the two subclasses to reflect different charging thresholds and charging rates. Figure 5.9 demonstrates how the `charge` method might be used on a list of `Accounts`. The method is passed as a higher-order parameter to `map` (a method of the `List` class), which applies it to every `Account` in the list. This produces a list of `Money`, which is then summed by the `sum` method (also a method of the `List` class). The original list of accounts can include both `CorpAccounts` and `PersAccounts` — the appropriate `charge` method for each is selected using dynamic despatch.

[1] In the implementation, of course, the result is a modified copy of `self`. •

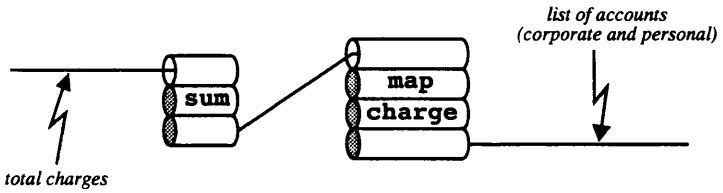
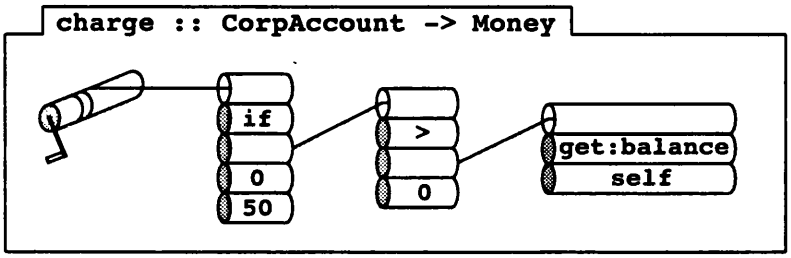
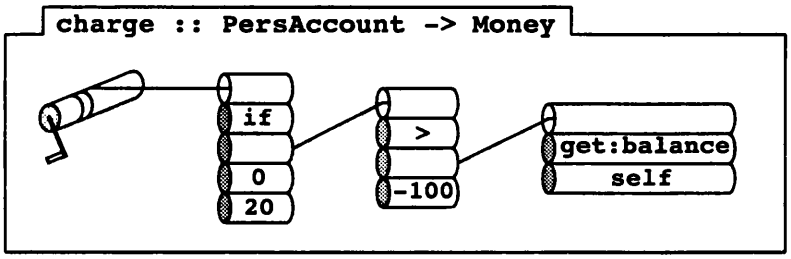


Figure 5.9: Using the Method `charge`

The two overridden definitions of `charge` are illustrated in Figure 5.10(a) and Figure 5.10(b). Note that corporate accounts are charged as soon as their balance is negative, whereas personal accounts are allowed to be 100 units overdrawn before incurring charges.



(a) Definition in class `CorpAccount`



(b) Definition in class `PersAccount`

Figure 5.10: Method Definitions for `charge`

5.4 Development Environment

In our current CLOVER prototype, the development environment consists of a three pane Smalltalk-like front-end comprising: (i) the class hierarchy, (ii) the class attributes and method types, and (iii) a graphical editor for defining methods using object-flow notation.

5.5 Summary

Object-flow is a new visual notation that facilitates the integration of OOP and FP. In particular, it integrates object identity with referential transparency, and supports higher-order methods, curried partial applications and lazy evaluation. The notation's main contribution is its resolution of a core incompatibility between OOP and FP by making object state changes explicit via an alternative notion of object identity.

Chapter 6

Conclusion

In this chapter, we assess the work presented in this thesis, discuss the project status, suggest directions for future work, summarise and conclude.

6.1 Critical Assessment

This thesis presented the design and implementation of a new language, CLOVER, that integrates OOP and FP. The language retains all the key features of FP, such as referential transparency, whilst also retaining all the key features of OOP. This compares favourably with the related languages identified in Chapter 2, each of which resolved the paradigm conflicts by omitting one of more of the problematic features.

However, several limitations have arisen from the design and implementation of CLOVER, including:

- OO patterns of programming which rely heavily on mutable state are not captured concisely by CLOVER's copy semantics and require additional explicit "plumbing";
- In order to ensure complete type safety, the user must provide an explicit type declaration for every method definition and every message send;
- The contravariant rule for subtyping higher-order functions and binary methods is counter-intuitive for OO programmers (because, in these cases, the type of the first argument is anti-monotonic);
- As presented in this thesis, compilation is monolithic. No refinements have been made to facilitate separate compilation (e.g. using a module mechanism).

6.2 Project Status

The CLOVER system comprises a prototype compiler (written in Miranda) to translate from CLOVER concrete syntax to Miranda target code. In addition, a prototype three pane Smalltalk-like browser has been produced to investigate appropriate development environments. Subsequent CLOVER-based projects by students at University College London have included: (i) re-writing the system in Clean and generating Clean target code, (ii) developing a spreadsheet-based front-end, and (iii) enhancing the translator to produce comprehensive error reporting.

6.3 Further Work

The language design for CLOVER is largely complete except for two main considerations: container classes and real-world interfaces. The former are required to support data aggregates such as lists and trees (e.g. via parameterised classes). File input/output, user interaction and event-driven programming is a large area of further work.

We have implemented proofs-of-concept for key CLOVER components, including the translations and type checker presented in this thesis and the visual programming notation. We have yet to supply formal semantics for our visual object-flow notation, although the approach outlined in [Erw97] might be appropriate. We also expect to continue applying the integration of OOP and FP to related areas, such as object-oriented functional spreadsheets [CB97].

There are still some areas where additional functionality is required — for example, the provision of `super` to perform dynamic despatch as if the distinguished object has the type of its superclass. We have not yet provided proofs of referential transparency or soundness and completeness for the type system and the various stages of compilation. Additionally, apart from simple optimisations of the message appliers, little attention has been paid to the performance of executable code.

Further work would also include the full implementation of a subtype checker and the design of an incremental type checker [PM93]. Run-time analyses would be assisted by the addition of algorithm animation and the extension of lexical profiling [CCP95] to visual profiling. Additionally, a revised object-oriented analysis and design notation is required because existing notations assume stateful objects. Alternatively, a transla-

tion from a standard implicit-state notation into explicit-state object-flow notation may prove useful. Finally, extensive usability testing would be required in order to establish meaningful usability results.

6.4 Summary

This thesis began by providing a background of related work in the area of object-oriented functional programming and then established the key research problems by discussing the difficult design issues. We then presented the design of a new language CLOVER (its design features, syntax, type system and abstract expression semantics), which resolved many of these design issues. Next, we presented an implementation of CLOVER as a translation from OOFP to FP. Finally, we discussed the inappropriateness of the traditional notion of object identity for OOFP and proposed an alternative notion, together with a supporting new visual programming notation.

6.5 Conclusions

The goal of this work was to investigate the following hypothesis:

The functional and object-oriented paradigms can be integrated, whilst retaining:

- *higher-order functions, curried partial applications, referential transparency, laziness and complete type safety from the functional paradigm;*
- *subtyping, subsumption, inheritance, method overriding, method overloading and dynamic despatch from the object-oriented paradigm.*

This goal has been achieved and the hypothesis demonstrated in Chapters 3, 4 and 5 by the design and implementation of the CLOVER language. Additionally, this work has contributed to the understanding of the design space of object-oriented and functional programming languages. In particular, we have provided the following contributions:

- a new design for completely type-safe dynamic method despatch and overloading;
- a new object-oriented semantics for partially-applied, higher-order methods;
- a new design for full overloading of methods in the presence of curried partial applications and dynamic despatch;
- a new visual notation and semantics for object state, object identity and object-oriented lazy evaluation.

Appendix A

Type Checking Algorithm

This appendix presents a simple type checking algorithm for CLOVER. The algorithm operates on programs in APC_I form (see Section 4.2.1); we therefore assume that programs have been parsed from their concrete syntax and overloading resolution has also been performed (see Section 4.2.2). Note that full type inference for inclusion polymorphism has long been considered problematic and is likely to be undecidable for CLOVER due to dynamic despatch, although recent work on *soft typing* (e.g. [AW93, BM96]) indicates research progress in this area.

The type checker takes a CLOVER program in APC_I form and returns True if the program is type correct and False if the program is not type correct. It achieves this by first constructing a representation of all types in the class hierarchy and then passing this and the program on to the function "tCheck".

Function for type checking APC_I

```
typeCheck :: Program_APC_I → Boolean
typeCheck apci = tCheck apci cTypes env tContext
                 where cTypes = deriveClassTypes apci
                       env = ( )
                       tContext = 'Invocation'
```

Two key data structures are introduced for use by the type checker: *ClassType* to define the type of each class (via set inclusion in a list of class names), and *TypeContext* to

define the context of a type (for scoping resolution).

Data structures for class types and type contexts

$$
\begin{array}{rcl}
ClassType & ::= & ClassName\ ClassName^{*} \\
\\
TypeContext & ::= & \texttt{'Invocation'} \\
 & | & \texttt{'Self'}\ ClassName
\end{array}
$$

The type of every class can be found by applying the function "deriveClassTypes" to a CLOVER program.

Function for deriving all class types

$$
\begin{array}{l}
\text{deriveClassTypes} :: Program_{APC_I} \rightarrow ClassType^{*} \\
\text{deriveClassTypes}\ apci\ =\ \text{fst}\ (\text{deriveClassTypes'}\ h) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\ (h\ pid) = apc \\
\\
\text{deriveClassTypes'} :: Node \rightarrow (ClassType^{*}\ ClassName^{*}) \\
\text{deriveClassTypes'}\ (cDef\ ns)\ =\ ((cId\ allCIds \oplus types)\ allCIds) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\ allCIds = cId \oplus CIds \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad cIds = \text{concat}\ (\text{map snd}\ subTypes) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad types = \text{concat}\ (\text{map fst}\ subTypes) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad subTypes = \text{map deriveClassTypes'}\ ns \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (CId\ as\ ms) = cDef
\end{array}
$$

The function "tCheck" forms the core of the typechecker and performs case-based selection on expression types. Note that, for applications, the function, the argument and the application itself are checked. For λ-expressions, the body is checked whether well-typed and also whether a subtype of the return type of the overall expression. For Let-expressions, the body of each binding and the main expression are checked whether well-typed and also whether the expression is a subtype of the overall Let-expression.

"tCheck" function for type checking APC_I

```
tCheck :: Program_APC_I → ClassType* → (VariableName Expr)* →
                TypeContext → Boolean
tCheck (h ('Object' cId es t)) cTypes env self  =  True
tCheck (h ('Variable' vId es t)) cTypes env self
  =  (member (map fst env) vId) and (inScope self)
     where  inScope 'Invocation'  =  False
            inScope ('Self' cId)  =  checkScope cId vId h
tCheck (h ('Builtin' pFun t)) cTypes env self  =  True
tCheck (h ('Apply' e1 e2 t)) cTypes env self
  =  (tCheck (h e1) cTypes env' self) and
     (tCheck (h e2) cTypes env self) and
     (subtype (getExprType e2) t1' cTypes) and
     (subtype t2' t cTypes)
     where  env'  =  ('DUMMY' e2) ⊕ env
            ('FunctionType' t1' t2')  =  getExprType e1
tCheck (h ('Lambda' aId e t)) cTypes (('DUMMY' de) ⊕ env) self
  =  (tCheck (h e) cTypes ((aId de) ⊕ env) self) and (f (getExprType e) t)
     where  f t1 ('FunctionType' t1' t2')  =  subtype t1 t2' cTypes
            f t1 t2  =  False
tCheck (h ('Lambda' aId e t)) cTypes env self  =  False
tCheck (h ('Let' bs e t) cTypes env self
  =  check and te and (map f e)
     where  check  =  subtype (getExprType e) t cTypes
            te  =  tCheck (h e) (bs ⊗ env) self
            f (vId ex)  =  tCheck (h ex) (bs ⊗ env) self
```

The function "checkScope" determines whether a given name is a valid attribute or method name for a particular class.

Function for checking name scope

```
checkScope :: ClassName → VariableName → Node → Boolean
checkScope cId vId ((cId as ms) ns)  =  checkScope' vId as ms
checkScope cId vId ((cId' as ms) ns)  =  or (map (checkScope cId vId) ns)

checkScope' :: VariableName → Attribute* → Method* → Boolean
checkScope' vId as ms  =  (member (map hd as) vId) or
                          (member (map hd ms) vId)
```

The function "getExprType" takes an expression and returns its (upper bound) type.

Function for obtaining type of an expression

```
getExprType :: Expr → Type
getExprType ('Object' cId es t)  =  t
getExprType ('Variable' vId t)   =  t
getExprType ('Apply' e1 e2 t)    =  t
getExprType ('Let' bs e t)       =  t
getExprType ('Literal' pVal t)   =  t
getExprType ('Builtin' pFun t)   =  t
```

The function "subtype" determines whether a given type is a subtype of another given type. This function supports function types (via contravariant subtyping) and primitive types.

Function for determining subtyping

```
subtype :: Type → Type → ClassType* → Boolean
subtype ('ClassType' cId) ('FunctionType' t1 t2) cTypes = False
subtype ('FunctionType' t1 t2) ('ClassType' cId) cTypes = False
subtype ('FunctionType' t1 t2) ('PrimitiveType' pType) cTypes = False
subtype ('PrimitiveType' pType) ('FunctionType' t1 t2) cTypes = False
subtype ('ClassType' cId) ('PrimitiveType' 'bool') cTypes
  = subtype ('ClassType' cId) ('ClassType' 'Boolean') cTypes
subtype ('ClassType' cId) ('PrimitiveType' 'char') cTypes
  = subtype ('ClassType' cId) ('ClassType' 'Character') cTypes
subtype ('ClassType' cId) ('PrimitiveType' 'num') cTypes
  = subtype ('ClassType' cId) ('ClassType' 'Number') cTypes
subtype ('PrimitiveType' 'bool') ('ClassType' cId) cTypes
  = subtype ('ClassType' 'Boolean') ('ClassType' cId) cTypes
subtype ('PrimitiveType' 'char') ('ClassType' cId) cTypes
  = subtype ('ClassType' 'Character') ('ClassType' cId) cTypes
subtype ('PrimitiveType' 'num') ('ClassType' cId) cTypes
  = subtype ('ClassType' 'Number') ('ClassType' cId) cTypes
subtype ('ClassType' cId1) ('ClassType' cId2) cTypes
  = subtype' (getClassType cId1) (getClassType cId2)
subtype ('FunctionType' t1 t2) ('FunctionType' t3 t4) cTypes
  = (subtype t3 t1 cTypes) and (subtype t2 t4 cTypes)
subtype t1 t2 cTypes = (t1 = t2)

subtype' :: ClassName* → ClassName* → Boolean
subtype' ()       ()  = True
subtype' ()       ys  = True
subtype' (x ⊕ xs) ys  = (member ys x) and (subtype' xs (remove ys x))

    getClassType :: ClassName → ClassType* → ClassName*
    getClassType cId cTypes = snd (hd (filter ((= cId) . fst) cTypes))
```

Appendix B

Example Translation

This appendix provides an example CLOVER program and the results of its translation from the concrete syntax (defined in Section 3.4), via the various stages of APC (defined in Section 4.2.1) and IDS (defined in Section 4.3.1), into target code (defined in Section 4.4.1).

The example illustrates the use of a class hierarchy (`Number` and subclass `BoundedNumber`), inheritance (method `fac` defined in `Number` and reused in `BoundedNumber`), overloading (two definitions of method `minus` in class `Number`), overriding (definition of method `minus` in subclass `BoundedNumber`), partial application (definition of method `dec` using method `minus`), primitive attributes (`prim` in class `Number`) and primitive methods ('−' in method `minus`).

B.1 Concrete Instance

The following CLOVER program contains a hierarchy and an invocation. The class hierarchy has been simplified to the classes `Object`, `Number`, `BoundedNumber` and `Boolean`. The program invocation evaluates the factorial of 5. Note that the arguments to methods `minus` and `if` (and their built-in counterparts '−' and `if`) are in an unusual order: in the former case, the application '`minus x y`' sends the message '`minus x`' to the distinguished object y and therefore computes 'y - x'; in the latter case, the distinguished object is the value to be tested (thereby placing `if` in the class `Boolean`) and therefore the test appears last in the argument list (for example, '`if 3 4 True`' returns the value 3).

The notation '...' indicates code that has been omitted for clarity.

```
class Object
attributes {}
methods {}
subclasses {

class Number
attributes {
    prim :: num ;
}
methods {
    fac :: Number -> Number {
    fac self
     = let {g = (multiply :: Number -> Number -> Number
                  self :: Number)
                 :: Number -> Number
                 x :: Number
                 :: Number ;
            x = fac :: Number -> Number
                 y :: Number
                 :: Number ;
            y = dec :: Number -> Number
                 self :: Number
                 :: Number ;
            t = (greaterthan :: Number -> Number -> Boolean
                  new Number (0 :: num) :: Number)
                 :: Number -> Boolean
                 self :: Number
                 :: Boolean ;
            } in
            ((if :: Object -> Object -> Boolean -> Object
               g :: Number ):: Object -> Boolean -> Object
              new Number (1 :: num) :: Number)
            :: Boolean -> Object
            t :: Boolean
            :: Object
    }
    minus :: Number -> Number -> Number {
    minus x self
     = new Number (
       (- :: num -> num -> num
         (getprim :: Number -> num
          x :: Number)
         :: num) :: num -> num
         (getprim :: Number -> num
          self :: Number) :: num
        :: num)
       :: Number
```

```
    }
    minus :: Number -> Number {
    minus self
     = new Number (
       (- :: num -> num -> num
         (getprim :: Number -> num
          self :: Number)
         :: num) :: num -> num
         0 :: num) :: num)
       :: Number
    }
    dec :: Number -> Number {
    dec = minus :: Number -> Number -> Number
          new Number (1 :: num) :: Number
          :: Number -> Number
    }
    greaterthan :: Number -> Number -> Boolean { ... }
    lessthan :: Number -> Number -> Boolean { ... }
    multiply :: Number -> Number -> Number { ... }
    getprim :: Number -> num {
    getprim self = prim :: num
    }
}
subclasses {
    class BoundedNumber
    attributes {upperbound :: Number ;
                lowerbound :: Number ;}
    methods {
    minus :: Number -> BoundedNumber -> BoundedNumber {
    minus x self
     = let {result = new BoundedNumber (checkedval :: Number;
                                        ub :: Number;
                                        lb :: Number)
                          :: BoundedNumber ;
            checkedval = ((if :: Number -> Number -> Boolean -> Number
                           lb :: Number)
                          :: Number -> Boolean -> Number
                          (((if :: Number -> Number -> Boolean -> Number
                             ub :: Number)
                            :: Number -> Boolean -> Number
                            tempval :: Number)
                            :: Boolean -> Number
                            uppertest :: Boolean) :: Number
                          lowertest :: Boolean)
                         :: Number ;
            lowertest = (lessthan :: Number -> Number -> Boolean
                         lb :: Number) :: Number -> Boolean
```

```
                    tempval :: Number
                   :: Boolean ;
        uppertest = (greaterthan :: Number -> Number -> Boolean
                    ub :: Number) :: Number -> Boolean
                    tempval :: Number
                   :: Boolean ;
        tempval = new Number (
                  (- :: num -> num -> num
                    (getprim :: Number -> num
                     x :: Number) :: num)
                  :: num -> num
                   (getprim :: BoundedNumber -> num
                    self :: BoundedNumber)
                   :: num :: num) :: Number ;
        lb = (getlowerbound :: BoundedNumber -> Number
              self :: BoundedNumber) :: Number ;
        ub = (getupperbound :: BoundedNumber -> Number
              self :: BoundedNumber) :: Number ;
        } in
        result :: BoundedNumber
    }
    minus :: BoundedNumber -> BoundedNumber { ... }
    multiply :: BoundedNumber -> BoundedNumber -> BoundedNumber { ... }
    getlowerbound :: BoundedNumber -> Number {
    getlowerbound self = lowerbound :: Number
    }
    getupperbound :: BoundedNumber -> Number {
    getupperbound self = upperbound :: Number
    }
    subclasses {}
}

class Boolean
attributes {
    prim :: bool ;
}
methods {
    if :: Object -> Object -> Boolean -> Object { ... }
}
subclasses {}
}

invocation
    fac :: Number -> Number
    new Number (5 :: num) :: Number
    :: Number
```

B.2 APC_{IO} Instance

The previous concrete instance of our example program translates into the following APC_{IO} instance. To reduce complexity, all type information has been omitted.

```
Object () ()
(Number (prim)
 ((fac (Lambda self (Let
    ((g (Apply (Apply (Variable multiply)
        (Variable self)) (Variable x)))
     (x (Apply (Variable fac) (Variable y)))
     (y (Apply (Variable dec) (Variable self)))
     (t (Apply (Apply (Variable greaterthan)
        (Object Number (Literal NumVal 0)))
        (Variable self)))))
    (Apply (Apply (Apply (Variable if) (Variable g))
      (Object Number (Literal NumVal 1))) (Variable t))))
  (minus (Lambda x (Lambda self
          (Object Number (Apply (Apply (Builtin Minus)
           (Apply (Variable getprim) (Variable x)))
            (Apply (Variable getprim) (Variable self)))))))
  (minus (Lambda self (Object Number (Apply (Apply
          (Builtin Minus) (Apply (Variable getprim)
           (Variable self))) (Literal NumVal 0))))))
  (dec (Apply (Variable minus) (Object Number (Literal NumVal 1))))
  (greaterthan ... )
  (lessthan ...)
  (multiply ... )
  (getprim (Lambda self (Variable prim))))
 (BoundedNumber
  (upperbound lowerbound)
  ((minus (Lambda x (Lambda self (Let
     ((result (Object BoundedNumber ((Variable checkedval)
               (Variable ub) (Variable lb))))
      (checkedval (Apply (Apply (Apply (Variable if)
                  (Variable lb)) (Apply (Apply (Apply
                   (Variable if) (Variable ub))
                    (Variable tempval)) (Variable uppertest)))
                     (Variable lowertest)))
      (lowertest (Apply (Apply (Variable lessthan)
                 (Variable lb)) (Variable tempval)))
      (uppertest (Apply (Apply (Variable greaterthan)
                 (Variable ub)) (Variable tempval)))
      (tempval (Object Number (Apply (Apply (Builtin Minus)
                (Apply (Variable getprim) (Variable x)))
                 (Apply (Variable getprim) (Variable self)))))
      (lb (Apply (Variable getlowerbound) (Variable self)))
```

```
      (ub (Apply (Variable getupperbound) (Variable self)))
     (Variable result))))
   (minus ... )
   (multiply ... )
   (getlowerbound (Lambda self (Variable lowerbound)))
   (getupperbound (Lambda self (Variable upperbound))))
  ())
 Boolean (prim) ((if ... )) ())

(Apply (Variable fac) (Object Number (Literal Numval 5)))
```

B.3 APC Instance

The previous APC_{IO} instance of our example program translates into the following APC instance. This illustrates: (i) the resolution of overloaded method `minus` as a result of translation from APC_{IO} to APC_{I}, and (ii) the expansion of inherited attributes and methods in subclass `BoundedNumber` as a result of translation from APC_{I} to APC.

```
Object () ()
(Number (prim)
 ((fac (Lambda self (Let
    ((g (Apply (Apply (Variable multiply)
        (Variable self)) (Variable x)))
     (x (Apply (Variable fac) (Variable y)))
     (y (Apply (Variable dec) (Variable self)))
     (t (Apply (Apply (Variable greaterthan)
        (Object Number (Literal NumVal 0)))
        (Variable self)))))
    (Apply (Apply (Apply (Variable if) (Variable g))
      (Object Number (Literal NumVal 1))) (Variable t))))
  (minus_1 (Lambda x (Lambda self
          (Object Number (Apply (Apply (Builtin Minus)
           (Apply (Variable getprim) (Variable x)))
            (Apply (Variable getprim) (Variable self)))))))
  (minus_2 (Lambda self (Object Number (Apply (Apply
          (Builtin Minus) (Apply (Variable getprim)
           (Variable self))) (Literal NumVal 0))))))
  (dec (Apply (Variable minus_1) (Object Number (Literal NumVal 1))))
  (greaterthan ... )
  (lessthan ...)
  (multiply ... )
  (getprim (Lambda self (Variable prim))))
 (BoundedNumber
  (prim upperbound lowerbound)
  ((fac (Reused Number))
   (minus_1 (Lambda x (Lambda self (Let
     ((result (Object BoundedNumber ((Variable checkedval)
              (Variable ub) (Variable lb))))
      (checkedval (Apply (Apply (Apply (Variable if)
                  (Variable lb)) (Apply (Apply (Apply
                   (Variable if) (Variable ub))
                    (Variable tempval)) (Variable uppertest)))
                     (Variable lowertest)))
      (lowertest (Apply (Apply (Variable lessthan)
                 (Variable lb)) (Variable tempval)))
      (uppertest (Apply (Apply (Variable greaterthan)
                 (Variable ub)) (Variable tempval)))
```

```
      (tempval (Object Number (Apply (Apply (Builtin Minus)
                (Apply (Variable getprim) (Variable x)))
                 (Apply (Variable getprim) (Variable self)))))
      (lb (Apply (Variable getlowerbound) (Variable self)))
      (ub (Apply (Variable getupperbound) (Variable self)))
     (Variable result))))
   (minus_2 ... )
   (dec (Reused Number))
   (greaterthan (Reused Number))
   (lessthan (Reused Number))
   (multiply ... )
   (getprim (Reused Number))
   (getlowerbound (Lambda self (Variable lowerbound)))
   (getupperbound (Lambda self (Variable upperbound)))))
  ())
 Boolean (prim) ((if ... )) ())

(Apply (Variable fac) (Object Number (Literal Numval 5)))
```

B.4 IDS Instance

The previous APC instance of our example program translates into the following IDS instance, illustrating reorganisation of the program structure by flattening the hierarchy and inverting the class-method containment relationship.

```
((Object ())
 (Number (prim))
 (BoundedNumber (prim upperbound lowerbound))
 (Boolean (prim)))
 ((fac (Lambda self (Let
    ((g (Apply (Apply (Variable multiply)
        (Variable self)) (Variable x)))
     (x (Apply (Variable fac) (Variable y)))
     (y (Apply (Variable dec) (Variable self)))
     (t (Apply (Apply (Variable greaterthan)
        (Object Number (Literal NumVal 0)))
        (Variable self)))))
    (Apply (Apply (Apply (Variable if) (Variable g))
      (Object Number (Literal NumVal 1))) (Variable t)))
   (Number BoundedNumber))
  (minus_1 ((Lambda x (Lambda self
           (Object Number (Apply (Apply (Builtin Minus)
            (Apply (Variable getprim) (Variable x)))
             (Apply (Variable getprim) (Variable self))))))
   (Number))
           (Lambda x (Lambda self (Let
     ((result (Object BoundedNumber ((Variable checkedval)
               (Variable ub) (Variable lb))))
      (checkedval (Apply (Apply (Apply (Variable if)
                  (Variable lb)) (Apply (Apply (Apply
                   (Variable if) (Variable ub))
                    (Variable tempval)) (Variable uppertest)))
                     (Variable lowertest)))
      (lowertest (Apply (Apply (Variable lessthan)
                 (Variable lb)) (Variable tempval)))
      (uppertest (Apply (Apply (Variable greaterthan)
                 (Variable ub)) (Variable tempval)))
      (tempval (Object Number (Apply (Apply (Builtin Minus)
                (Apply (Variable getprim) (Variable x)))
                 (Apply (Variable getprim) (Variable self)))))
      (lb (Apply (Variable getlowerbound) (Variable self)))
      (ub (Apply (Variable getupperbound) (Variable self)))
     (Variable result)))
   (BoundedNumber)))
  (minus_2 ((Lambda self (Object Number (Apply (Apply
```

```
          (Builtin Minus) (Apply (Variable getprim)
           (Variable self))) (Literal NumVal 0)))))
   (Number))
          ( ...
   (BoundedNumber)))
  (dec (Apply (Variable minus_1) (Object Number (Literal NumVal 1)))
   (Number BoundedNumber))
  (greaterthan ...
   (Number BoundedNumber))
  (lessthan ...
   (Number BoundedNumber))
  (multiply ( ...
   (Number))
            ( ...
   (BoundedNumber))
  (getprim (Lambda self (Variable prim))
   (Number BoundedNumber))
  (getlowerbound (Lambda self (Variable lowerbound))
   (BoundedNumber))
  (getupperbound (Lambda self (Variable upperbound))
   (BoundedNumber)))
 (if ...
  (Boolean)))

(Apply (Variable fac) (Object Number (Literal Numval 5)))
```

B.5 Target Code Instance

The above IDS instance of our example program translates into the following target code instance, illustrating the generation of a meta type (`metatype`) with class and message types, a program invocation (`main`), a class constructor (`i_new_Number`), a dynamic method despatcher (`d_fac`), a method implementation (`i_fac`) and message appliers (`apply1`, `apply2` and `apply3`).

Note that, in the prototype which targets Miranda code, it is necessary to permute the arguments for the built-in operators '−' and '`if`'.

```
|| Metatype definition

metatype ::= Number num
           | BoundedNumber [metatype]
           | Boolean bool
           | Message3 (metatype->metatype->metatype->metatype)
           | Message2 (metatype->metatype->metatype)
           | Message1 (metatype->metatype)


|| Invocation

main = apply1 w_fac (i_new_Number (Number 5))


|| Constructors

i_new_Number a1 = (Number a1)
i_new_BoundedNumber a1 a2 a3 = (BoundedNumber [a1,a2,a3])
i_new_Boolean a1 = (Boolan a1)


|| Method definitions

w_fac = (Message1 d_fac)
d_fac (Number as) = i_fac_Number (Number as)
d_fac (BoundedNumber as) = i_fac_Number (BoundedNumber as)
i_fac_Number w_self
 = apply3 w_if w_g (i_new_Number 1) w_t
   where w_g = apply2 w_multiply w_self w_x
         w_x = apply1 w_fac w_y
         w_y = apply1 w_dec w_self
         w_t = apply2 w_greaterthan (i_new_Number 0) w_self
```

```
w_minus_1 = (Message2 d_minus_1)
d_minus_1 a1 (Number as) = i_minus_1_Number a1 (Number as)
d_minus_1 a1 (BoundedNumber as)
 = i_minus_1_BoundedNumber a1 (BoundedNumber as)
i_minus_1_Number w_x w_self
 = i_new_Number ((-) (apply1 w_getprim w_x) (apply1 w_getprim w_Self))
i_minus_1_BoundedNumber w_x w_self
 = w_result
   where w_result = i_new_BoundedNumber w_checkedval w_ub w_lb
         w_checkedval = apply3 w_if w_lb
                        (apply3 w_if w_ub w_tempval uppertest)
                        w_lowertest
         w_lowertest = apply2 w_lessthan w_lb w_tempval
         w_uppertest = apply2 w_greaterthan w_ub w_tempval
         w_tempval = i_new_Number((-) (apply1 w_getprim w_x)
                     (apply1 w_getprim w_Self)
         w_lb = apply1 w_getlowerbound w_self
         w_ub = apply1 w_getupperbound w_self

w_minus_2 = (Message1 d_minus_2)
d_minus_2 (Number as) = i_minus_2_Number (Number as)
d_minus_2 (BoundedNumber as) = i_minus_2_BoundedNumber BoundedNumber as)
i_minus_2_Number w_self
 = i_new_Number ((-) (apply1 w_getprim w_self) 0)
i_minus_2_BoundedNumber ...

w_dec = (Message1 d_dec)
d_dec (Number as) = i_dec_Number (Number as)
d_dec (BoundedNumber as) = i_dec_Number (BoundedNumber as)
i_dec_Number w_self = apply1 w_minus_1 (i_new_NUmber 1)

w_greaterthan = (Message2 d_greaterthan)
d_greaterthan a1 (Number as) = i_greaterthan_Number a1 (Number as)
d_greaterthan a1 (BoundedNumber as)
 = i_greaterthan_Number a1 (BoundedNumber as)
i_greaterthan_Number ...

w_lessthan = (Message2 d_lessthan)
d_lessthan a1 (Number as) = i_lessthan_Number a1 (Number as)
d_lessthan a1 (BoundedNumber as)
 = i_lessthan_Number a1 (BoundedNumber as)
i_lessthan_Number ...

w_multiply = (Message2 d_multiply)
d_multiply a1 (Number as) = i_multiply_Number a1 (Number as)
d_multiply a1 (BoundedNumber as)
 = i_multiply_BoundedNumber a1 (BoundedNumber as)
```

```
i_multiply_Number ...
i_multiply_BoundedNumber ...

w_if = (Message3 d_if)
d_if a1 a2 (Boolean as) = i_if_Boolean a1 a2 (Boolean as)
i_if_Boolean ...


|| Labelled field selectors

w_getprim = (Message1 d_getprim)
d_getprim (Number as) = i_getprim_Number (Number as)
d_getprim (BoundedNumber as) = i_getprim_BoundedNumber (BoundedNumber as)
i_getprim_Number w_self
 = prim
   where (Number prim) = w_self
i_getprim_BoundedNumber w_self
 = prim
   where (BoundedNumber [(Number prim),w_upperbound,w_lowerbound]) = w_self

w_getlowerbound = (Message1 d_getlowerbound)
d_getlowerbound (BoundedNumber as)
 = i_getlowerbound_BoundedNumber (BoundedNumber as)
i_getlowerbound_BoundedNumber w_self
 = w_lowerbound
   where (BoundedNumber [(Number prim),w_upperbound,w_lowerbound]) = w_self

w_getupperbound = (Message1 d_getupperbound)
d_getupperbound (BoundedNumber as)
 = i_getupperbound_BoundedNumber (BoundedNumber as)
i_getupperbound_BoundedNumber w_self
 = w_upperbound
   where (BoundedNumber [(Number prim),w_upperbound,w_lowerbound]) = w_self


|| Message appliers

apply3 (Message3 f) a1 a2 a3 = f a1 a2 a3
apply2 (Message3 f) a1 a2 = (Message1 (f a1 a2))
apply2 (Message2 f) a1 a2 = f a1 a2
apply1 (Message3 f) a1 = (Message2 (f a1))
apply1 (Message2 f) a1 = (Message1 (f a1))
apply1 (Message1 f) a1 = f a1
```

Bibliography

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[AF95] A. Aiken and M. Fahndrich. Dynamic Typing and Subtype Inference. *Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 182–191, June 1995.

[AG93] H. Ait-Kaci and J. Garrigue. Label-Selective λ-Calculus, Syntax and Confluence. In *Proceedings of the 13th International Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science 761, pages 24–40, Springer-Verlag, 1993.

[AH87] G. Agha and C. Hewitt. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 47–74, MIT Press, 1987.

[AM90] A. Aiken and B. Murphy. Static Type Inference in a Dynamically Typed Language. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages (POPL'91)*, pages 279–290, January 1991.

[AP93] H. Ait-Kaci and A. Podelski. Towards a Meaning of LIFE. *Journal of Logic Programming*, Volume 16(3&4), pages 195–234, August 1993.

[AS97] K. Achatz and W. Schulte. Functional Object-Oriented Programming with Object-Gofer. In *Informatik '97: Informatik als Innovationsmotor*, pages 552–561, Springer-Verlag, September 1997.

[AW93] A. Aiken and E. Wimmer. Type Inclusion Constraints and Type Inference. Technical Report, IBM Almaden Research Center, 1993.

[AWL94] A. Aiken, E. Wimmers, and T. Lakshman. Soft Typing with Conditional Types. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages (POPL'94)*, pages 163–173, January 1994.

[Bac78] J. Backus. Can Programming be Liberated From the Von Neumann Style? A Functional Style and its Algebra of Programs. *Communications of the ACM (CACM)*, Volume 21(8), pages 613–641, August 1978.

[BB64] E. Berkeley and D. Bobrow, editors. *The Programming Language LISP, its Operation and Applications.* Information International Inc, 1964.

[BC96] L. Braine and C. Clack. Introducing CLOVER: an Object-Oriented Functional Language. In W. Kluge, editor, *Implementation of Functional Languages, 8th International Workshop (IFL'96), Selected Papers*, Lecture Notes in Computer Science 1268, pages 1–20, Springer-Verlag, September 1996.

[BC97] L. Braine and C. Clack. An Object-Oriented Functional Approach to Information Systems Engineering. In *Proceedings of the CAiSE'97 4th Doctoral Consortium on Advanced Information Systems Engineering*, 12 pages, June 1997.

[BC97a] L. Braine and C. Clack. Object-Flow. In *Proceedings of the 13th IEEE Symposium on Visual Languages (VL'97)*, pages 422–423, September 1997.

[BC97b] L. Braine and C. Clack. The CLOVER Rewrite Rules: A Translation from OOFP to FP. In *Draft Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL'97)*, pages 467–488, September 1997.

[BC98] L. Braine and C. Clack. Simulating an Object-Oriented Financial System in a Functional Language. In *Draft Proceedings of the 10th International*

Workshop on Implementation of Functional Languages (IFL'98), pages 487–496, September 1998.

[BDG+88] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common Lisp Object System Specification. *SIGPLAN Notices*, 23(Special Issue), September 1988.

[Ber92] E. Berger. FP + OOP = Haskell. Technical Report, Department of Computer Science, University of Texas at Austin, March 1992.

[BGL95] M. Burnett, A. Goldberg, and T.Lewis, editors. *Visual Object-Oriented Programming: Concepts and Environments*. Manning Publications, 1995.

[BGW91] D. Bobrow, R. Gabriel, and J. White. CLOS: Integrating Object-Oriented and Functional Programming. *Communications of the ACM (CACM)*, 34(9):28–38, September 1991.

[BKK+86] D. Bobrow, K. Khan, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86)*, SIGPLAN Notices 21(11), pages 17–29, November 1986. Reprinted in A. Cardenas and D. McLead, editors, *Research Foundations in Object-Oriented and Semantic Database Systems*, pages 70–90, Prentice-Hall, 1990.

[BM96] F. Bourdoncle and S. Merz. Primitive subtyping $\wedge$ implicit polymorphism $\models$ object-orientation. Presented at *Third International Workshop on Foundations of Object-Oriented Languages (FOOL 3)*, July 1996.

[BM97] F. Bourdoncle and S. Merz. Type checking higher-order polymorphic multi-methods. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 302–315, January 1997.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*, Second Edition. Benjamin-Cummings, 1994.

[Bud95] T. Budd. *Multiparadigm Programming in Leda.* Addison-Wesley, 1995.

[Can82] H. Cannon. *Flavors: A non-hierarchical approach to object-oriented programming.* Symbolics, Inc., 1982.

[Car88] L. Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, Volume 76(2/3), pages 138–164, February/March 1988. First appeared in *Proceedings of the International Symposium on Semantics of Data Types*, pages 51–67, 1984.

[CB97] C. Clack and L. Braine. Object-Oriented Functional Spreadsheets. In *Proceedings of the 10th Glasgow Workshop on Functional Programming (GlaFP'97)*, 12 pages, September 1997.

[CCP95] C. Clack, S. Clayman, and D. Parrott. Lexical Profiling — Theory and Practice. *Journal of Functional Programming*, Volume 5(2), pages 225–277, 1995.

[CL91] L. Cardelli and G. Longo. A Semantic Basis for Quest. *Journal of Functional Programming*, Volume 1(4), pages 417–458, 1991.

[CL96] Y. Caseau and F. Laburthe. Introduction to the CLAIRE Programming Language. Technical Report, Department Mathematiques et Informatique, Ecole Normale Superieure, Paris, April 1996.

[CSK+97] J. Chang, J. Song, J. Kim, H. Kim, and S. Han. Implementation of an Object-Oriented Functional Language on the Multithreaded Architecture. In *Proceedings of the 1997 International Conference on Parallel and Distributed Systems*, pages 294–301, December 1997.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, Volume 17(4), pages 471–522, 1985.

[Dil88] A. Diller. *Compiling Functional Languages.* John Wiley & Sons, 1988.

[DK82] A. Davis and R. Keller. DataFlow Program Graph. In *IEEE Computer*, Volume 15(2), page 26–41, 1982.

[DK94] S. Drossopoulou and S. Karathanos. ST&T: Smalltalk with Types. Technical Report DOC 94/11, Department of Computing, Imperial College, London, July 1994.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages (POPL'82)*, pages 207–212, January 1982.

[DV96] L. Dami and J. Vitek. Introduction to HOP, a Functional and Object-Oriented Language. Submitted for publication, 1996. Available from http://cuisun9.unige.ch/d̃ami/Hop/

[Erw97] M. Erwig. Semantics of Visual Languages. In *Proceedings of the 13th IEEE Symposium on Visual Languages (VL'97)*, pages 300–308, September 1997.

[FHM94] K. Fisher, F. Honsell, and J. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, Volume 1(1), pages 3–37, Spring 1994. First appeared in *Proceedings of the Eighth IEEE Symposium on Logic in Computer Science*, pages 26–38, June 1993.

[FM95] K. Fisher and J. Mitchell. The Development of Type Systems for Object-Oriented Languages. *Theory and Practice of Object Systems*, Volume 1(3), pages 189–220, 1995.

[GJL87] D. Gelernter, S. Jagannathan, and T. London. Environments as First Class Objects. In *Proceedings of the Fourteenth ACM Symposium on Principles of Programming Languages (POPL'87)*, pages 98–100, January 1987.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[GWM+93] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. Tutorial and Manual, Computing Laboratory, Oxford University, October 1993.

[Hil92] D. Hils. Visual Languages and Computing Survey: Data Flow Visual Programming Languages. *Journal of Visual Languages and Computing*, Volume 3(1), pages 69–101, 1992.

[HL91] F. Henglein and K. Laufer. Programming with Structures, Functions, and Objects. In *Proceedings of the 17th Latin American Informatics Conference (PANEL'91)*, pages 333-352, July 1991.

[Hol90] C. Holt. viz: A Visual Language Based on Functions. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 221–226, 1990.

[Jon94] M. Jones. The Implementation of the Gofer Functional Programming System. Research Report YALEU/DCS/RR-1030, Yale University, New Haven, May 1994.

[Kim95] T. Kimura. Object-Oriented Dataflow. In *Proceedings of the 11th IEEE Symposium on Visual Languages (VL'95)*, pages 180–186, September 1995.

[Lan64] P. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, Volume 6, pages 308–320, 1964.

[LBF+91] D. Lau-Kee, A. Billyard, R. Faichney et al. VPL: An Active, Declarative Visual Programming System. In *Proceedgings of the 1991 IEEE Workshop on Visual Languages*, pages 40–46, 1991.

[Ler95] X. Leroy. The Caml Special Light System: Modules and Efficient Compilation for Caml. Research Report 2721, Institut National de Recherche en Informatique et Automatique (INRIA), 21 pages, November 1995.

[MCB90] M. Mannino, I. Choi, and D. Batory. The Object-Oriented Functional Data Language. In *IEEE Transactions on Software Engineering*, Volume 16(11), pages 1258–1272, November 1990.

[McC97] C. McClure. *Software Reuse Techniques*. Prentice-Hall, 1997.

[Mey91] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1991.

[MHH91] W. Mugridge, J. Hammer, and J. Hosking. Multi-Methods in a Statically-Typed Programming Language. In P. America, editor, *Proceedings of the Fifth European Conference on Object-Oriented Programming (ECOOP'91)*, Lecture Notes in Computer Science 512, pages 307–324, Springer-Verlag, 1991.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer System Science*, Volume 17, pages 348–375, 1978.

[MMM91] J. Mitchell, S. Meldal, and N. Madhav. An Extension of Standard ML Modules with Subtyping and Inheritance. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages (POPL'91)*, pages 270–278, January 1991.

[MMR95] M. Muller, T. Muller, and P. Roy. Multiparadigm Programming in Oz. In D. Smith, O. Ridoux and P. Roy, editors, *Visions for the Future of Logic Programming: Laying the Foundations for a Modern Successor of Prolog"*, Workshop in Association with ILPS'95, December 1995.

[Moo86] D. Moon. Object-Oriented Programming with Flavors. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86)*, SIGPLAN Notices 21(11), pages 1–8, November 1986.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*, MIT Press, 1990.

[MYH+84] N. Monden, I. Yoshimoto, M. Hirakawa et al. HI-VISUAL: A Language Supporting Visual Interaction in Programming. In *Proceedings of the 1984 IEEE Workshop on Visual Languages*, pages 199–205, 1984.

[NA92] R. Nikhil and Arvind. Id: a Language with Implicit Parallelism. In J. Feo, editor, *A Comparative Study of Parallel Programming Languages: The Salishan Problems*, pages 169–215, Elsevier Science Publishers, 1992.

[Oor96] W. Oortmerssen. The Bla Language: Extending Functional Programming with First Class Environments. Masters thesis, Department of Computational Linguistics, University of Amsterdam, 1996.

[Pey87] S. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[PHA+97] J. Peterson, K. Hammond, L. Augustsson, B. Boutel, W. Burton, J. Fasel, A. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, E. Meijer, S. Peyton-Jones, A. Reid, and P. Wadler. *Report on the Programming Language Haskell. A Non-strict, Purely Functional Language*, Version 1.4, April 1997. Available from http://haskell.org/report/

[PM93] J. Poswig and C. Moraga. Incremental Type Systems and Implicit Parametric Overloading in Visual Languages. In *Proceedings of the 9th IEEE Symposium on Visual Languages (VL'93)*, pages 126–133, August 1993.

[PVM95] J. Poswig, G. Vrankar, and C. Moraga. VisaVis: a Higher-order Functional Visual Programming Language. *Journal of Visual Languages and Computing*, Volume 5, pages 83–111, 1995.

[QM97] Z. Qian and B. Moulah. Combining Object-Oriented and Functional Language Concepts. Department of Mathematics and Computer Science, University of Bremen, available from http://www.informatik.uni-bremen.de/~qian/abs-comcon.html, August 1997.

[RR96] J. Reppy and J. Riecke. Simple Objects for Standard ML. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI)*, SIGPLAN Notices 31(5), pages 171–180, May 1996.

[RV97] D. Remy and J. Vouillon. Objective ML: A Simple Object-Oriented Extension of ML. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 40–53, January 1997.

[RW91] J. Rasure and C. Williams. An Integrated Data Flow Visual Language and Software Development Environment. *Journal of Visual Languages and Computing*, Volume 2, pages 217–246, 1991.

[Sar93] J. Sargeant. Uniting Functional and Object-Oriented Programming. In S. Nishio and A. Yonezawa, editors, *Object Technologies for Advanced Software (ISOTAS'93)*, Lecture Notes in Computer Science 742, pages 1–26, Springer-Verlag, November 1993.

[SKA94] J. Sargeant, C. Kirkham, and S. Anderson. The Uflow Computational Model and Intermediate Format. Technical Report UMCS-94-5-1, Department of Computer Science, University of Manchester, May 1994.

[Soc93] A. Socorro. Design, implementation and evaluation of a declarative object-oriented programming language. DPhil thesis, Computing Laboratory, University of Oxford, Trinity Term 1993.

[SPL89] M. Szpakowski, T. Pietrzykowski, J. Laskey et al. *Prograph Reference: A very high-level, pictorial, object-oriented programming environment.* TGS Systems, 1989.

[Ste84] G. Steele, Jr. *Common LISP: The Language.* Digital Press, 1984.

[Sto85] W. Stoye. The Implementation of Functional Languages using Custom Hardware. PhD thesis, Computer Laboratory, University of Cambridge, May 1985.

[Tur85] D. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the 2nd Conference on Functional Programming Languages and Computer Architecture (FPCA'85)*, Lecture Notes in Computer Science 201, pages 1–16, Springer-Verlag, 1985.

[Tur85a] D. Turner. Functional Programs as Executable Specifications. In C. Hoare and J. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 29–54, Prentice-Hall, 1985.

[WB89] P. Wadler and S. Blott. How to Make ad-hoc Polymorphism Less ad-hoc. In *Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages (POPL'89)*, pages 60–76, January 1989.

[WR13] A. Whitehead and B. Russell. *Principia Mathematica*. Cambridge, 1913.