A Formal Semantics for C++

Michael Norrish
Michael.Norrish@nicta.com.au
(Version 293)

Abstract

This document describes a formal operational model of the dynamic semantics of much of the C++ language (as specified in the ISO Standard [5]). The formal model was developed in the HOL theorem-prover, providing additional guarantees as to its good sense. This report presents and explains extracts from the mechanised source-code that was fed to HOL.

This work was performed under funding from QinetiQ's Systems Assurance Group under the UK MOD Output 3a research project entitled *Robust Languages*.

Contents

1 Introduction **3**
1.1 Reading HOL Source Code 4
1.2 Understanding C++ in Three Phases 12

2 C++'s Basic Types **13**
2.1 Bytes & Memory 15
2.2 Hierarchical Environments 17

3 Phase 1: Name Resolution **19**
3.1 The Name Resolution State 19
3.2 Template Names 21
3.3 Resolving Expressions 21
3.4 Resolving Declarations: Establishing Phase 1 Names 25

4 Phase 2: Templates **28**
4.1 Instantiation and Matching 31
4.2 Program Instantiation 31

5 Phase 3: Dynamics **36**
5.1 Dynamic States . . . 37
5.2 The Dynamic Relation . . . 37
5.3 Special Syntactic Forms . . . 39
5.4 Simple Expression Rules . . . 40
5.5 Statements in a Small-step Style . . . 51
5.6 Exceptions . . . 64
5.7 Basic Object-Orientation . . . 72
5.8 Reference Types . . . 81
5.9 Polymorphism & Multiple Inheritance . . . 86
5.10 Object Lifetimes . . . 91
5.11 Class R-Values . . . 100

6 Validation **110**

7 Omissions and Possible Fixes **114**

A Mechanised Sources **115**
A.1 Building HOL Source-Files . . . 115

B Annotated Bibliography **117**

List of Figures

1 A Program Requiring Name Resolution . . . 19
2 Name Resolution in Function Applications . . . 22
3 Name Resolution for Variables . . . 24
4 Defining a New Global Variable in Phase 1 . . . 25
5 Name Resolution for a Global Class Declaration . . . 27
6 Name Resolution for a Global Function Definition . . . 29
7 A Difficult Program for Template Interpretation . . . 30
8 A Template Program . . . 32
9 A Template Program that Breaks Siek and Taha's Model . . . 33
10 Finding the Best Function Instantiation . . . 36
11 The HOL Type of Dynamic State . . . 38
12 Rules for the Standard Binary Operators . . . 44
13 Rules for the Standard Unary Operators . . . 44
14 Rules for "C-style" Dereferencing of Pointers . . . 47
15 Taking the Address of a Non-static Member . . . 48
16 Dereferencing a Pointer-to-Member . . . 49

17 Rule for the Completion of an Assignment Expression 51
18 Where Functions Must Not Interleave 52
19 Making a Function Call 53
20 Statement Rules for Expression and If Statements 57
21 Dynamic Rules for Loop Interruptions and Traps 58
22 Exiting a Block 61
23 Finishing Initialization of a Non-class, Non-reference Variable 64
24 Catching a Typed Exception 68
25 C++ Code Demonstrating OO-Polymorphism 74
26 The Rule for Virtual Member Function Dispatch 75
27 Selecting a Non-static Member Function 77
28 Calculating the Offset of a Non-static Data Member 78
29 The `nonclass_conversion` Relation 80
30 Performing a Polymorphic `dynamic_cast` 81
31 How References are Initialized 86
32 Multiple Inheritance with Shared Base Objects 88
33 A Lop-sided 'V' Inheritance Hierarchy 90
34 Making a Call to a Class Constructor 95
35 C++ Constructors with *mem-initializers* 97
36 Variable Declarations Generated from Figure 35 98
37 Allocating a Non-Class Object on the Heap with `new` 100
38 Allocating a Class Object on the Heap 101
39 Allocating Memory for Class R-Values 103
40 The Abstract Syntax Corresponding to Test Program `t6` 113

1 Introduction

This document presents the substance of the mechanised C++ semantics that is developed in the accompanying HOL source files (see Appendix A). Those files sum to over 12 000 lines; this document tries to cover both the important parts in detail, and to describe the less important parts at a high level.

The HOL mechanisation itself is necessarily the only *formal* part of the deliverable. This document quotes from the sources liberally, and aims to make these relatively easily understood by accompanying HOL extracts with English prose. Where prose and rule appear to conflict, this will almost certainly reflect a problem with the prose and not the rule. The rule has been type-checked, and in some cases, will have also been validated to some extent.

In addition to the ISO Standard itself [5], the report draws on a number of other sources. The annotated bibliography in Appendix B describes all of these. The remainder of this introduction describes how to read the HOL syntax that appears throughout the report (in Section 1.1), and then gives a high-level picture of the C++ semantics in Section 1.2.

Section 2 describes the basic types that underlie the semantics. Here, for example, are discussions of how C++'s hierarchical namespaces are modelled with a custom data type in HOL. Sections 3–5 describe each of the three phases identified in Section 1.2 in more detail. Section 6 describes how the semantics has been and might be validated to increase confidence in its correctness. Finally, Section 7 describes the omissions and faults in the semantics.

This document supersedes all previous deliverables.

1.1 Reading HOL Source Code

This report contains a large number of extracts from the HOL source code found in the `holsrcs` directory of the deliverable. HOL's Description manual [4] has full details for the following, which is a brief summary of HOL syntax and semantics.

Where quotations are made from underlying HOL sources, the origin of the quotation will be identified using the form HOL:`thyname`. The source code for theory `thyname` is available in the file

```
holsrcs/thynameScript.sml
```

When compiled, the corresponding theory will have a readable signature in the file

```
holsrcs/thynameTheory.sig
```

1.1.1 HOL Syntax

HOL is a powerful logical language, equipped with the usual features of predicate logic (quantifiers such as $\forall$ and propositional connectives such as $\vee$ and $\Rightarrow$), as well as a rich, typed term language. HOL's types can be simple, as with the types `:bool`, that of boolean values "true" and "false", and `:num`, the type of natural numbers. HOL also includes type-operators that take other types as arguments in order to create fresh types. For example, the function arrow -> creates the type of functions between two existing types. In this way,

```
:num -> bool
```

is the type of functions from natural numbers to booleans. Another type operator (`list`) allows for the type of lists of elements drawn from a particular type. Finally, in higher-order logic, unlike in predicate logic, there is no distinction between terms and formulas. The latter are simply terms of type `:bool`.

Terms in HOL come in four forms: variables, constants (defined by the user, or provided by the system), function applications and abstractions. Function applications are usually written without parentheses around the argument, so that one will see `f x` (an application of function `f` to argument `x`) rather than `f(x)`. Of course, sometimes the nature of the values will require parentheses: `f x + y` could be parenthesised `(f x) + y`, which is not the same as `f(x + y)`.

Functions are often also "curried", meaning that instead of `f(x,y)`, one will often see `f x y` (though the former is still possible). In `f(x,y)`, the type of `f` (a function) will be

```
:xty # yty -> result_ty
```

(where `#` is the Cartesian product type operator). In the second term, the type of `f` is

```
:xty -> (yty -> result_ty)
```

Abstractions are λ-expressions, forming functions. For example, the term

```
λx. x + y
```

is the function which takes an argument `x` and returns the result of adding `x` to the value `y`. The type of this function will be

```
:num -> num
```

Two further derived forms occur frequently in the C++ model: `let` and conditional expressions. A term of the form

`let` pat `=` exp_1 `in` exp_2

evaluates exp_2 in a context where the names occurring in pat are bound to the value given by exp_1. For example,

```
let x = 3 * 10 in x + 6
```

has value 36. The use of `let` expressions serves only to structure the source, and is useful for readability. By way of contrast, in a programming language like SML, where there are side effects, the use of `let` can make significant semantic differences.

Conditional expressions are written

`if` g `then` e_1 `else` e_2

where g must be of type `:bool`.

1.1.2 Standard HOL Types

Numbers HOL has three different numeric types: natural numbers (`:num`), integers (`:int`) and real numbers (`:real`). The latter are not used in the C++ formalisation because there is no mechanisation of the language's floating point numbers. All types support the standard arithmetic operators, such as addition (`+`), subtraction (`-`) and multiplication (`*`). In addition, HOL supports types of fixed-width memory "words".

Pairs The type of pairs of σs and τs is written `:`σ `#` τ. Pair values are written inside parentheses, with components separated by commas, *e.g.*, `(1,4)`. The function `FST` returns the first component of a tuple, and `SND` returns the second.

Lists The type of lists of elements drawn from a type α is written `:`α `list` (*i.e.*, the type operator appears as a suffix). Lists can either be empty (`[]`) or the result of "cons"-ing an element onto the front of an existing list. The list consisting of element `h` followed by list `t` is written `h::t`. A literal list of a fixed number of elements can also be written between square brackets, with successive elements separated by semi-colons. Thus,

```
[1; 2; 3]
```

is an alternative to writing

```
1 :: 2 :: 3 :: []
```

Lists come equipped with various standard functional language operations such as

```
++    : 'a list -> 'a list -> 'a list
CONS  : 'a -> 'a list -> 'a list
EL    : num -> 'a list -> 'a
FOLDL : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
HD    : 'a list -> 'a
LAST  : 'a list -> 'a
MAP   : ('a -> 'b) -> 'a list -> 'b list
MEM   : 'a -> 'a list -> bool
TL    : 'a list -> 'a list
ZIP   : 'a list -> 'b list -> ('a # 'b) list
```

The list append operation (++) is written as an infix.

Options The option type has two constructors

```
NONE : 'a option
SOME : 'a -> 'a option
```

allowing one to encode a type that represents all of the values of an existing type, and an extra value, usually representing failure or absence. The `THE` function reverses the action of `SOME`. If applied to `NONE`, its value is unspecified.

Finite Maps The finite map is a form of function where the domain is guaranteed to be finite. The type of finite maps from σ to τ is written `:`σ `|->` τ. The HOL syntax for the application of a finite map `fm` to an element `x`, is `fm ' x`, with an apostrophe separating function and argument. (Note that the result of the application is unspecified if the argument is not in the map's domain.)

The empty finite map (one with no domain) is `FEMPTY`, and updating a map `fm` so that its value at `k` is `v` (ignoring whether or not `fm` may have already had a value for `k`), is written `fm |+ (k,v)`. The function `FDOM` returns the domain of a finite map.

Records Record literals are written as a sequence of assignment to field-names in between ASCII angle brackets, thus:

```
<| fld1 := 3; fld2 := 5; fld3 := (F,[]) |>
```

New record values can be constructed from old ones with the `with` keyword. For example

```
rec with <| fld2 := 10; fld6 := T |>
```

is a record value that is everywhere identical to `rec` except that its fields `fld2` and `fld6` have different values.

When updating records, in addition to the assignment syntax (`:=`), one can also use `updated_by`, which allows a function to be applied to the old value of the field. So,

```
rec with <| fld2 updated_by SUC; fld6 := T |>
```

is a record value everywhere the same as `rec`, except that its `fld6` is true, and its `fld2` is one bigger.

1.1.3 Definitions in HOL

There are three main forms of definition used in HOL: definition of new algebraic types, definition of functions specified equationally, and inductive definition of relations.

Defining Types Algebraic types naturally correspond to the syntax of formal languages. They are defined in a style that resembles `datatype` declarations in functional languages such as Haskell or SML. For example, the definition of the type for a small language of arithmetic expressions might be:

```
ArithExp = Number of int
         | Variable of string
         | Plus of ArithExp # ArithExp
         | Times of ArithExp # ArithExp
         | FnCall of string # ArithExp list
```

In HOL's data type definitions, the string to the left of the `=` (`ArithExp` here) is the name of the new type. After the equals sign, there is then a list of different possible forms for the values inhabiting the type. In the example, we assert that expressions come in five different forms. Each form is optionally accompanied by data (following the `of` keyword). If the arithmetic expression is an addition, for example, then it consists of (recursively) two arithmetic expressions representing the left and right arguments (the `#` symbol is the type operator for Cartesian products).

As a result of a definition like the above, the new type is established, and five new constants (known as the new type's *constructors*) are defined, with types

```
Number   : int -> ArithExp
Variable : string -> ArithExp
Plus     : ArithExp # ArithExp -> ArithExp
Times    : ArithExp # ArithExp -> ArithExp
FnCall   : string # ArithExp list -> ArithExp
```

Thus, the new type `ArithExp` is inhabited by values such as `Number 5`, `Variable "i"`, and more complicatedly:

```
Plus (Number "3", FnCall ("sin", [Variable "pi"]))
```

Finally, in order to support the common, "curried" style of functional programming, one can define multiple arguments to constructors by separating them with the special symbol `=>`. If one were to write

```
SimpExp = Number of int
        | Plus of SimpExp => SimpExp
```

the new type `SimpExp`'s second constructor `Plus` would have type

```
SimpExp -> SimpExp -> SimpExp
```

So, rather than taking its two arguments as a pair (as in the first `ArithExp` example above), the `Plus` constructor for `SimpExp` takes its two arguments one after the other.

When a type has been defined in this way, an expression can be analysed with a "case expression" that allows pattern-matching. For example, one might write an evaluator for `SimpExp` values:

```
eval e  =  case e of
              Number i -> i
           || Plus e1 e2 -> eval e1 + eval e2
```

Successive cases are separated by the `||` symbol, and the pattern is separated from the body by the arrow `->`. This is very similar to what is possible in functional programming languages such as SML and Haskell.

Defining Functions In HOL, definitions of functions look similar to function definitions in functional programming languages, with features such as pattern-matching and recursion common. For example, the following is the definition of a factorial function

```
(FACT 0 = 1) ∧
(FACT (SUC n) = SUC n * FACT n)
```

The individual equations are separated by conjunction symbols because the text of the definition becomes the statement of the theorem characterising the behaviour of the new constant. (If the recursion is not well-founded, or if there is some other error in the quoted text, then HOL does not allow the definition to proceed, and no theorem is produced.)

Defining Relations Inductive relations allow the definition of systems of rules, common in the definitions of operational semantics. For example, one might write a big-step rule for applicative order reduction in the λ-calculus as

$$\frac{\Gamma \vdash M \Downarrow (\lambda x.M_0) \qquad \Gamma \vdash N \Downarrow v_0 \qquad (x \mapsto v_0), \Gamma \vdash M_0 \Downarrow v}{\Gamma \vdash M\,N \Downarrow v}$$

This should be read as stating that an application $M\,N$ reduces to a value v if M reduces to an abstraction $(\lambda x.M_0)$, if N reduces to some value v_0, and if the body M_0 reduces to v, while the bound variable x is linked to v_0.

The same rule might be presented in HOL syntax as

```
    apeval G M (LAM x M0) ∧
    apeval G N v0 ∧
    apeval ((x,v0) :: G) M0 v
  ⇒
    apeval G (APP M N) v
```

The use of the conjunction and implication symbols makes the propositional structure of the rule explicit. Also, the relation being defined is always the first symbol (`apeval` here) of the conclusion. Note also how the commonly used curried style is used to define `apeval`.

Another example is the rule *[var-to-fvalue]* from the dynamic semantics for C++:

```
(* RULE-ID: var-to-fvalue *)
    (lookup_type s vname = SOME ty) ∧
    function_type ty ∧
    vname ∈ FDOM s.fnencode
  ⇒
    mng (s, EX (Var vname) se) (s, EX (FVal vname ty NONE) se)
```

(Aside) A "pretty", more "mathematical", version of the same rule might appear

$$\frac{\mathsf{lookup_type}(s, v) = \lfloor \tau \rfloor \qquad \mathsf{function_type}(\tau) \qquad v \in \mathsf{dom}(s.\mathsf{fnencode})}{(s, \mathsf{EX}(\mathsf{Var}(v), se)) \;\rightarrow\; (s, \mathsf{EX}(\mathsf{FVal}(v, \tau, *), se))}$$

Here I have (arbitrarily) chosen to represent the `mng` relation with an arrow ($\rightarrow$), to use Greek for the type variable (`ty` *vs* τ), and to use $\lfloor _ \rfloor$ and $*$ to represent `SOME` and `NONE` respectively. I have made EX and FVAL take tuples of arguments, though the original `EX` and `FVAL` take curried arguments, and I have introduced various LaTeX spacing commands, all in an attempt to lay the rule out in an aesthetically pleasing way. Unfortunately, this sort of effort does not scale to very large rules. It is also entirely manual (at least in HOL4; contrast Isabelle's support for automatic generation of "reasonable" LaTeX). The Ott tool [11] might also represent a way to solve this problem. If a translation is manual, it is inherently error-prone. *(End Aside)*

In the ASCII rendering, we see that the rule has three premises (separated by $\wedge$), and that its conclusion defines the constant `mng`. The inductive relation `mng` is similar to the reduction relation for the λ-calculus. That was a ternary relation describing how one language form (the second argument) might evolve into another (the third argument), all in a context Γ.

Here, there is no separate context, so `mng` is just a binary relation, but like `apeval` it is also defining the evolution of language forms. The first argument is a pair of a state `s` and an expression, and this can be seen as the "starting point". The second argument is another pair, and it is the result of the execution step.

There are two important things to note about rules of this form.

- Rules do not *require* a step to take place. Rather they state that a step is *permitted*. If another rule describes another possible behaviour for the same form, then that second behaviour is also permitted, and the rules collectively describe a non-deterministic system.

- Rules' premises represent preconditions on the behaviour. If the preconditions are not satisfied then the rule can not "fire", and the behaviour may not occur. Of course, another rule may allow some other behaviour to occur for the same starting point.

The three premises in *[var-to-fvalue]* require that (1) the variable's name `vname` "looks up" in the state `s` to some type `ty`; that (2) this type `ty` be a function type; and that (3) `vname` also be present in the domain of the state's

fnencode map. If all of these conditions are satisfied, the behaviour can occur.

In this case, the behaviour leaves the state unchanged (the same s occurs in both arguments), but the expression form changes from a Var to an FVal, where the latter includes the function's type and the fact that it is not a member function (the NONE parameter to FVal).

1.2 Understanding C++ in Three Phases

When we attempt to understand the meaning of a C++ source file (or "translation unit" to use the standardese), this task is best broken down into three phases.

When we begin, we have a sequence of parsed *external declarations* (see the Standard's formal grammar in its Annex A: Grammar Summary). Throughout this semantics, we assume that this sequence is well-formed syntactically, that some trusted compiler has already checked the sources for syntactic errors of the sort compilers can detect. For example, this means that we assume that there are no variables left undeclared, and that all the various expressions are well-typed. This is a not inconsiderable simplification of the basic task, but it does seem fair to claim that such analysis is not as interesting a problem. This abstract syntax, as consumed and manipulated by the semantic model, is described in Section 2 below.

The first phase of understanding, or of ascribing meaning, is to do what I refer to as "name resolution". In this phase, bare names are resolved into fully-qualified names wherever possible. For example, this phase turns a program such as

```
int x;
int f(int i) { return i + x; }
```

into

```
int ::x;
int ::f(int i) { return i + ::x; }
```

where the names declared in the top, global namespace (x and f) have been replaced with unambiguous versions of their names wherever they occur. Note that the result is no longer valid C++ (it is illegal to use the explicit qualification in names being defined), but the next phase of understanding does not expect valid C++ in any case. In the presence of hierarchical namespaces, and class namespaces inheriting from bases, Phase 1 is not entirely trivial. It is described in some detail in Section 3.

The second phase of the semantics is to deal with templates. The input to template resolution is a translation unit with most of its names resolved, and where the translation unit consists of both “ground” (non-template) declarations or definitions, and template declarations or definitions. The ground definitions may refer to various template classes or functions. If so, the appropriate template instantiations need to be made, producing fresh ground declarations. These new declarations need to first have their names resolved (requiring a nested “call” to Phase 1), and may in turn require more template instantiations. In fact, template instantiation may never terminate. Templates are further described in Section 4.

The final phase of the semantics is execution, or “dynamics”. In this phase, top-level declarations are executed, resulting in the dynamic initialization of variables, which can in turn result in the execution of source-code. The exact order of evaluation of external declarations is allowed to vary (see [5, §3.6.2]), and may or may not precede the execution of a program’s `main` function. The rules governing dynamic behaviour specify what is to occur when execution does occur, but do not specify how various executions are knitted together. All of the model’s dynamic rules are presented in Section 5.

2 C++’s Basic Types

The most fundamental types in the semantics are those expressing the basic abstract syntax of C++. The declaration of C++ types is (from HOL:`types`)

```
CPP_Type =
  Void |
  BChar (* "Basic char" *) |
  Bool |
  Unsigned of basic_integral_type |
  Signed of basic_integral_type |
  Class of CPP_ID  |
  Float |
  Double |
  LDouble |
  Ptr of CPP_Type |
  MPtr of CPP_ID => CPP_Type | (* member pointer *)
  Ref of CPP_Type |
  Array of CPP_Type => num |
  Function of CPP_Type => CPP_Type list |
```

```
  Const of CPP_Type |
  TypeID of CPP_ID
```

This definition allows recursion: for example, a C++ type can be a pointer to another C++ type (using the `Ptr` constructor). In this simple prefix notation, the type of an "array of ten pointers to `int`", is written

```
Array (Ptr (Signed Int)) 10
```

(`Int` is one of the four possible values inhabiting `basic_integral_type`, along with `Char`, `Short` and `Long`.)

Similarly, a function taking two `int`s and returning a `char` is written

```
Function BChar [Signed Int; Signed Int]
```

Identifiers In the presence of templates, identifiers can take on forms such as

```
List<int>::fldname
```

This means that identifiers are a type in the model that must in turn be mutually recursive with the type of types. In the example above, the type `int` appears within an identifier. It is also clear that identifiers occur within types, because identifiers are the basis for naming and referring to class types.

Therefore, we must add the following to the above definition of C++ types:

```
CPP_ID = IDConstant of bool => IDComp list => IDComp ;

IDComp = IDTempCall of string => TemplateArg list
       | IDName of string
```

In other words, values of identifier type are constructed by applying the function `IDConstant` to three arguments: a boolean indicating whether or not this is an "absolute" identifier (represented in the concrete syntax by prefixing it with `::`), a list of ID components and one final ID component. We will frequently see "bare" names that are not absolute, and which do not have any outer namespaces attached. To abbreviate this common case, we define the `Base` function:

```
Base s = IDConstant F [] (IDName s)
```

An ID component might either be a simple name (the `IDName` case), or can be a simple name applied to multiple "template arguments". There are three sorts of template arguments: types, templates and values; giving

```
TemplateArg = TType of CPP_Type
            | TTemp of CPP_ID
            | TVal of TemplateValueArg
```

So, the type above would be represented as

```
IDConstant F [IDTempCall "List" [TType (Signed Int)]]
             (IDName "fldname")
```

Expressions and Statements Expressions are specified in exactly the same way as types, with constructors such as `Assign` (assignment), `Deref` (the `*` or pointer dereferencing operator) and `New`. The abstract syntax need not be a perfect match for the concrete syntax. For example, there is an `ExpTypeID` operator (for `typeid` applied to an expression argument), and `TyTypeID` for when `typeid` is applied to a type.

The rules presented in Section 5 cover the dynamic behaviours various expression forms. The HOL declaration is in the file HOL:`expressions`.

Statements are similar again (see HOL:`statements`), with constructors such as `CIf`, `Ret` and `Block`. However, the type here is rather more complicated because statements not only include expressions but must be mutually recursive with other syntactic categories: variable and class declarations, "class entries" (those things that can appear with a `class` declaration), and initializers (which appear in variable declarations and explicitly initialize the variable in question).

2.1 Bytes & Memory

After specifying abstract syntax for programs, one must continue by describing the state that is manipulated by the action of those programs. In fact, each of the three phases manipulates slightly different states, and each will be detailed in the relevant sections. However, there are a few general observations possible.

Bytes The fundamental type in the dynamic semantics is that of the byte. (See HOL:`memory` for more on these matters.) Using HOL4's support for n-bit words, it is possible to define a type called `byte`, which is a word

containing CHAR_BIT many bits, where CHAR_BIT is a natural number under-specified to be at least 8, but possibly more.

It is then possible to define representation and valuation functions between the HOL types of integer and byte. These functions are partial, meaning that they can either return NONE to indicate failure, or SOME(v) to indicate the successful return of the value v. These functions capture, again in a suitably underspecified way, how bytes can be translated into values, and how those same values can be converted back into bytes. For example, we know that for the unsigned char type, all values in the range 0 up to $2^{\texttt{CHAR_BIT}}-1$ must have corresponding bit patterns. For types other than char, the use of functions in both directions is perhaps not quite under-specified enough: it assumes that if an implementation can represent a value at all, then it will always represent a value in the same way. For example, this reduces, though does not eliminate, the opportunities for signed zeroes to occur.

Each primitive type is given a fixed size (in numbers of bytes). This is done in an underspecified way so that again, one can only conclude that there are enough bytes in an int value to represent the mandated range of values (from $-(2^{16}-1)$ up to $2^{16}-1$).

In the model to come, the function most used is

```
INT_VAL : CPP_Type -> byte list -> int option
```

which attempts to interpret the given list of bytes as a value of the provided type, and returns its integer value, if it has one. The function might return NONE if the list of bytes is of the wrong length, or if it is not a valid bit pattern for the type. (The latter might occur if the required type is a pointer value and the hardware checks such values for validity before even allowing them into address registers.)

Memory Memory is represented as a function from natural numbers to bytes. The address 0 is reserved as the representation for the null pointer. Strictly speaking, one might imagine that the map should be from some machine word (an array of four bytes, say) to bytes. However, even with the domain of the map being $\mathbb{N}$, any given program will only be able to address a finite amount of memory because it will only be able to generate a finite number of addresses. (All pointer types have a fixed, finite number of bytes making up their representations.)

2.2 Hierarchical Environments

To reflect the hierarchical nesting of namespaces and classes, the model uses a type of hierarchical environments, giving maps from structured names into information about those names. These maps are instances of a type called `fmaptree` (see HOL:`fmaptree`). This type has one constructor:

```
FTNode : 'value -> ('key |-> ('key,'value)fmaptree) ->
         ('key,'value)fmaptree
```

The `|->` type operator returns finite maps, so `FTNode` takes a value, and a finite map from keys to more `fmaptrees`, and returns a new `fmaptree`. This type rather resembles the trie data structure. Like lists and other "container" types, the `fmaptree` is polymorphic, here both in the type of keys, and the type of values.

The operation to lookup the sub-tree at a particular key list position is `apply_path`:

```
(apply_path [] ft = SOME ft) ∧
(apply_path (h::t) ft = if h ∈ FDOM (map ft) then
                          apply_path t (map ft ' h)
                        else NONE)
```

The functions `item` and `map` are also heavily used, and return the value, and sub-trees of an `FTNode` respectively.

```
item (FTNode i fm) = i

map (FTNode i fm) = fm
```

In the particular context of C++, there are two sorts of environment (see HOL:`environments`). The first maps from namespace components into values of type `envinfo`. An `envinfo` is a record of three components, each of which are finite-maps.

```
envinfo = <|
  varmap   : string |-> addr # CPP_ID # CPP_ID list ;
  typemap  : IDComp |-> CPP_Type ;
  classenv : IDComp |-> class_env
|>
```

The varmap is a map from variable names to their l-value information.[1] The domain of this map can be strings because only functions can have template-structured names, and these do not live in memory in the same way as objects. The typemap maps ID components to types, giving static information both for objects and functions. The classenv field gives information about any classes that might be declared at this level of the namespace hierarchy.

The environment type is then an abbreviation for a

```
(string, envinfo) fmaptree
```

At each point in the namespace tree, indexed by a path (or list) of names, there is an envinfo value about the objects and classes stored in that scope.

The class_env type is an abbreviation for another sort of fmaptree

```
(IDComp, class_envinfo) fmaptree
```

In other words, a class_env is a structured map, where the components can be full-blown IDComp values. This is necessary because classes can be constructed from template calls (whereas namespaces are necessarily identified by just strings).

The information attached to each node of a class_env is the following record type:

```
class_envinfo = <|
   (* ironically, the location of the static variables is
      only available dynamically, as classes are
      initialized *)
   statvars : string |-> addr # CPP_ID # CPP_ID list ;
   info     : state_class_info ;
   refs     : string # addr |-> addr # CPP_ID # CPP_ID list
|>
```

The statvars field records the same information for a class's static variables as the envinfo records for normal variables. The refs field records per-class information about reference members. Both of these fields are only used dynamically (as static variables are declared, and as new classes are constructed respectively). Finally, the info field records the static information associated with a class.

[1]It should be clear that an address is necessary to specify an object's identity. The additional identifier and identifier list are used to store the dynamic information about class types that is necessary to implement polymorphism. For more on this, see Section 5.9 below.

3 Phase 1: Name Resolution

Name resolution must occur in a separate phase before dynamic evaluation, and must rewrite declarations so that their name dependencies are made explicit. This is exemplified by the program in Figure 1, where the name `x` that occurs in the function `ns1::f` must be a reference to the `x` that occurs in the outermost, global namespace. A naïve execution of the sequence of declarations in the program would put the global `x` into its namespace, and then enter namespace `ns1`, where it would first declare the function `f`, and then the second `x`. A later call to `ns1::f` would correctly open up the entirety of the namespace, and immediately mask the global `x` with `ns1::x`, causing the evaluation of the body to proceed erroneously.

```
int x = 3;
namespace ns1 {
  int f(int n) { return n + x; }
  int x = 2;
}
```

Figure 1: A program demonstrating the need to have name resolution be a separate phase before dynamic evaluation.

Even if one imagined a version of the dynamics that did perform name resolution as it evaluated declarations, this semantics would still need to transform the body of `ns1::f` to include the correct reference to `::x`. The rest of this section of the report will describe the relation that turns the program in Figure 1 into

```
int ::x = 3;
int ::ns1::f(int n) { return n + ::x; }
int ::ns1::x = 2;
```

The main relation in Phase 1 is called `phase1`, and is defined in HOL:`name_-resolution`. In the remainder of this section, I will discuss some of the more interesting aspects of this phase of analysis.

3.1 The Name Resolution State

In order to track the current set of names that are in scope, Phase 1 uses two environments, as per Section 2.2 above, to capture what is known about the

current nested scopes, as well as some extra fields to describe the names that are *visible*. For example, in the program of Figure 1, the global x is visible when ::ns1::f is defined, but there is no x in the namespace ::ns1, at least at that stage.

The HOL definition of the Phase 1 state is (see HOL:name_resolution):

```
P1state = <|
  current_nspath : string list ;
  dynclasses : string |-> bool # IDComp list #
                          TemplateArg list ;
  dynobjs : string |-> bool # IDComp list #
                       TemplateArg list # dynobj_type ;
  dynns : string |-> string list ;
  global : state ;
  accdecls : ext_decl list
|>
```

The three dyn fields record what names are visible in three different categories: namespaces (dynns), classes (dynclasses) and objects (dynobjs). Each maps to information sufficient to provide an exact location for the name.

In the case of namespace names, it is enough to provide a path from the root, and such a path will just be of strings. For objects and classes, the path has to be of ID components because objects and classes can be nested inside classes (and classes might be template classes). The boolean also records whether or not the name is local or non-local. The list of TemplateArg values records any template parameters that the name may be associated with if it is a template. Finally, in the case of objects, it is also necessary to record what sort of object the name is. The options are given in the type dynobj_type:

```
dynobj_type =
   dStatMember | dMember | dVirtualMember | dNormalObj
```

Note that in this context an “object” might actually be a function (and it is functions that provide the interest); elsewhere (particularly in the dynamics) functions are not considered objects because they don’t occupy allocated memory.

The global field of a P1state is a state from the dynamic semantics. In Phase 1 the vast majority of the information stored in such a state is ignored; the state is part of the P1state only for its two environments, which are

accessed as the fields genv (the global environment), and env (the local environment).

Finally, the accdecls field records the accumulating translated declarations. It is this that provides the final output of Phase 1.

3.2 Template Names

One significant issue is the resolution of names in templates. When a template definition is instantiated, it is important to specify how the names occurring in the template are bound. Typically, such names might bind to global names that are in scope at the point of the template's definition, or to member functions associated with the template argument.

The rule is actually fairly straightforward, at least in principle: function names are allowed to bind to template names when the statically determined types of the function arguments refer to template parameters. Such names have to be left alone in Phase 1.

3.3 Resolving Expressions

The first interesting phase of name resolution comes when rewriting expressions. This is done with the function phase1_expr, defined in HOL:name_-resolution, and of type

```
: frees_record -> P1state -> CExpr -> CExpr
```

where the frees_record types records a set of template arguments, which names must be treated specially. (See HOL:frees for more on the calculation of this notion of "free variable".)

Most of the clauses of the function phase1_expr are uninteresting recursions, *e.g.*:

```
phase1_expr avds ps (COr e1 e2) =
  COr (phase1_expr avds ps e1) (phase1_expr avds ps e2)
```

The first interesting clause is the handling of function applications, for which see Figure 2. This is a complicated piece of code, though the structure is relatively straightforward. At the top level, the function calls itself recursively to resolve the arguments of the function call, generating the value elist'. It is the calculation of the resolved e' that introduces the complications. The first test checks if e is an unqualified variable. If not, then the recursion can proceed "normally". If e is just a simple variable however,

```
phase1_expr avoids ps (FnApp e elist) =
  let elist' = MAP (phase1_expr avoids ps) elist in
  let e' =
      if is_unqvar e then
        let fnm = dest_unqvar e in
        let atys =
           εatys. listRel (expr_type ps.global RValue) elist'
                          atys
        in
          if ¬(DISJOINT
                  avoids
                  (FOLDL (λa ty. a UNION tyfrees ty) {} atys))
          then
            e
          else if fnm ∈ FDOM ps.dynobjs then
            phase1_expr avoids ps e
          else
            let foldthis ps0 ty =
                let nss =
                  FST (ass_nspaces_classes avoids ps0 ty) in
                let nsl = SET_TO_LIST nss
                in
                  FOLDL (λps00 ns. open_ftnode ns ps00) ps0 nsl
            in
            let ps' = FOLDL foldthis ps atys
            in
              Var (idattach_locn
                     (ps'.dynobjs ' fnm)
                     (IDConstant F [] (IDName fnm)))
      else
        phase1_expr avoids ps e
  in
    FnApp e' elist'
```

Figure 2: Name Resolution in Function Applications

then the code binds `fnm` to the name of the variable, and `atys` to the types of the arguments.

The next test calculates the free type names of the argument types. If this set overlaps with the names of any template arguments, then the variable must be left alone, as *per* Section 3.2 above (it will be the subject of future work if the containing template is instantiated). Next it is possible that the name is already in the `dynobjs` map, which records information about the names that are necessarily in scope. If so, this binding for the name takes precedence, and a normal recursive call can occur. (This reflects [5, §3.4.2, paragraph 2a].)

The final part of this clause calculates the set of associated namespaces and associated classes, in accordance with the language of [5, §3.4.2, paragraph 2], and then looks for the given name in this new extended namespace. (The function `open_ftnode` is used to overlay an existing `P1state` with the names of another namespace.)

Another interesting case is dealing with variable names, assuming that they have not already been caught by the function-application clause just discussed. The `Var` clause is presented in Figure 3. There are three cases depending on the form of the variable. The first is when the variable is an "absolute" name, with a leading `::`. This might still be a member reference, as in the following

```
class C {
  int x;
  int f() { return ::C::x; }
};
```

If this possibility is ruled out, then the name can be left unchanged. (If it's a static member, the name must be fine on its own.)

The second case is when there is a single bare name, with no qualification. The qualified version of the name can be produced immediately by consulting the `dynobjs` map, giving `qid`. Then it is necessary to determine what sort of name the identifier is. If it is a virtual member function, the name needs to be left unqualified, but attached to a `this->`. If it is a normal member, then the qualified name needs to be used in the same way. Finally, if it is a static member, or non-class object, the qualified name should be used as is.

The last case is when there is a qualified name, such as `name1::name2`. This is fiddly to resolve because the first element of the qualification may refer to a namespace or a class. In either case, it is possible to resolve this

```
phase1_expr avoids ps (Var id) =
   case id of
      IDConstant T sfs sf ->
        if id_objtype ps.global id = dMember then
          let cnm = class_part id in
          let ps' = open_classnode avoids.tyfvs cnm ps
          in
            SVar (Deref This) (mk_dynobj_id ps' (IDtl id))
        else Var id
   || IDConstant F [] sf ->
        (let qid = idattach_locn (ps.dynobjs ' (sfld_string sf)) id
         in
           (case SND (SND (SND (ps.dynobjs ' (sfld_string sf)))) of
               dVirtualMember -> SVar (Deref This) id
            || dMember -> SVar (Deref This) qid
            || dStatMember -> Var qid
            || dNormalObj -> Var qid))
   || IDConstant F (h::t) sf ->
        (let s = sfld_string h in
         let qid =
             if s ∈ FDOM ps.dynclasses then
               idattach_locn (ps.dynclasses ' s) id
             else
               IDConstant T (MAP IDName (ps.dynns ' s) ++ (h::t)) sf
         in
           if id_objtype ps.global qid = dMember then
             let cnm = class_part qid in
             let ps' = open_classnode avoids.tyfvs cnm ps
             in
               SVar (Deref This) (mk_dynobj_id ps' (IDtl qid))
           else Var qid)
```

Figure 3: Name Resolution for Variables

(class names mask namespace names), generating an unambiguous `qid`. This then needs to be treated just as in the first case, possibly generating a member reference.

3.4 Resolving Declarations: Establishing Phase 1 Names

The code in `phase1_expr` uses the information that has been stored in the Phase 1 states by earlier and higher-level traversals of program syntax. In particular, when a declaration is seen, this must cause an adjustment of the information about names that is stored in the Phase 1 state.

For example, when a new global variable is declared, the `NewGVar` function is called to record this. Its type is

```
: CPP_Type -> IDComp -> P1state -> P1state
```

and its definition appears in Figure 4. The input to the `NewGVar` function

```
NewGVar ty sfnm s =
  let sfnm' =
    IDConstant T (MAP IDName s.current_nspath) sfnm in
  let (targs,sfstr) = break_sfld sfnm in
  let ty' =
    rewrite_type
      (FOLDL (λa ta. a UNION tafrees ta) {} targs)
      s ty
  in
    s with <|
      dynobjs :=
        s.dynobjs |+ (sfstr, (T,MAP IDName s.current_nspath,
                              targs, dNormalObj)) ;
      global updated_by
        state_NewGVar ty s.current_nspath sfnm ;
      accdecls :=
        (s.accdecls ++ [Decl (VDec ty' sfnm')])
    |>
```

Figure 4: Defining a New Global Variable in Phase 1

is the type of the variable being declared, its bare name (`sfnm` here), and the Phase 1 state being updated. The type is rewritten so that any of the names it mentions are themselves resolved, and the state's `dynobjs` field is

updated to map the name to a full identifier, using the current namespace path. (The 4-tuple has `T` as its first component because this is known to be a global variable; were we defining a local name, this component would be `F` instead.)

Class Declarations The most interesting behaviour at the level of declarations comes with classes. Here scopes are twisted in order to allow member functions to refer to member names that have actually been declared later in the class. For example, in

```
int x;
class C {
  int f(int i) { return i + x; }
  int x;
}
```

the reference to `x` in the body of `C::f` is to `C::x`, not to the global `x`. This special dispensation only extends as far as the bodies of member functions. Sub-classes and their names are not so lucky; there the rule is that the class declaration must come first. If there is a conflict with an external type-name, this is an error. For example, the following is an error:

```
struct B { int x; };
struct C {
  B b;
  struct B { int j; };
};
```

To make this correct, one should either explicitly qualify the type in the declaration of member `b` (one could write `::B b;` to refer to the external class), or reorder so that the declaration of the nested class came before the definition of `b` (which would then make field `b` of `C` have type `C::B`).

This analysis is illustrated in the definition of `phase1_gclassdefn`, one clause of which is given in Figure 5. The first pass over the fields appears in the binding for value `ps2`. The function called there, `extract_class_names` passes over the fields of the class-info value (`ci`), and adds the names it sees there to the state `ps1`. The resolved fields are then calculated in the last `let` binding (to `flds'`). Note that the state returned from the function adds the resolved declaration to its accumulating list of declarations (`accdecls`), but otherwise resets its dynamic state to that which held immediately after the declaration of the class (`ps0`).

```
phase1_gclassdefn avds cnm (SOME ci) ps =
  let ancs' = MAP (λ(id,b,p). (resolve_classid ps id, b, p))
                  ci.ancestors in
  let fullnm =
        case cnm of
           IDConstant b [] sf ->
              IDConstant b (MAP IDName ps.current_nspath) sf
        || IDConstant b sfs sf -> resolve_classid ps cnm in
  let ps0 = ps with global updated_by
              new_class fullnm
                        (SOME (<| ancestors := ancs' ;
                                  fields := [] |>, {})) in
  let ps1 = open_path avds.tyfvs T (id_sfs fullnm) ps0 in
  let ps2 = FOLDL (λs cebp.
                     extract_class_names avds s
                        (T, id_sfs fullnm) cebp)
                  ps1
                  ci.fields in
  let ps3 = open_path avds.tyfvs T (id_sfs fullnm) ps2 in
  let flds' =
      FOLDL (λcelist (ce,b,p).
               let ce' = phase1_gcentry avds ps3 cnm ce
               in
                 (celist ++ [(ce',b,p)]))
            []
            ci.fields
  in
    ps3 with <|
      dynobjs := ps0.dynobjs ;
      dynclasses := ps0.dynclasses ;
      accdecls :=
        (ps.accdecls ++
         [Decl (VStrDec fullnm
                  (SOME <| ancestors := ancs' ;
                           fields := flds' |>))])
    |>
```

Figure 5: Name Resolution for a Global Class Declaration

Function Definitions The final aspect of name resolutions worth illustating comes with the constant `phase1_fndefn`, which handles resolution for top-level function definitions. This is another large definition, presented in Figure 6. The big case-split on the form of the function name (`fnm`) binds two different Phase 1 states. The `declared_ps` is what is returned; and may reflect the fact that a new function is being defined. (When the name is qualified in some way, then the function has already been declared (usually in a class).)

The `body_ps` is the name information that needs to be used when the body of the function is analysed. For example, consider the following:

```
namespace ns {
  int x;
  class C { int f(); int i; };
}
int ns::C::f() { return x + i; }
```

When one comes to analyse the function `f`, both the global variable `ns::x`, and the member variable `C::i` are in scope. This adjustment of scopes is done by the calls to `open_path`.

4 Phase 2: Templates

In this section, I describe how the semantics models templates. I have been inspired by Siek and Taha [12], though as I shall discuss, the dynamics of their model is too simplistic for the full language of templates, in particular handling template parameters that are themselves references to templates ("higher order templates" if you will).[2]

One might imagine that it be possible to treat templates at "run-time", as if one had written a one-pass C++ interpreter. However, the example in Figure 7 demonstrates that such a goal is impossible, or at least that one would have to write a two pass interpreter. Any reference to the figure's `List` template, perhaps within a function that was called at a great stack depth, causes the need to statically initialize the global `node_count` before the program even begins. A naïve interpreter that attempts to execute the program source as the source is read in will come unstuck.

As the interpreter sees the template declarations above, it does nothing. Then it performs its global declarations, and jumps into main. At this point it has already failed to do the right thing. Instead, the putative interpreter

[2] Siek and Taha do model `typedef` declarations within classes, which I do not.

```
phase1_fndefn avds retty fnm pms body ps =
  let retty' = rewrite_type avds ps retty in
  let pms' =
    MAP (λ(nm,ty). (nm, rewrite_type avds ps ty)) pms in
  let funty = Function retty' (MAP SND pms') in
  let (fnm',declared_ps, body_ps) =
    case fnm of (* if the name is qualified, then there must be an
                   existing declaration, so we don't need to alter
                   anything in the state *)
       IDConstant T [] sf -> ARB (* must be an error, see 8.3 p1 *)
    || IDConstant T (sf1::sfs) sf2 ->
          (fnm, ps, open_path avds.tyfvs T (sf1::sfs) ps)
    || IDConstant F [] sf ->
          (IDConstant T (MAP IDName ps.current_nspath) sf,
           NewGVar funty sf ps,
           NewGVar funty sf ps)
    || IDConstant F (sf1::sfs) sf2 ->
          let fnm' = resolve_classid ps fnm in
          let (b,sfs,sf) = dest_id fnm'
          in
            (fnm', ps, open_path avds.tyfvs b sfs ps)
  in
  let ps' =
    FOLDL (λps (n,ty).
               newlocal ps (IDName n) ty) body_ps pms' in
  let body' = phase1_stmt avds ps' body
  in
    declared_ps with
      accdecls := (declared_ps.accdecls ++
                   [FnDefn retty' fnm' pms' body'])
```

Figure 6: Name Resolution for a Global Function Definition

```
template <class T> class List {
  T item;
  List *next;
  static int node_count;
};

template<class T> int List<T>::node_count = 0;

int main(void)
{
  List<int> mylist;
  ...
}
```

Figure 7: A program demonstrating the difficulty of interpreting templates.

would have to scan the whole program for template applications so that it can generate the appropriate global variable initializations. This is no better than explicitly pre-compiling templates, so I have adopted an explicit two-phase compilation approach.

We need to specify the possible sorts of arguments that can be passed to templates. The Standard is quite explicit here [5, §14.3 para 1], there are three sorts of arguments: types, templates, and "non-type, non-template" arguments (meaning references to objects with linkage, or numbers). Thus:

```
TemplateArg = TType of CPP_Type
            | TTemp of CPP_ID
            | TVal of TemplateValueArg
```

Finally, there are four different sorts of non-type, non-template arguments [5, §14.3.2 para 1]:

```
TemplateValueArg =
    TNum of int
  | TObj of CPP_ID
      (* id is of suitable global (one that has
         linkage etc). *)
  | TMPtr of CPP_ID => CPP_Type
  | TVAVar of string (* => CPP_Type *)
      (* can have a value (of the given type)
         substituted for this *)
```

This presentation is slightly simplified because the standard also allows arithmetic on these arguments, where this is appropriate (between `TNum` and `TVAVar` parameters).

4.1 Instantiation and Matching

(This seection describes formalisation done in HOL:`instantiation`.)

Types and identifiers can be *instantiated*: mappings from variable names to values are applied over the structure of the value (type or identifier), and occurrences of variable names are replaced by the appropriate element from the range of the function. Because there are three sorts of variables (corresponding to the three different sorts of template argument), an instantiation is actually a triple of functions (one for each sort of variable).

In Siek and Taha [12], instantiation is a very elegant operation. In a more faithful model of more of C++, more complexities intrude. In addition to the need for three mappings, the model must also accept that instantiation can result in an invalid result. Instantiation must become partial, which is modelled by making the types of the various instantiation functions be of the form

$$\texttt{inst<}\tau\texttt{>} : \mathit{substitution} \rightarrow \tau \rightarrow \tau\ \mathsf{option}$$

The partiality arises at the lowest level, as in the following example:

```
template<class T> void f<T>(int x) { T::staticfield = x; }
void g() { f<int>(3); }
```

This must be an error because it is not sensible to write `int::staticfield`. (Other type substitutions may also cause this to be an error, but this error can be detected as the substitution is done, without any need to look up any information about the argument.)

The partiality of instantiation does not prevent us from defining a partial order over types, such that $\tau_1 \leq \tau_2$ when τ_2 is a more specialised/instantiated version of τ_1. As in Siek and Taha [12], we can prove reflexivity, transitivity, and antisymmetry (up to renaming of free variables). (These are sanity results, requiring substantial proof.)

Given the partial order, it is straightforward to find the best match amongst a set of template definitions for a given template call.

4.2 Program Instantiation

Siek and Taha have an elegant model for program instantiation. A program is a sequence of definitions (of classes, and of static member functions). A

```
template <class T> class Foo { static int f(); };
template <class T> int Foo<T>::f() { return 3; }

template <class T> class Bar { static int g(); };
template <class T> int Bar<T>::g() { return Foo<T*>::f(); }
```

Figure 8: In Siek and Taha's model, the definition of class `Foo<T>` will get instantiated to provide a definition of class `Foo<T*>` when a reference to that type is seen inside the definition of `Bar<T>::g`.

definition may cause an existing template to be instantiated because of a reference to that template within the definition. When a member function definition is encountered, if its body includes a reference to other functions, these functions may need to be instantiated.

For example, when analysing the program in Figure 8, the Siek and Taha's model will see the reference to `Foo<T*>::f()` in the definition of `Bar<T>::g()` and instantiate the definition of `Foo` (it knows that it does not already have an instantiation for a type of the form `Foo<T*>`). This instantiation will result in a template definition (one with free variables), which may or may not be required in the rest of the program.

This model breaks down in the presence of template parameters that are templates because it becomes impossible to determine the dependencies of a template definition. In the program in Figure 9, it is impossible to tell what definition should be instantiated when processing the definition of `Baz<A>::g`. In the presence of template parameters, Siek and Taha's model is too eager.

My model only performs instantiations when there is a ground instance to drive the instantiation. Otherwise, it is similar to what is presented in Siek and Taha. In particular, it is crucial to avoid instantiating member functions unless they are called for.

The model in HOL:`templates` script is based around a working state of two components: a Phase 1 state, and a sequence of ground declarations. In addition, the template relations take a sequence of template declarations, the "patterns". These patterns do not change as template instantiation is performed, whereas the other components do. Thus, the general forms of the relations' definition is

```
template_relation pats (ps0,grds0) (ps,grds) = ...
```

Template instantiation only begins when all of the program's declara-

```
template <class T> struct Foo { static int f(); };
template <class T> int Foo<T>::f() { return 3; }

template <class T> struct Bar { static int f(); };
template <class T> int Bar<T>::f() { return 4; }

template <template <class> class A> struct Baz {
  static int g();
};
template <template <class> class A>
int Baz<A>::g() {
  return A<int>::f();
}

int main() { return Baz<Foo>::g(); }
```

Figure 9: Siek and Taha's model's early instantiation breaks down when it sees the definition of `Baz<A>::g`, and the call to `A<int>` in particular. At this point, it can not tell which template is being instantiated in the body. By way of contrast, my model doesn't instantiate anything until it sees the definition of `main`. (This program is available as `notes/siek-taha-tempvar.cpp`)

tions have been analysed in Phase 1. This means that the process begins with a complete Phase 1 state, and with all of the translation unit's definitions "to hand". This is the only real difference between the model here and Siek & Taha's. Their core relation is presented in their Figure 6, and consists of 4 rules, including one that checks when instantiation can finish. It is their last rule, (C-FUN) that represents a problem: when their system encounters a member function definition, it analyses it there and then.

In contrast, my model only performs Phase 1 analysis on such functions. All template work is deferred until the end of Phase 1. Further, template work requires that all class definitions have member function definitions stripped from them and turned into separate top-level definitions. This is not a difficult operation to perform. At this point, my model repeatedly performs the equivalent of Siek and Taha's rule (C-INSTFUN).

The top-level definition (from HOL:`templates`) is

```
template_analysis pats =
  TC ((template_phase2_destructors pats RUNION
       template_phase2_constructors pats RUNION
       template_phase2_statmems pats RUNION
       template_phase2_fns pats) O
      TC (template_phase1 pats))
```

where `TC` is transitive closure, `RUNION` is relation union, and `O` is relation composition.

The first phase is `template_phase1`, which is instantiates any needed template classes:

```
template_phase1 pats (ps0, grds0) (ps, grds) =
  ∃id id' sub ci0.
    id ∈ used_ttypes pats grds0 ∧
    id ∉ declared_types grds0 ∧
    (∀sub_id. sub_id is_sub_tid id ⇒
              sub_id IN declared_types grds0) ∧
    best_class_match pats id sub (id', ci0) ∧
    phase1 ([P1Decl
               (Decl
                  (VStrDec id
                     (cinfo_inst sub (THE ci0))))], ps0)
           ([], ps) ∧
    (grds = grds0 ++ [LAST ps.accdecls])
```

This rule first looks for a type name `id` that is both referred to in the ground input declarations (`grds0`), and which is not declared there. It then checks

that all identifiers that are a proper prefix of `id` *are* declared.[3] Then, the relation `best_class_match` scans the template declarations, and returns the best (most specific) match. In fact, it returns both the instantiation required to turn the template ID (`id'`) into the desired ID (`id`) and the template class declaration, which is `ci0`.

The penultimate conjunct of the definition instantiates the class-info, and performs a Phase 1 analysis of it. The resulting class declaration can be appended to the list of ground declarations.

The second phase of template analysis looks to instantiate member functions (including constructors and destructors), as well static member objects. The pattern is similar for each constant. The following is the rule for member functions.

```
template_phase2_fns pats (ps0, grds0) (ps, grds) =
  ∃id id' sub retty pms bod.
    id ∈ used_tfns pats grds0 ∧
    id ∉ defined_fns grds0 ∧
    (∀sub_id. sub_id is_sub_tid id ⇒
              sub_id ∈ declared_types grds0) ∧
    best_function_match pats id sub
                         (id', (retty, pms, bod)) ∧
    phase1 ([P1Decl
               (FnDefn
                  (THE (type_inst sub retty))
                  id
                  (MAP (λ(n,ty).
                          (n,THE (type_inst sub ty)))
                       pms)
                  (THE (stmt_inst sub bod)))], ps0)
           ([], ps) ∧
    (grds = grds0 ++ [LAST ps.accdecls])
```

The basic pattern of this rule is similar to that of `template_phase1`. First, an identifier `id` is found that is both referred to among the ground declarations, and which is not ground itself. If a matching function can be found, then this is instantiated, and passed to the Phase 1 analysis to have its names resolved. It is at this point that names we avoided in Figure 2 (page 22), will get their chance to find resolution in appropriate namespaces, because arguments that had been templates will now be concrete, ground types.

[3] Thus, a reference to something like `C<int>::D<char>` will result in the instantiation of `C<int>` first.

Note that this rule also handles template instantiation for function templates that are not associated with a class, though it is an incomplete picture because the best_function_match relation does not consider other ground functions, and so doesn't deal with overloading resolution.

The definition of best_function_match appears in Figure 10. A lot of the "heavy lifting" in this definition of the behaviour of templates comes from the operations of matching and instantiation. This is the strength of Siek and Taha's approach. In this definition, the best match is one that finds a sub that does indeed instantiate the pattern id' to the desired id (instantiating only the variables that are identified as template parameters). The second half of the definition requires that the match be "best" by requiring that all other contenders be instantiable to it.

```
best_function_match Temps id sub (id', (retty,pms,bod)) =
  (∃targs.
      (cppID_inst sub id' = SOME id) ∧
      MEM (TemplateDef targs (FnDefn retty id' pms bod))
          Temps ∧
      sub only_changes targsfrees targs) ∧
  ∀id2 retty2 pms2 bod2 sub2 targs.
      MEM (TemplateDef targs (FnDefn retty2 id2 pms2 bod2))
          Temps ∧
      sub2 only_changes targsfrees targs ∧
      (cppID_inst sub2 id2 = SOME id) ⇒
      ∃sub'. (cppID_inst sub' id2 = SOME id') ∧
             sub' only_changes targsfrees targs
```

Figure 10: Finding the Best Function Instantiation

5 Phase 3: Dynamics

Much of the core semantics in this section is based on the C semantics presented in my PhD thesis [6]. In particular, details of the way in which side effects are created and applied remain the same, as does the use of an evaluation context, and the way in unspecified order of evaluation is handled.

5.1 Dynamic States

The dynamic semantics updates values of the type `state` (see HOL:`states`), given in full in Figure 11. The first four components of the state are sets of addresses. The `allocmap` and `initmap` sets are from my C model [6], and record which addresses have been allocated and initialized, respectively. The new field `hallocmap` is necessary to allow memory to be allocated on the heap, and for its life-span to persist beyond the end of the current block. The second new field, `constmap` records which memory has been allocated with the `const` *cv*-qualifier. (Updating such memory causes undefined behaviour.)

5.2 The Dynamic Relation

The fundamental relation in the dynamics semantics is `mng` (or "meaning"), which is a binary relation on states and abstract syntax forms. For reasons to do with the prevention of function call interleaving (explained below in Section 5.5), this one relation is used for both expression and statement forms. (One might otherwise imagine two mutually recursive relations: one for statements and the other for expressions.)

Thus the type of `mng` is

```
: (state # ExtE) -> (state # ExtE) -> bool
```

making it a binary relation on pairs of states and *extended expressions*. An extended expression is either

- a syntactic expression coupled with a side effect information record (containing the three fields, `update_map`, `ref_map` and `pending_ses`, (R, Υ and Π in the terminology of my thesis)); or
- a statement coupled with a continuation, which latter is a function that takes a value and returns an expression. This latter is used to recreate the expression in which the function call that generated the statement occurred. Also, all expressions within statements (such as those that appear as guards in loops and `if`-statements), are actually extended expressions.

In HOL:`statements`, the declaration of extended expression (`ExtE`) is thus mutually recursive with the type of statements:

```
ExtE = EX of CExpr => se_info
     | ST of CStmt => conttype
```

```
state = <|
  allocmap : addr -> bool ;  (* the set of stack-allocated addresses *)
  hallocmap: addr -> bool ;  (* the set of heap-allocated addresses *)
  constmap : addr -> bool ;  (* the set of read-only addresses *)
  initmap  : addr -> bool ;  (* the set of initialised addresses *)

  fnmap    : CPP_ID |-> fn_info ;
             (* map from function 'names' to type information about
                the given functions *)
  fnencode : CPP_ID |-> byte list ;
             (* map encoding function 'name' as a byte sequence
                so that its address can be stored in memory *)
  fndecode : byte list |-> CPP_ID ;
             (* map inverting fnencode *)

  genv: environment ; (* non-local environment *)
  env : environment ; (* local version of the above *)

  locmap   : addr -> byte ;
             (* memory.  Domain might also be ( void * ) words *)

  stack    : (environment # CExpr option #
              (addr->bool) # (addr#CPP_ID) list) list ;
             (* stack of (environment, this, allocation map, and objects
                that need to have their destructors called)
                Updated as blocks are entered and left *)
  rvstk    : (num # addr # CPP_ID) option list ;
             (* optional return addresses for object r-values, also
                identifying the allocation level for the r-value along with
                the type of the object being constructed. *)

  thisvalue: CExpr option ;
             (* the value (i.e., this will always be an ECompVal
                with a pointer value) of the this expression *)

  current_exns : CExpr list
                 (* stack of exceptions that might be subjected
                    to a bare throw *)
|>
```

Figure 11: The HOL type of dynamic state. There are two environment values, genv, and env. The former is for non-local, persistent identifiers, the latter for local identifiers. Because there is no such thing as a local namespace, there will only be a top-level node in the env field (which may, however point to an arbitrarily deep class_env).

Most of the time reduction occurs between expressions and expressions, or between statements and statements, allowing one to imagine that one has the $\rightarrow_e$ and $\rightarrow_s$ from the PhD's C semantics. For example, when evaluating expressions, rules in the dynamics have conclusions of the form

```
mng (s0, EX e0 se0) (s, EX e se)
```

where `s0` and `s` are the initial and final states; `e0` and `e` are the initial and final expression forms; and `se0` and `se` are the initial and final "side effect records".

Similarly, when evaluating statements, conclusions are typically of the form

```
mng (s0, ST st0 c) (s, ST st c)
```

where `st0` and `st` are the initial and final statement forms. Note that the continuation (`c` above) never changes within statement evaluation, meaning that statement rules will always actually repeat the given continuation from initial to final tuple.

5.3 Special Syntactic Forms

In the abstract syntax types representing both expressions and statements, I have added special forms that only arise as a result of evaluation and could never be seen in an input program. The most important of these are the forms for representing values and l-values within the expression type.

The `ECompVal` constructor has type

```
: byte list -> CPP_Type -> CExpr
```

and represents values as sequences of bytes, coupled with their type.

The `LVal` constructor is used to represent l-values, and has type

```
: addr -> CPP_Type -> CPP_ID list
```

Thus, an l-value is represented by a combination of its base address and its type, along with the list of identifiers that allow us to represent the dynamic types of basic object orientation; see Section 5.7 below.

In addition, there is an analogue to `LVal` for functions, called `FVal`. This represents the identity of a function, and has type:

```
: CPP_ID -> CPP_Type -> CExpr option -> CExpr
```

A function is identified by its name, its type, and if a (non-static) member function, the expression denoting the class object for which it is to be called.

Finally, there is the special value `UndefinedExpr` used to represent the occurrence of undefined behaviour within an expression.

5.4 Simple Expression Rules

Introducing Scopes for Expressions Because C++ expressions can cause the creation of object r-values (for which, see Section 5.11), it is necessary that the internal stack be extended by a new entry when the evaluation of an expression within a statement begins. This new element on the stack records the creation of objects during the evaluation of the expression, and also records the state of the `allocmap` that held when the expression began, so that it can be returned to on finishing the expression's evaluation.

The model assumes that all expressions within statements are initially wrapped inside the `NoScope` constructor, of type

```
: CExpr -> CExpr
```

This wrapper is dissolved once for every whole expression, in the rule *[noscope]*:

```
(* RULE-ID: noscope *)
     T
   ==>
     mng (s, EX (NoScope e) se)
         (s with stack updated_by
                  CONS (s.env,s.thisvalue,s.allocmap,[]),
          EX e se)
```

The rules for standalone and conditional statements, and for declarations with initialisors pop this extra "frame" off the stack when expression evaluation terminates.

Literals We begin with two rules for literals. We don't have any rules for other literal forms, such as floating point constants, though it is clear what they would look like.

```
(* RULE-ID: number-literal *)
     (REP_INT (Signed Int) n = SOME bl)
   ⇒
     mng (s, EX (Cnum n) se)
         (s, EX (ECompVal bl (Signed Int)) se)
```

The only difference with character constants is that the underlying number is pushed into a different sized space:

```
(* RULE-ID: char-literal *)
     (REP_INT BChar n = SOME bl)
   ⇒
     mng (s, EX (Cchar n) se) (s, EX (ECompVal bl BChar) se)
```

Though not really a literal, the special C++ expression form `this` also has a simple rule:

```
(* RULE-ID: this *)
     T
   ⇒
     mng (s, EX This se) (s, EX (THE s.thisvalue) se)
```

Variables Looking up object variables becomes a little complicated in the presence of references and object orientation.

```
(* RULE-ID: var-to-lvalue *)
     (lookup_type s vname = SOME ty0) ∧
     object_type ty0 ∧
     (lookup_addr s vname = SOME (a,cnm,p)) ∧
     (ty = if class_type ty0 then Class cnm else ty0)
   ⇒
     mng (s, EX (Var vname) se) (s, EX (LVal a ty p) se)
```

The call to `lookup_type` determines the variable's static type, which will have been set in the appropriate part of the state when the variable was declared. The second premise checks to see that the variable is of object type. If so, the variable will have an address. Accompanying the address is information (`cnm` and `p`) that gives dynamic type information if the object is of class type.

It may not be clear how a variable may come to have a dynamic type that is separate from its static type. In fact, this is only possible in the presence of references, which are treated as aliases for real variables. Thus, in a function such as

```
int f(C &c) { return c.memfn(); }
```

the variable `c` is initialized to "point at" some existing variable, and the address maps are set up so that `c` is indeed a perfect alias for some existing l-value. But, the argument may in fact have been a derived class of `C`, and so `c`'s dynamic type won't be the same as its static type.

Variables can also denote functions:

```
(* RULE-ID: var-to-fvalue *)
     (lookup_type s vname = SOME ty) ∧
     function_type ty ∧
     vname ∈ FDOM s.fnencode
   ⇒
     mng (s, EX (Var vname) se) (s, EX (FVal vname ty NONE) se)
```

Contextual Evaluation Just as in the C semantics, most nested evaluation of expressions is mediated through one rule:

```
(* RULE-ID: econtext-expr *)
     mng (s0, EX e0 se0) (s, EX e se) ∧
     valid_econtext f
   ⇒
     mng (s0, EX (f e0) se0) (s, EX (f e) se)
```

Here, `f` is a function of type `:CExpr->CExpr`, but restricted by the predicate `valid_econtext`. This predicate restricts where evaluation can occur. For example, a function satisfying `valid_econtext` would be

$$\lambda e.\ \texttt{CAnd}\ e\ e_2$$

for all possible values e_2. Such a function allows reduction to occur to the left of the `CAnd` constructor (*i.e.*, `&&`). The corresponding function with its "hole" on the right of the `CAnd` is not a valid context function.

If an undefined behaviour occurs, this is reflected by having the expression that caused it become the special `UndefinedExpr` value. This value can rise to the top of any expression:

```
(* RULE-ID: econtext-undefinedness *)
     valid_econtext f
   ⇒
     mng (s, EX (f UndefinedExpr) se) (s, EX UndefinedExpr se)
```

The notion of where a function l-value decays into a pointer to a function is also controlled by a context:

```
(* RULE-ID: fcontext *)
     fnid ∈ FDOM s.fnencode ∧
     (s.fnencode ' fnid = bytes) ∧
     valid_fvcontext f
   ⇒
     mng (s, EX (f (FVal fnid ty NONE)) se)
         (s, EX (f (ECompVal bytes (Ptr ty))) se)
```

The definition of `valid_fvcontext` is

```
valid_fvcontext f =
   valid_econtext f ∧
   ∀args. ¬(f = λf'. FnApp f' args)
```

stating that a function l-value can decay as in the rule above, as long as it is not at the head of a function application.

Finally, there is the rule allowing normal l-values to decay into their r-value forms (the "l-value to r-value conversion"):

```
(* RULE-ID: lvcontext *)
     valid_lvcontext f ∧
     lval2rval (s0,e0,se0) (s,e,se)
   ⇒
     mng (s0, EX (f e0) se0) (s, EX (f e) se)
```

The `lval2rval` relation can result in an `UndefinedExpr` if the l-value causes a reference to a value that has already been updated within the same phase of execution, as might happen in the expression `i++ + i` for example. Otherwise, if the l-value denotes an object not of class type, the l-value turns into a list of bytes (an `ECompVal`), ready for further manipulations to occur.

Operators The rules governing the behaviour of the standard operators are as in the original C semantics. The rules for the standard computational binary operators (arithmetic and shift operators) are presented in Figure 12, and depend on an auxiliary relation `binop_meaning`, which is defined in HOL:`operators`. Being a relation, it allows for nondeterminism and failure.

Were the model to cope with operator overloading correctly, these rules would remain unchanged. Operator overloading would be resolved in Phases 1 and 2, allowing uses of overloaded operators to be rewritten to the function calls that they really are.

There are analogous rules for the standard unary operators (arithmetic and logical negation, unary plus, and bit-wise complement), presented in Figure 13.

Sequential Operators The logical operators `&&` and `||`, the ternary conditional operator `?:`, and the comma operator all evaluate their arguments in a prescribed order, and must exhaust the pending side effects that may have accumulated in a state before they can move from one argument to the next, if indeed they move on at all. For example, the rule where logical-and returns false is

```
(* RULE-ID: binop-fails *)
     (∀res restype. ¬binop_meaning s f v1 (strip_const type1)
                                       v2 (strip_const type2)
                                       res restype)
   ⇒
     mng (s, EX (CApBinary f (ECompVal v1 type1)
                             (ECompVal v2 type2)) se0)
         (s, EX UndefinedExpr se0)

(* RULE-ID: binop-computes *)
     binop_meaning s f v1 (strip_const type1)
                       v2 (strip_const type2)
                       res restype
   ⇒
     mng (s, EX (CApBinary f (ECompVal v1 type1)
                             (ECompVal v2 type2)) se)
         (s, EX (ECompVal res restype) se)
```

Figure 12: Rules for the Standard Binary Operators

```
(* RULE-ID: unop-computes *)
     unop_meaning f ival (strip_const t) result rt
   ⇒
     mng (s, EX (CApUnary f (ECompVal ival t)) se)
         (s, EX (ECompVal result rt) se)

(* RULE-ID: unop-fails *)
     (∀res rt. ¬unop_meaning f ival (strip_const t) res rt)
   ⇒
     mng (s0, EX (CApUnary f (ECompVal ival t)) se0)
         (s0, EX UndefinedExpr se0)
```

Figure 13: Rules for the Standard Unary Operators

```
(* RULE-ID: and-false *)
     is_zero t v
   ⇒
     mng (s, EX (CAnd (ECompVal v t) sub2) se)
         (s, EX (ECompVal (signed_int 0) Bool) se)
```

When the first argument is true (non-zero), the truth value of the second argument is the result. The conversion of the second argument to a truth value is achieved by negating twice:

```
(* RULE-ID: and-true *)
     ~is_zero t v ∧
     is_null_se se
   ⇒
     mng (s, EX (CAnd (ECompVal v t) sub2) se)
         (s, EX (CApUnary CNot (CApUnary CNot sub2)) base_se)
```

The `is_null_se` predicate tests whether or not a side effect record is empty of pending side effects. The `base_se` value is the empty side effect record; it is the appropriate starting point for a new phase of execution after a sequence point has been reached.

The comma operator (`CommaSep` here) always evaluates both of its arguments. As with the other operators, evaluation of the first argument is handled by the contextual evaluation rule; we need only provide a rule for when the left-hand expression has been fully evaluated.

```
(* RULE-ID: comma-progresses *)
     final_value (EX e1 se)
   ⇒
     mng (s0, EX (CommaSep e1 e2) se)
         (s0, EX e2 base_se)
```

The whole expression is an l-value if the second expression is (this is different from C's behaviour) [5, §5.18] (see also HOL:`notes/comma-lvalue.cpp`). The `final_value` predicate (from HOL:`statements`) checks an extended expression to confirm that it represents a completely evaluated form. Its definition is

```
(final_value (EX e se) =
   is_null_se se ∧
   ((∃v t. e = ECompVal v t) ∨
    (∃a t p. e = LVal a t p) ∨
    (∃b a i. e = ConstructedVal b a i))) ∧
(final_value (ST s c) = F)
```

The ConstructedVal constructor is used to refer to class r-values (for which see Section 5.11).

Pointer Operations The two significant pointer operations are dereferencing and address-taking (the * and & operators respectively). C++ adds two class-related variants of these operations to the existing pair, which come across from C practically unchanged. Thus the rule for taking an address:

```
(* RULE-ID: addr-lvalue *)
(* See 5.3.1 p2-5 - taking the address of an lvalue *)
     (SOME result = ptr_encode s a t pth)
   ⇒
     mng (s, EX (Addr (LVal a t pth)) se)
         (s, EX (ECompVal result
                          (Ptr (static_type (t,pth)))) se)
```

The function static_type computes the static type for a l-value given access to the type and path of identifiers. (For non-class values, the static type is simply the second argument to LVal.)

There must be two rules for dereferencing a normal pointer; it may or may not point to a valid object. In fact, there must also be a third rule because of the possibility that the pointer value being dereferenced is actually a pointer to a function. These rules are presented in Figure 14.

More interesting from a C++ perspective are the rules for taking the address of class-members (and then using those pointers to members to access members). The concrete syntax for this is rather ugly:

```
struct C { int x; int y; };
int C::* cintptr = &C::x; // or &C::y
```

In the abstract syntax, the address taking operation is written MemAddr, of type

```
: CPP_ID -> IDComp -> CExpr
```

where the first argument is the name of the class, and the second is the name of the field.

If the member whose address is being taken is actually a static member, then a normal pointer is generated, as can be seen from the type attached to the ECompVal in the conclusion of the rule. (There is a similar rule for taking the address of a static member function.)

```
(* RULE-ID: deref-objptr *)
(* 5.3.1 p1 - pointer to an object type *)
     object_type t ∧
     (SOME mval = ptr_encode s addr t' pth) ∧
     (static_type (t',pth) = t)
   ⇒
     mng (s, EX (Deref (ECompVal mval (Ptr t))) se)
         (s, EX (LVal addr t' pth) se)


(* RULE-ID: deref-objptr-fails *)
     object_type t ∧
     ((∀addr t' p. ¬(SOME mval = ptr_encode s addr t' p)) ∨
      (∃t' p. SOME mval = ptr_encode s 0 t' p))
   ⇒
     mng (s, EX (Deref (ECompVal mval (Ptr t))) se)
         (s, EX UndefinedExpr se)


(* RULE-ID: deref-fnptr *)
(* 5.3.1 p1 - pointer to a function type *)
     v ∈ FDOM s.fndecode
   ⇒
     mng (s, EX (Deref
                  (ECompVal v
                            (Ptr (Function retty argtys))))
                se)
         (s, EX (FVal (s.fndecode ' v)
                      (Function retty argtys)
                      NONE) se)
```

Figure 14: Rules for “C-style” Dereferencing of Pointers

```
(* RULE-ID: mem-addr-static-nonfn *)
(* 5.3.1 p2 *)
     object_type ty ∧
     MEM (FldDecl fldname ty, T, prot)
         (cinfo s cname).fields ∧
     (lookup_addr s (mk_member cname fldname) =
        SOME (addr, pth)) ∧
     (SOME ptrval = ptr_encode s addr ty (SND pth))
   ⇒
     mng (s, EX (MemAddr cname fldname) se)
         (s, EX (ECompVal ptrval (Ptr ty)) se)
```

The function `cinfo` takes a state and a class-name and returns the information about that class in the form of a record, one of whose fields is called `fields`, being a list of all the declarations occurring within the class.

When the member is not static, the rules become a little more involved. The rule for taking the address of a non-static member, *[mem-addr-nonstatic]* is presented in Figure 15. Functions and data members are not declared in

```
(* RULE-ID: mem-addr-nonstatic *)
     (encode_offset cnm fldname = SOME bl) ∧
     ((∃prot. MEM (FldDecl fldname ty, F, prot)
                  (cinfo s cnm).fields) ∨
      (∃prot v rt args bod.
          MEM (CFnDefn v rt fldname args bod, F, prot)
              (cinfo s cnm).fields ∧
          (ty = Function rt (MAP SND args))))
   ⇒
     mng (s, EX (MemAddr cnm fldname) se)
         (s, EX (ECompVal bl (MPtr cnm ty)) se)
```

Figure 15: Taking the Address of a Non-static Member

quite the same way within a class (because the functions may be accompanied by their implementations), which explains the disjunctive hypothesis in the figure. The function `encode_offset` takes a class and field-name and returns the encoding of this offset as a list of bytes. It is this that will be written into memory if the offset is stored in a variable.

Once one has a pointer-to-member value, one can dereference it, as long as one also had a class object to hand. Unlike normal C-style dereferencing,

dereferencing a pointer-to-member is a binary operator. The concrete C++ syntax looks like:

```
int f(int C::* cintptr, C c)
{
  return c.*cintptr;
}
```

There is also a `->*` operator for when one has a pointer to a class (by analogy with the `->` operator for field dereferencing). In the abstract syntax, the one operator is called `OffsetDeref`.

The rule for dereferencing a pointer-to-member (*[offset-deref]*) is presented in Figure 16. The rule is complicated by the fact that if the pointer to a member is to a virtual function, then the result must be a reference to a virtual function in the class on which the dereference is performed. The in-

```
(* RULE-ID: offset-deref *)
     (encode_offset cnm2 fldname = SOME bl) ∧
     (derive_objid obj = SOME (a,Class cnm1,p)) ∧
     (fld = if function_type fldty then
              let (r,a) = dest_function_type fldty
              in
                if is_virtual s cnm2 fldname r a then
                  IDConstant F [] fldname
                else
                  mk_member cnm2 fldname
            else
              mk_member cnm2 fldname)
   ⇒
     mng (s, EX (OffsetDeref obj
                  (ECompVal bl (MPtr cnm2 fldty)))
                se)
         (s, EX (SVar obj fld) se)
```

Figure 16: Dereferencing a Pointer-to-Member

variant in the model is that if a reference is made to a virtual function, then it must occur as the right-hand argument of a field-selection as a bare name (hence the `IDConstant F []...` above), and conversely that if a reference is not to a virtual function, then the field dereference must be to a completely

qualified identifier.[4]

The `derive_objid` function in the figure is used to derive object l-value information (*i.e.*, address, type and path) from values that may be both normal l-values or object r-values. Its definition is

```
(derive_objid (LVal a t p) = SOME (a,t,p)) ∧
(derive_objid (ConstructedVal b a cnm) =
      SOME(a,Class cnm,[cnm])) ∧
(derive_objid _ = NONE)
```

It is possible to have null member pointers (see [5, §4.11]). We specify `encode_offset` in such a way that the null member pointer constant is not in its range, and add the rule

```
(* RULE-ID: offset-deref-fails *)
    T
  ⇒
    mng (s, EX (OffsetDeref
                 obj
                 (ECompVal null_member_ptr (MPtr cnm2 fldty)))
               se)
        (s, EX UndefinedExpr se)
```

It is undefined behaviour to apply the null member pointer to any class.

Assignment The rules for assignment do not need to change from their presentation in C. Nonetheless, this semantics adopts Clive Feather's proposal [2] for handling the infamous language in the standard:

> *Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored.*

Rather than count references made on the RHS of an assignment, as in my thesis [6], the model now is that references can occur before updates, but not the other way round. This makes the rules for assignment considerably simpler, at the cost of requiring an analysis of all possible execution paths to see if any of them result in an update before a reference.

The rule for a completed assignment expression is presented in Figure 17. The `NONE` value that is the first argument ot `Assign` is where a binary

[4] Note the contrast with C: in C++, one can write `c.::B::fld`, which is unambiguous, if ugly, about which `fld` member is meant.

```
(* RULE-ID: assign-completes *)
    ¬class_type lhs_t ∧
    (nonclass_conversion s (v0,t0) (v,lhs_t) ∧ (* 5.17 p3 *)
     (se = add_se (a, v) se0) ∧ (resv = ECompVal v lhs_t)
                          ∨
     (∀v. ¬nonclass_conversion s (v0, t0) (v, lhs_t)) ∧
     (resv = UndefinedExpr) ∧
     (se = se0))
   ⇒
     mng (s, EX (Assign NONE (LVal a lhs_t [])
                             (ECompVal v0 t0)) se0)
         (s, EX resv se)
```

Figure 17: Rule for the Completion of an Assignment Expression

operator can be installed, implementing the `op=` syntax (`+=`, `>>=` etc.). The rule that deals with this is

```
(* RULE-ID: assign-op-assign *)
     T
   ⇒
     mng (s0, EX (Assign (SOME f) (LVal n t p) e) se0)
         (s0, EX (Assign NONE
                         (LVal n t p)
                         (CApBinary f (LVal n t p) e)) se0)
```

(Note that the hypothesis true (`T`) means there are no preconditions on this transition occurring.)

5.5 Statements in a Small-step Style

One might imagine that stating the statement part of a dynamic semantics in a small-step style should be easy. (My thesis [6, §7.1] explains why the way I formulated statements in a big-step style was a mistake.) The literature contains many examples of how to express constructs such as `while` and `if` in a small-step style. However, in C and C++ this is not as straightforward as one might think because of the need to prevent function bodies from interleaving.

Imagine a program such as that in Figure 18, and how one might evaluate the return-expression in `main`. If one simply expanded the bodies of

the called functions into the expression as the functions were ready to be called, one would be permitting the simultaneous evaluation of the bodies of f and g. But the C++ standard explicitly forbids this (§1.8 fn8), and the C standard also hints that it is forbidden.

```
int global;
int f(int x) { return global * 2 + x; }
int g(int y) { while (y > 0) { global += 2; y--; } }

int main(void) {
  global = 10;
  return f(6) + g(10);
}
```

Figure 18: Where Functions Must Not Interleave

One has to arrange the semantics so that expression evaluation can continue non-deterministically until a function call is encountered and the function call is made (after arguments have been evaluated). At this point, all further evolution of the program must occur within the function body, no matter how deeply nested the function call may have been within an enclosing expression. (This problem does not occur if statement evaluation is big-step because the hypothesis in the expression rule for a function call would be a statement rule that required the complete evaluation of the function body.)

At the base level, a function call turns an EX piece of syntax into an extended expression tagged with ST. This is shown in the rule *[function-call]* of Figure 19, governing function calls (calls to constructors are set up a little differently, see *[constructor-function-call]*).

The hypotheses in *[function-call]* are relatively involved. The first hypothesis is really a static matter: find_best_fnmatch is a placeholder for the process that calculates which function is actually to be called. This should really occur in Phase 1. The second hypothesis checks that if the function to be called is going to return a class r-value, then the rvrt argument to the FnApp_sqpt constructor needs to have been filled out by *[allocate-rvrt]*. Note also how the rvrt argument is pushed onto the rvstk component of the stack, so that the body of the function will know where to construct its result.

The third hypothesis calculates what the value for this should be in the body of the function being called. In other words, if the function is the mem-

```
(* RULE-ID: function-call *)
     find_best_fnmatch s0 fnid (MAP valuetype args) rtype
                       params body ∧
     ((rvrt = NONE) ⇒ ¬class_type (strip_const rtype)) ∧
     (case thisobj of
         SOME (LVal a (Class cnm) p) ->
            (SOME thisaddr = ptr_encode s0 a (Class cnm) p) ∧
            (thisval = SOME (ECompVal thisaddr
                                         (Ptr (Class cnm))))
      || _ -> (thisval = NONE)) ∧
     (pdecls = MAP (λ((n,ty),a).
                          VDecInit ty (Base n)
                             (CopyInit (EX (NoScope a) base_se)))
                   (ZIP (params, args)))
   ⇒
     mng (s0, EX (FnApp_sqpt rvrt
                             (FVal fnid ftype thisobj)
                             args)
                 se)
         (s0 with <|
            stack updated_by
                  (CONS (s0.env,s0.thisvalue,s0.allocmap,[]));
            rvstk updated_by (CONS rvrt);
            thisvalue := thisval;
            env := empty_env
          |>,
          ST (Block T pdecls [body]) (return_cont se rtype))
```

Figure 19: Making a Function Call

ber function of some class, the `thisobj` variable will be `SOME (LVal a t p)` giving the location of the object of the given class. Note that if this is a member function, resolution of OO-polymorphism through `virtual` functions will have already been resolved and the appropriate function will have been selected (see Section 5.7.2). Otherwise, if this is not a member function, the value for `this` will be `NONE`.

Finally, the last hypothesis sets up the initialization of the parameters by specifying the value of the variable `pdecls`. Each parameter is a new local variable, copy-initialized by the parameter value.

In the conclusion of the rule, see Section 5.8.2 for the details behind `return_cont`, and Section 5.5.2 for the operations applied to `s0`, setting up the new function's scope. The use of `CONS rvrt` to update the `rvstk` field pushes the `rvrt` value onto the front of the list.

Finally, note that when this transition is made the standard rule for evaluating an expression within a context (*[expr-econtext]*, p42) can not fire, because its hypothesis is of the form

```
mng (s0, EX e0 se0) (s, EX e se)
```

but our conclusion is of the form

```
mng (s0, EX e0 se0) (s, ST st c)
```

Instead, the new rule

```
(* RULE-ID: econtext-stmt *)
     mng (s0, EX e se0) (s, ST stmt c) ∧
     valid_econtext f
   ⇒
     mng (s0, EX (f e) se0) (s, ST stmt (cont_comp f c))
```

can fire, turning the enclosing expression into a statement form, with an ever more elaborate continuation. The rôle of the continuation is to do no more than record where the function-call was, so that when the statement form finishes execution, its result can be slotted back into the appropriate expression.

These rules have specified what happens when an expression evaluation switches to a statement evaluation. In the opposite direction, when a function call is about to return, one of the rules is

```
(* RULE-ID: return-rvalue *)
     is_null_se se0
   ⇒
     mng (s, ST (Ret (EX (ECompVal v t) se0)) (RVC c se))
         (s with rvstk updated_by TL, EX (c (ECompVal v t)) se)
```

In this situation, the `return` statement has completely evaluated its argument into a value, and there are no remaining side effects to be evaluated. This means that the value can be put back into the containing expression tree, and expression evaluation can continue. This is reflected by the switch from `ST` to `EX`, and the application of the continuation `c` to the value. The `TL` function "pops" the `rvstk`

Note how the argument to the return statement is itself an extended expression. This means that the argument will be tagged with `EX` initially, but may later evolve to be a statement and continuation (tagged with `ST`) if the return-expression includes a function call. This means that the rule for evaluation of the argument of return is

```
(* RULE-ID: return-eval-under *)
     mng (s1, exte0) (s2, exte)
   ⇒
     mng (s1, ST (Ret exte0) c) (s2, ST (Ret exte) c)
```

where `exte0` and `exte` may be statements or expressions.

Statement Evaluation Strategy The basic idea behind all of the rules for statements are that they should be evaluated until they yield a "final" form. As is explained further in Section 5.8.2 below, such an evaluation has to be done in the context of the continuation that is accompanying the statement. In essence, a final form is both one that can not be further evaluated, and also something that can be returned to a higher level as some sort of result. The definition of `final_stmt` is from HOL:`statements`:

```
(final_stmt EmptyStmt c = T) ∧
(final_stmt Break c = T) ∧
(final_stmt Cont c = T) ∧
(final_stmt (Ret e) c =
   case c of
      LVC f se0 -> (∃a t p se. (e = EX (LVal a t p) se) ∧
                              is_null_se se)
   || RVC f se0 -> final_value e ∧ no_class_lval e) ∧
(final_stmt (Throw exn) c = ∃e. (exn = SOME e) ∧
                               final_value e) ∧
(final_stmt _ c = F)
```

The `Throw` form implements exceptions (for which, see Section 5.6); most statements will evaluate to either an `EmptyStmt`, or a `Ret`.

5.5.1 Simple Statements

The rules for the simplest statement forms, expression statements (using the `Standalone` "constructor") and if statements are given in Figure 20. While an expression statement explicitly terminates (yielding an `EmptyStmt`), the if statement just evolves into one of its two branches.

Loops The loop forms in C++ are modelled just as they were in my thesis [6, §3.4.5], as syntactic sugar for forms involving one loop and different arrangements of the `Trap`, `continue` and `break` primitives. This allows just one, unconditional rule for all loops:

```
(* RULE-ID: loop *)
     T
   ⇒
     mng (s, ST (CLoop guard bdy) c)
         (s, ST (CIf guard (Block F [] [bdy; CLoop guard bdy])
                           EmptyStmt) c)
```

Traps, and Loop Interruptions Again, following my thesis, the rules for traps are straightforward, though numerous because of the different possible combinations of `continue` (written `Cont`) and `break` (written `Break`). The simplest rules for these forms are presented in Figure 21. The rule for the way in which `break`, `continue`, `return` and exceptions all cause flow of control to alter within a block is presented in Section 5.5.2 below.

Declarations in Guards C++ allows variables to be declared in the guard positions of loop forms and if statements. This is a syntactic nicety that we can assume has been compiled away. For example, something like

```
if (int i = e) { ... }
```

can be rewritten into

```
{ int i = e; if (i) { ... } }
```

I take a similar attitude to C++'s relaxation of C's rule that declarations and statements can intermingle. Such an intermingling can be rewritten to successive nested blocks, all of which respect the basic C rule that a block is a sequence of declarations followed by a sequence of statements.

```
(* RULE-ID: standalone-evaluates *)
     mng (s1, exte) (s2, exte')
   ⇒
     mng (s1, ST (Standalone exte) c)
         (s2, ST (Standalone exte') c)

(* RULE-ID: standalone-finishes *)
     is_null_se se ∧ final_value e ∧
     (s.stack = (env,thisv,amap,[]) :: rest)
   ⇒
     mng (s, ST (Standalone e) c)
         (s with <| stack := rest; allocmap := amap |>,
          ST EmptyStmt c)


(* RULE-ID: if-eval-guard *)
     mng (s0, RVR guard) (s, RVR guard')
   ⇒
     mng (s0, ST (CIf guard t e) c)
         (s, ST (CIf guard' t e) c)


(* RULE-ID: if-true *)
     scalar_type t ∧ is_null_se se ∧ ¬is_zero t v ∧
     (s.stack = (env,thisv,amap,[]) :: rest)
   ⇒
     mng (s, ST (CIf (EX (ECompVal v t) se) thenstmt elsestmt) c)
         (s with <| stack := rest; allocmap := amap |>,
          ST thenstmt c)


(* RULE-ID: if-false *)
     scalar_type t ∧ is_null_se se ∧ is_zero t v ∧
     (s.stack = (env,thisv,amap,[]) :: rest)
   ⇒
     mng (s, ST (CIf (EX (ECompVal v t) se) thenstmt elsestmt) c)
         (s with <| stack := rest; allocmap := amap |>,
          ST elsestmt c)
```

Figure 20: Statement Rules for Expression and If Statements

```
(* RULE-ID: trap-stmt-evaluation *)
     mng (s0, ST st c) (s, ST st’ c)
   ⇒
     mng (s0, ST (Trap tt st) c) (s, ST (Trap tt st’) c)

(* RULE-ID: trap-break-catches *)
     T
   ⇒
     mng (s, ST (Trap BreakTrap Break) c) (s, ST EmptyStmt c)

(* RULE-ID: trap-continue-catches *)
     T
   ⇒
     mng (s0, ST (Trap ContTrap Cont) c) (s0, ST EmptyStmt c)

(* RULE-ID: trap-continue-passes-break *)
     T
   ⇒
     mng (s, ST (Trap ContTrap Break) c) (s, ST Break c)

(* RULE-ID: trap-break-passes-continue *)
     T
   ⇒
     mng (s, ST (Trap BreakTrap Cont) c) (s, ST Cont c)

(* RULE-ID: trap-emptystmt-passes *)
     T
   ⇒
     mng (s0, ST (Trap tt EmptyStmt) c) (s0, ST EmptyStmt c)

(* RULE-ID: trap-ret-passes *)
     final_value e
   ⇒
     mng (s, ST (Trap tt (Ret e)) c) (s, ST (Ret e) c)
```

Figure 21: Dynamic Rules for Loop Interruptions and Traps

5.5.2 Statements in Blocks

The compound statement is the basic syntactic structure expressing scope at the statement level. Its manifestation in this model is as the constant `Block`, with type

```
: bool -> var_decl list -> CStmt list -> CStmt
```

The initial boolean field is used to record whether or not a block has been entered. It starts as false (`F`). The rule for entering a block is

```
(* RULE-ID: block-entry *)
    T
  ⇒
    mng (s, ST (Block F vds sts) c)
        (s with stack updated_by
                      (CONS (s.env, s.thisvalue, s.allocmap, [])),
         ST (Block T vds sts) c)
```

The field `stack` is a stack of local environments, a value for the `this` pointer, a value recording the allocated memory map, and a list of class-objects that have been allocated at this allocation level, and will need destroying later. When a block is entered, the old value for the environment needs to be stored so that it can be restored on block exit. Recall that the field `env` stores all of the information about *local* entities. Information on non-local entities is all recorded in the `genv` field.

After a block is entered, its variable declarations must be executed. This is handled by the rule *[block-declmng]*:

```
(* RULE-ID: block-declmng *)
    declmng mng (d0, s0) (ds, s)
  ⇒
    mng (s0, ST (Block T (d0 :: vds) sts) c)
        (s, ST (Block T (ds ++ vds) sts) c)
```

The auxiliary `declmng` (which takes `mng` as a parameter, allowing it to recurse back to it) has type

```
: ((state # ExtE) -> (state # ExtE) -> bool) ->
  (var_decl # state) -> (var_decl list # state) -> bool
```

The “return” of a list of new variable declarations allows termination to be indicated (by returning the empty list), and also allows complicated object constructions, which involve multiple calls to initialize sub-objects, done

through special forms of variable declaration. There is more on the simple forms of declmng in Section 5.5.3 below.

When the declaration list is exhausted, execution of a block's body of statements can proceed. The congruence rule for a block is *[block-stmt-evaluate]*:

```
(* RULE-ID: block-stmt-evaluate *)
     mng (s0, ST st c) (s, ST st' c)
   ⇒
     mng (s0, ST (Block T [] (st :: sts)) c)
         (s, ST (Block T [] (st' :: sts)) c)
```

Note the form of Block required: it must have no further variable declarations to evaluate, and must have been entered (has its boolean flag set to T).

If the first statement inside a Block is, or becomes, an EmptyStmt, and there are more statements beyond it to evaluate, then the EmptyStmt can be discarded:

```
(* RULE-ID: block-emptystmt *)
     ¬(sts = [])
   ⇒
     mng (s, ST (Block T [] (EmptyStmt::sts)) c)
         (s, ST (Block T [] sts) c)
```

Alternatively, if the head statement is an exception or interruption form, and it is followed by other statements, then it causes all following statements to disappear:

```
(* RULE-ID: block-interrupted *)
     final_stmt exstmt c ∧
     ¬(exstmt = EmptyStmt) ∧
     ¬(sts = [])
   ⇒
     mng (s, ST (Block T [] (exstmt::sts)) c)
         (s, ST (Block T [] [exstmt]) c)
```

Then, when a final statement is the only thing remaining in a block, the block itself can exit, clearing its local stack frame in the environment, and propagating the final statement upwards. This gives us *[block-exit]*, in Figure 22. By requiring that the last component of the 4-tuple at the head of the stack component of the state be empty, we require that all local objects have had their destructors called (for more on this, see Section 5.10). If this condition is met, the various stacks can be popped, and the final statement

```
(* RULE-ID: block-exit *)
     (s.stack = (env,this,amap,[]) :: stk') ∧
     final_stmt st c ∧
   ⇒
     mng (s, ST (Block T [] [st]) c)
         (s with <| stack := stk';
                    allocmap := amap;
                    env := env;
                    thisvalue := this |>,
          ST st c)
```

Figure 22: Exiting a Block

lifted up a level. In this way, a return statement deep within multiple blocks can eventually make its way to the top level, where its value can be passed to the continuation through rules *[return-rvalue]* (page 54) or *[return-lvalue]* (page 85).

Such a return can not happen prematurely: the two `return` rules both have `EX` tags in their "result" arguments, and the rule *[block-stmt-evaluate]*, as well as all the other rules calling for statement evaluation recursively, requires its recursive call to be an `ST`-to-`ST` evaluation.

5.5.3 Simple Declarations

This section discusses simple declaration forms, which are treated in a way that is similar to the treatment in my thesis. Declarations involving class types are typically not simple, and are discussed in Sections 5.7 (basic object orientiation) and 5.10 (object lifetimes). There are two simple forms of declaration: a bare declaration of a variable, such as

```
int x;
```

or a declaration coupled with an initialization

```
int x = 3;
```

These two different forms are represented with the constructors `VDec` and `VDecInit` respectively, both constructing values in the `var_decl` type.

As already mentioned, the C++ mechanisation uses an auxiliary relation `declmng` to do most variable declaration work. This relation is defined in HOL:`declaration_dynamics`. Its simplest rule is

```
(* RULE-ID: decl-vdec-nonclass *)
     vdeclare s0 ty name s ∧
     object_type ty ∧
     ¬class_type (strip_array ty)
   ⇒
     declmng mng (VDec ty name, s0) ([], s)
```

This rule states that if one is declaring a variable of non-class type (and which is not an array of a class type), then it suffices to allocate space for it through the `vdeclare` relation, storing its address in the appropriate part of the state's environment, and also recording the type. The empty list in the second argument to `declmng` signals that no further work needs to be done for this declaration.

When a variable is to be initialized, this can occur as a direct initialization:

```
int x(3);
```

or as a copy-initialiation:

```
inx x = 3;
```

For non-classes this syntactic difference makes no difference in behaviour. The first rule is for the direct initialization (`DirectInit0` form), where the transition is immediately to a copy-initialization:

```
(* RULE-ID: decl-vdecinit-start-evaluate-direct-nonclass *)
     ~class_type ty ∧
     vdeclare s0 ty name s ∧
     (SOME (a,pth) = lookup_addr s name) ∧
     (loc = if ref_type ty then RefPlace NONE name
              else ObjPlace a)
   ⇒
     declmng mng
              (VDecInit ty name (DirectInit0 [arg]), s0)
              ([VDecInitA ty loc (CopyInit (EX arg base_se))], s)
```

The transition is to the `VDecInitA` form, which records where the initialization is to be performed. The `vdeclare` relation is used, as before, to allocate space for the new object, and subsequent steps in the evaluation of the declaration will fill this space in. (In the case of a reference, `vdeclare` will not allocate any space, but will record a type for the new name.)

The rule for the `CopyInit` form is similar: there is a transition to the `VDecInitA` form and a call to `vdeclare`. (As it happens this rule can be applied for the declaration and initialization of class objects too.)

```
(* RULE-ID: decl-init-start-eval-copy *)
     vdeclare s0 ty name s ∧
     (SOME (a,pth) = lookup_addr s name) ∧
     (loc = if ref_type ty then RefPlace NONE name
            else ObjPlace a)
   ⇒
     declmng mng
             (VDecInit ty name (CopyInit arg), s0)
             ([VDecInitA ty loc (CopyInit arg)], s)
```

Once a `VDecInitA` form has been achieved, the initializing expression can be evaluated.

```
(* RULE-ID: decl-vdecinit-evaluation *)
     mng (s0, exte) (s, exte') ∧
     ((f = CopyInit) ∨ (f = DirectInit))
   ⇒
     declmng mng (VDecInitA ty loc (f exte), s0)
                 ([VDecInitA ty loc (f exte')], s)
```

Note that the initializer form `f` can only be a `DirectInit` when initializing classes. All non-class objects will be a `CopyInit` form by this point.

When initializing a non-reference, expressions that yield l-values must be allowed to undergo the "l-value to r-value" conversion:

```
(* RULE-ID: decl-vdecinit-lval2rval *)
     lval2rval (s0,e0,se0) (s,e,se) ∧
     ¬ref_type ty ∧
     ((f = CopyInit) ∧ ¬class_type (strip_const ty) ∨
      (f = DirectInit))
   ⇒
     declmng mng (VDecInitA ty loc (f (EX e0 se0)), s0)
                 ([VDecInitA ty loc (f (EX e se))], s)
```

The extra condition on the `CopyInit` case reflects the fact that there may be a copy-constructor for the class type that requires a reference type as a parameter.

Finishing an Initialization When a non-class, non-reference initialization is ready, the rule *[decl-vdecinit-finish]* of Figure 23 will apply. The value of the initializing expression is first converted to be of the appropriate type, and the resulting value (`v'`, a list of bytes), is copied into memory using the `val2mem` relation. Note that the state's `initmap` is also initialized in the

```
(* RULE-ID: decl-vdecinit-finish *)
     (e = ECompVal v ty) ∧
     nonclass_conversion s0 (v,ty) (v',dty) ∧
     is_null_se se ∧
     ~class_type dty ∧
     (s = val2mem (s0 with initmap updated_by (UNION) rs) a v') ∧
     (rs = range_set a (LENGTH v')) ∧
     ((f = CopyInit) ∨ (f = DirectInit)) ∧
     (s.stack = (env,thisv,amap,[]) :: rest)
   ⇒
     declmng mng
         (VDecInitA dty (ObjPlace a) (f (EX e se)), s0)
         ([], s with <| stack := rest; allocmap := amap |>)
```

Figure 23: Completing the Initialization of a Non-class, Non-reference Variable

process. The fact that the location attached to the `VDecInitA` constructor is an `ObjPlace` ensures that the variable being initialized is not a reference.

5.6 Exceptions

Exceptions are modelled in a way similar to the treatment of `return`, `break` and `continue`. One difference is that exceptions propagate further: the `return` "value" only propagates up as far as a function call (within an expression). In contrast, an exception will continue to propagate up through the call-stack until it hits a suitable handler.

This much allows a preliminary sketch of the behaviour. The `throw` form is actually an expression (`EThrow`), but we set things up so that there is a statement-level version of `throw` as well (`Throw`), and it will be this that propagates through statement syntax. The rule *[expression-throw-some]* describes the behaviour when `EThrow` has an argument:

```
(* RULE-ID: expression-throw-some *)
     T
   ⇒
     mng (s, EX (EThrow (SOME e)) se)
         (s, ST (Throw (SOME (EX e se))) c)
```

The variable `c` represents the continuation that would normally convert the result of the statement into a value to be inserted into a containing expres-

sion tree. Because thrown values can't ever turn into values until they initialize a handler, this c can be anything at all. Because throw-expressions are of type `void` ([5, §15,p1]), they will never evaluate while a sub-expression of some containing expression.

At the statement level, the `Throw` form takes an extended expression as an argument. This evaluates its argument as one might expect (rule *[throw-expr-evaluation]*)

```
(* RULE-ID: throw-expr-evaluation *)
     mng (s0, RVR e0) (s, RVR e)
   ⇒
     mng (s0, ST (Throw (SOME e0)) c) (s, ST (Throw (SOME e)) c)
```

When a `throw`'s expression has been completely evaluated, we have something that can then propagate upwards through the abstract syntax of statements.

We already have a rule specifying how `throw`-statements traverse `Block` values: *[block-interrupted]*, repeated here:

```
(* RULE-ID: block-interrupted *)
     final_stmt exstmt c ∧
     ¬(exstmt = EmptyStmt) ∧
     ¬(sts = [])
   ⇒
     mng (s, ST (Block T [] (exstmt::sts)) c)
         (s, ST (Block T [] [exstmt]) c)
```

The predicate `final_stmt` is true of `throw` and `return` statements with fully evaluated arguments, as well as of `break`, `continue` and the EmptyStmt form. The latter doesn't cause an interruption, so is excluded by the second hypothesis to the rule. The final hypothesis ensures that there isn't an infinite loop on this rule. The same predicate is used in the rule for exiting a block (*[block-exit]*, page 61).

There is also rule *[trap-exn-passes]* for exception statements escaping the `Trap` form (which is used for handling `continue` and `break`):

```
(* RULE-ID: trap-exn-passes *)
     exception_stmt exn
   ⇒
     mng (s, ST (Trap tt exn) c) (s, ST exn c)
```

Because exceptions arise from expressions, the statement level rules need to acknowledge this possibility. Thus, this rule for `if` (*[if-exception]*):

```
(* RULE-ID: if-exception *)
     is_exnval guard ∧
     (s.stack = (env,thisv,ampa,[]) :: rest)
   ⇒
     mng (s, ST (CIf guard thenstmt elsestmt) c)
         (s with <| stack := rest; allocmap := amap |>,
          mk_exn guard c)
```

where the definition of `is_exnval` is (from HOL:`statements`)

```
(is_exnval (ST (Throw (SOME e)) c) = final_value e) ∧
(is_exnval _ = F)
```

The function `mk_exn` takes an exception value and replaces its continuation information with something appropriate for the level of the containing statement:

```
mk_exn (ST (Throw (SOME e)) c0) c = ST (Throw (SOME e)) c
```

Of course, exceptions can arise in all the other expressions that appear within statement forms, so there are similar rules for the standalone expression and `return` forms, as well as for the statement-level `throw` form itself. (The rule *[expression-throw-some]* turns an `EThrow` into a `Throw` statement immediately, without evaluating the argument. When the argument *is* evaluated, it may itself cause an exception.)

Because exceptions can arise in variable declarations, there is also a rule for handling these. This is *[block-declmng-exception]*:

```
(* RULE-ID: block-declmng-exception *)
     ((f = CopyInit) ∨ (f = DirectInit)) ∧
     declmng mng (d0, s0) ([VDecInitA ty loc (f e)], s) ∧
     is_exnval e ∧
     (e = ST (Throw (SOME ex)) c')
   ⇒
     mng (s0, ST (Block T (d0 :: vds) sts) c)
         (s, ST (Block T [] [Throw (SOME ex)]) c)
```

Again, note how the continuation initially associated with the exception (`c'`) is ignored.

5.6.1 Handling Exceptions

Handling exceptions is done with the `try-catch` form, which is a sequence of handlers associated with a statement that might raise an exception. In the concrete syntax, programmers write something like

```
try {
  statement*
}
handler+
```

where a *handler* is of the form

```
catch (guard) { statement* }
```

and a *guard* can be "..." (*i.e.*, three full-stops), a type, or a standard parameter declaration (associating a type with a name).

At the abstract syntax level, this is captured by the following HOL declarations (in statementsScript):

```
exn_pdecl = (string option # CPP_Type) option

stmt = ...
     | Catch of CStmt => (exn_pdecl # CStmt) list
```

Statements can evaluate as usual under a Catch *[catch-stmt-evaluation]*:

```
(* RULE-ID: catch-stmt-evaluation *)
    mng (s0, ST st c) (s, ST st' c)
  ⇒
    mng (s0, ST (Catch st hnds) c) (s, ST (Catch st' hnds) c)
```

Non-exception statements pass through Catch statements, ignoring the handlers *[catch-normal-stmt-passes]*:

```
(* RULE-ID: catch-normal-stmt-passes *)
    final_stmt st c ∧
    ¬exception_stmt st
  ⇒
    mng (s0, ST (Catch st hnds) c) (s0, ST st c)
```

There are three rules governing how handlers interact with thrown exceptions. The first describes the behaviour when the handler parameter is given as "..." *[catch-all]*:

```
(* RULE-ID: catch-all *)
    (exn = SOME (EX e base_se))
  ⇒
    mng (s0, ST (Catch (Throw exn) ((NONE, hnd_body) :: rest)) c)
        (s0 with current_exns updated_by (CONS e),
         ST (Block F [] [hnd_body; ClearExn]) c)
```

This rule introduces two new features, the current_exns field of the program state, and the ClearExn statement-form. Both are present to support the ability of programs to use throw without an argument to re-throw the "current exception". This is covered below in Section 5.6.2.

Otherwise, the behaviour is clear: if the top handler has "..." as its parameter, then this handler is entered (and the other handlers are discarded).

When the top handler has an explicitly-typed parameter, the handler is only entered if the type of the thrown value matches: *[catch-specific-type-matches]* in Figure 24.

```
(* RULE-ID: catch-specific-type-matches *)
     (exn = SOME (EX e base_se)) ∧
     exception_parameter_match s0 pty (value_type e) ∧
     (pname = case pnameopt of SOME s -> Base s
                            || NONE -> (Base " no name "))
   ⇒
     mng (s0, ST (Catch
                   (Throw exn)
                   ((SOME(pnameopt, pty), hnd_body) :: rest))
                 c)
         (s0 with current_exns updated_by (CONS e),
          ST (Block F [VDecInit pty pname
                                  (CopyInit (EX e base_se))]
                      [hnd_body; ClearExn]) c)
```

Figure 24: Catching a Typed Exception

The string " no name " is chosen arbitrarily as the name of the invisible temporary if the handler has just a type as its parameter and no associated name. This name is chosen so as to not mask any existing names in scope (no legal C++ program can have variable names that include spaces).

If the declared type pty matches the type of the exception value, then the exception value copy-initializes the parameter, and the handler body is executed. The constant exception_parameter_match checks the match, embodying the rules in [5, §15.3, paragraph 3]. If there is no match, then the remaining handlers have to be tried *[catch-specific-type-nomatch]*:

```
(* RULE-ID: catch-specific-type-nomatch *)
     (exn = SOME (EX e base_se)) ∧
     ¬exception_parameter_match s0 pty (value_type e)
   ⇒
     mng (s0, ST (Catch
                    (Throw exn)
                    ((SOME(pnameopt, pty), hnd_body) :: rest))
                 c)
         (s0, ST (Catch (Throw exn) rest) c)
```

If no handlers, remain, the exception propagates further *[catch-stmt-empty-hnds]* (generalised to allow any statements to pass through):

```
(* RULE-ID: catch-stmt-empty-hnds *)
     T
   ⇒
     mng (s0, ST (Catch st []) c) (s0, ST st c)
```

5.6.2 Using `throw` with No Argument

If flow of control is within an exception handler, or within a function body that has been called from such, then it is permissible to use the expression `throw` without any arguments to cause the current exception to be rethrown. This requires the model to track what the current handled exception is. More, the standard requires the state to track the notion of "most recently caught" exception [5, §15.1, paragraph 7]), which requires the state to track a stack of exceptions that have been caught.

The expression version `EThrow` is converted to the statement form as soon as it is encountered *[expression-throw-none]*:

```
(* RULE-ID: expression-throw-none *)
     T
   ⇒
     mng (s, EX (EThrow NONE) se) (s, ST (Throw NONE) c)
```

(The choice of c is again irrelevant.)

There are then two rules for the statement form `Throw NONE`. If there is a current exception, all is well *[bare-throw-succeeds]*:

```
(* RULE-ID: bare-throw-succeeds *)
     (s0.current_exns = e::es)
   ⇒
     mng (s0, ST (Throw NONE) c)
         (s0 with current_exns := es,
          ST (Throw (SOME (EX e base_se))) c)
```

Otherwise, the program must call the `::std::terminate` function

```
(* RULE-ID: bare-throw-fails *)
     (s0.current_exns = [])
   ⇒
     mng (s0, ST (Throw NONE) ct)
         (s0, ST (Standalone (EX callterminate base_se)) ct)
```

where the special expression `callterminate` is defined to be

```
   FnApp (Var (IDConstant T [IDName "std"]
                            (IDName "terminate")))
         []
```

Above, the rules for handlers also use a statement-form `ClearExn`. This special value has the following rule *[clear-exn]*:

```
(* RULE-ID: clear-exn *)
     (s0.current_exns = e::es)
   ⇒
     mng (s0, ST ClearExn c)
         (s0 with current_exns := es, ST EmptyStmt c)
```

This ensures that when a handler finishes the most recently caught exception is no longer recorded as such. If a handler rethrows the current exception, or throws a fresh exception of its own, and this exception escapes the handler, then the flow of control will never reach the ClearExn, and this rule will not fire.

5.6.3 Exception Specifications

The standard's §15.4 specifies a method whereby functions can specify which exception types they will produce. If an unexpected exception value occurs, this results in a call to `std::unexpected`. This is not modelled in the dynamic semantics as it can be emulated with a compile-time rewriting of the program. If a function `f` is specified to only raise exceptions X and Y, then it can be rewritten to be

```
f(args)
{
  try {
    body
  }
  catch (X) { throw; }
  catch (Y) { throw; }
  catch (...) { std::unexpected(); }
}
```

5.6.4 Exceptions and Object Lifetimes

Exceptions complicate the story about the construction and destruction of objects. When a constructor runs it will typically cause a sequence of sub-objects to also be constructed. If at any stage, an exception is raised during this process, then those objects that have been constructed need to have their destructors called, but naturally, those that haven't yet been constructed shouldn't have destructors called.

Consider for example

```
Class::Class(objty p1) : b1(3), b2(e) { body }
```

There are three objects that have their constructors called as a result of a call to this constructor. One is the parameter `p1`, and the other are the base classes (or data-members) `b1` and `b2`. When this constructor is called, `p1` is always eventually destroyed, but (sub-)objects `b1` and `b2` should live on, unless *body* or e cause an exception to be raised.

To get this situation to work in the model, the state's `stack` field records object creations *twice*. All object creations are associated with a particular allocation level or depth of the `stack`. One creation is recorded at the top of the stack, and the other is recorded at the (lower) level where the enclosing object is being created. If an exception is being propagated out of a block, the destructor is called for objects when they first appear (higher/later in the stack). Otherwise, objects that have creations recorded deeper in the stack (which will happen if they are sub-objects), don't get their destructors called until that level is topmost.

For more on this, see Section 5.10, and the definition of `realise_-destructor_calls` in HOL:`declaration_dynamics`.

5.7 Basic Object-Orientation

The inspiration for this part of the semantics is the article by Wasserrab *et al* [15], which provides a detailed model for multiple inheritance in a simple C++-like language.

5.7.1 Class Declarations

A class declaration is similar to the original C model's declaration of a `struct` type. A class declaration takes two parameters, the name of the class, and an optional "class-info" argument.[5] The class-info, if present, is a list of fields, coupled with a list of ancestors. The latter allows inheritance from zero or many ancestors. Each ancestor is coupled with a boolean flag indicating whether or not it is a `virtual` ancestor. As the model ignores protection issues, there is no scope for indicating protection status for ancestors.

The fields are of two sorts, declarations of members (whether of "data" fields or member functions), and nested classes. Member declarations are accompanied by a flag indicating whether or not they are static, and a protection indicator (i.e., `public`, `protected`, or `private`). Again, protection information is entirely ignored: any field access is assumed to have been OK-ed earlier by the compiler.

Member function definitions give the function's name, return-type, parameter list (types and names), function body, and a flag indicating whether or not it is virtual. (Of course, even in the absence of an explicit declaration that a member function is virtual, it may be so because of an ancestor's prior declaration.)

When a class declaration is encountered , its member functions are entered into the state's function map. The same function map is used for normal (non-class) functions, but the structured nature of C++ identifiers allows the model to distinguish both sorts of function.

5.7.2 Class L-Values and Member Functions

Classes can not be converted into r-values as other values can. This is because of the problems that arise with multiple inheritance. In particular, with multiple inheritance in place, it is no longer true that one can extract

[5] The information argument is optional to allow the situation where a forward declaration of a class occurs.

the byte sequence for a given l-value by starting at the base address and taking as many bytes from memory as the size of the type. In particular, virtual base classes may be at completely different places in memory, not necessarily even contiguous with the rest of the object. (This is demonstrated for the g++ compiler by the little program in `notes/mult-inheritance-layout.cpp`.) The special way in which class r-values are handled is described in Section 5.11.

The presence of classes means that the model's presentation of l-values changes from the way it was in the original C model. In particular, an l-value that is statically typed as a base class needs to know dynamically that it is really of a derived type. This information is traditionally recorded in `vtable` fields. Following Wasserrab *et al*, my model instead records an additional path accompanying every l-value. This path is a list of class names (values of type `CPP_ID`), listing the path through the ancestry-tree that leads from the most-derived class to the static class type. Moreover, the type that accompanies every l-value (recall that the `LVal` constructor takes three arguments, an address, a type and a list of identifiers), will have as the type, the *dynamic* type of the l-value.

Consider, for example, the code in Figure 25. The body of function `g` constructs the l-value `*b` when it calls `f`. As in the C model, this l-value will be associated with some address, and the static type, which is `Class B`. The actual l-value will be of the form

```
LVal a (Class D2) [D2,D1,B]
```

The last element of the paths in l-values is always the static class of the value.

In Figure 25, the expression of interest is `b->f()`, which is syntactic sugar for `((*b).f)()`. (Note how the member selection is syntactically subordinate to the function application.) The *[virtual-fn-member-select]* rule (see Figure 26) governs the evaluation of `(*b).f` once the left-hand-side (`*b`) has evaluated to an l-value.[6]

In our model, the l-value's address will be the same as the address of its most-derived class. In other words, the `a` of the rule will be the same as the address of the object `d`. This is not what happens in typical compilers, which will actually make the pointer to the `B` sub-class point at the address of that sub-object's fields in the wider object's memory layout. Then, the fact that

[6] Even when the model allows for class r-values, they will be given a memory location (and thus, a life-time). This will enable them to also be l-values. In essence, it is not possible to create an object of any sort without giving it a location. Contrast numbers, which can be "created", and not given a memory location, simply by writing them down.

```
class B {
  public: virtual int f() { return 3; }
};

class D1 : public B {
  public: int f() { return 4; }
};

class D2 : public D1 { };

int g(class B *b) { return b->f(); }

int main()
{
  D2 d;
  return g(&d);
}
```

Figure 25: C++ code demonstrating OO-polymorphism. The program will return 4. Though it appears as if `B`'s function `f` is called, the version of `f` called will actually be that attached to `D1`.

```
(* RULE-ID: virtual-fn-member-select *)
     (derive_objid obj = SOME (a, Class C, Cs)) ∧
     (s,{}) |- LAST Cs has least method
                 (IDName fld) -:
                 (static_retty,F,args0,body0) via
                 Ds ∧
     (s,{}) |- (C,Cs ^ Ds) selects
                 (IDName fld) -:
                 (dyn_retty,F,args,body) via
                 Cs'
   ⇒
     mng (s, EX (SVar obj (IDConstant F [] (IDName fld))) se)
         (s, EX (FVal (mk_member (LAST Cs') (IDName fld))
                      (Function dyn_retty (MAP SND args))
                      (SOME (LVal a (Class C) Cs'))) se)
```

Figure 26: The Rule for Virtual Member Function Dispatch

the most-derived class is a D2 is implicitly recorded in the vtable, which will contain a pointer to D1::f.

In the rule, the variable C containing the name of the static class will be D2, and Cs, the path variable, will be [D2,D1,B]. The fld variable will be f. Then, the first hypothesis will check the class hierarchy to determine where an f can be found, starting at the static type, i.e., at B. This will reveal that, with respect to B there is an f at path [B]. This will be the value for Ds. The same check determines four pieces of information about the function: its static return type (in the variable static_retty, that it is not static (virtual functions can not be static), and what its arguments and body are (args0, and Body0).

The second hypothesis calculates which function must be called given the class's dynamic type. This is complicated because of the need to handle multiple inheritance (the definition of "selects-via" is discussed below in Section 5.9), but in this case, the path Cs' will be found to be [D2,D1], and the information (dyn_retty,F,args,body) found will be that for the function D1::f.

The result of the rule in this circumstance is that the expression b->f turns into a reference to the function D1::f (this is the call to mk_member), coupled with with the fact that the function call is being made on an object

whose dynamic type is D2, and whose static type is D1 (as is appropriate for the body of the function).

Note that the function in this rule is known to be virtual by virtue of the fact that the fieldname is unqualified: it is of the form

```
IDConstant F [] (IDName fld)
```

Name resolution will ensure that non-virtual members all have qualified names.

Other Sorts of Member Function There are two other sorts of member functions in C++: static member functions, and normal functions. Each sort has its own rule in the semantics. First, the rule for static member functions:

```
(* RULE-ID: static-fn-member-select *)
     (derive_objid obj = SOME (a, Class cnm,  Cs)) ∧
     (s,{}) |- path (LAST Cs) to class_part fldid via Ds ∧
     (s,{}) |- LAST Ds has least method
                  (IDtl fldid) -: (retty,T,ps,body)
                  via [LAST Ds] ∧
     (ftype = Function retty (MAP SND ps)) ∧
     is_qualified fldid
   ⇒
     mng (s, EX (SVar obj fldid) se)
         (s, EX (FVal fldid ftype NONE) se)
```

The third hypothesis (is_qualified) checks that the field name has multiple components to it. This qualification will be introduced by name resolution if not already provided by the user. Name resolution will also ensure that the name given to the field is qualified with the name of the class where the field actually resides.

Strictly, in this rule, the first hypothesis is redundant because of this preprocessing: we know that the field will occur in the class that is at the end of the path Ds, and that this final element will be equal to the class-part of the identifier. The second hypothesis reinforces this: the deduced path from LAST Ds (which equals class_part fldid) to the host class for the given field is the singleton [LAST Ds]. Because the member is static (witness the T as argument 2 of the 4-tuple returned by least-method), the final function value does not include a class component.

The rule for normal, non-static member functions is given in Figure 27. This is very similar to the rule for static functions, but this time the identity

```
(* RULE-ID: nstatic-fn-member-select *)
     (derive_objid obj = SOME (a, Class cnm, Cs)) ∧
     (s,{}) |- path (LAST Cs) to class_part fldid via Ds ∧
     (s,{}) |- LAST Ds has least method
                  (IDtl fldid) -: (retty,F,ps,body)
                  via [LAST Ds] ∧
     (ftype = Function retty (MAP SND ps)) ∧
     is_qualified fldid
   ⇒
     mng (s, EX (SVar obj fldid) se)
         (s, EX (FVal fldid ftype
                      (SOME (LVal a (Class cnm) (Cs ^ Ds))))
               se)
```

Figure 27: Selecting a Non-static Member Function

of the class for which the function is being called is important, and recorded in the third argument to `FVal`. The dynamic type of the object stays the same, but the static type is adjusted to reflect the class where the function being called is defined.

Calling a Member Function Once the name of the function to be called has been found, the step of actually entering the body of the function can be taken. This is described in rule *[function-call]*, already presented in Figure 19.When a member function is called, the `FVal` constructor stores the address and type of the object that the function is a member of. The rule extracts this information to initialize the `this` value used in the body of the member function.

Field Selection Field selection is also based on the notion of being able to find the most-derived declaration of the given field in the ancestor hierarchy. There is no need to worry about adjusting `this` pointers, or performing analyses with dynamic types as field selections are always done with respect to a class's static type. However, there is an additional complexity, stemming from the need to give addresses to selected fields, so that they can become well-formed l-values. In turn, this relies on knowing how a class is laid out in memory.

The standard *does* require that fields belonging to a particular class type

```
(* RULE-ID: nstatic-data-field-select *)
     (s,{}) |- path (LAST p) to cnm2 via p' ∧
     (s,{}) |- LAST p' has least
                 (IDtl fldid) -: (ty, F)  (* F = non-static *)
                 via [LAST p'] ∧
     object_type ty ∧
     (mdp = (cnm1 = cnm2) ∧ (p = [cnm1])) ∧
     is_qualified fldid ∧
     (class_part fldid = cnm2) ∧
     (SOME offn = lookup_offset s mdp fldid)
   ⇒
     mng (s, EX (SVar (LVal a (Class cnm1) p) fldid) se)
         (s, EX (LVal
                  (a + subobj_offset s (cnm1, p ^ p') + offn)
                  ty
                  (SND (default_path ty))) se)
```

Figure 28: Calculating the Offset of a Non-static Data Member

be laid out in the order in which they appear.[7] But there is no specification of how base sub-objects are laid out. (Recall, moreover, that in the presence of a virtual base-class, an object that is not most-derived may be split over distinct parts of memory.)

The rule for finding the offset of a non-static data member is given in Figure 28. Recall that static analysis done by Phase 1 will have turned all data field references into fully-qualified identifiers. That means the dynamics already knows exactly which sub-class holds the desired field. In the rule, that sub-class is `cnm2`. The first hypothesis confirms that there is indeed a path from the static type of the l-value to that sub-class (which might be itself, of course). The second hypothesis then both extracts the type of the field and confirms that it is not static.

The rule also determines whether or not the sub-class is actually the most-derived class, and records this in the boolean variable `mdp`. This is done so that the underspecified calculation in `lookup_offset` can return a different value for field offsets depending on whether or not a class is most-derived. (It seems very unlikely that an implementation would do this; most would put any virtual bases at the end of their internal layout, meaning that

[7] The order is actually required to hold as long as they have the members have the same access-specifiers, but the model doesn't handle accessibility.

all other offset calculations could proceed undisturbed.)

The variable `offn` is the offset of the field within its host class. The function `subobj_offset` is used to calculate the offset of the sub-class within the larger containing (dynamic) class, allowing a final offset to be calculated. Finally, the second component of `default_path ty` function returns the singleton list consisting of the class name if `ty` is a class type, and the empty list otherwise.

What of vtables? The use of paths *à la* Wasserrab *et al.* does away with the need for vtables. On the other hand, we wouldn't want to specify the model in such a way as to preclude this perfectly reasonable implementation strategy. In particular, vtables will be catered for just as in the standard, by maintaining that it is only in POD ("plain old data") types where one can rely on the address of the first field being the same as the address of the containing `struct`. The calculation of sizes must also be under-specified to allow for the presence of extra, user-invisible data at the start of a class.

5.7.3 Casting and Other Type Conversions

We have already seen one use of implicit conversion from one type of value to another, in the rule *[assign-completes]* (Figure 17, page 51), where the relation `nonclass_conversion` is used to this end. This relation governs the way in which various values are allowed to be silently converted from one type to another. Its definition is from the file HOL:`declaration_dynamics`, and is repeated in Figure 29.

The most important of C++'s explicit casts is the `dynamic_cast` operation which allows for run-time checked manoeuvering around a class hierarchy. First up-casts to unambiguous bases are permitted (this is the rule for references; there are parallel rules for the dynamic casting of pointers):

```
(* RULE-ID: dyncast-derived-base-ref *)
(* assume that base is accessible (checked by compiler) *)
    (strip_const dty = Class dcnm) ∧
    (s,{}) |- path (LAST p) to dcnm unique ∧ (* static check *)
    (s,{}) |- path (LAST p) to dcnm via p'
  ⇒
    mng (s, EX (DynCast (Ref dty) (LVal a (Class scnm) p)) se)
        (s, EX (LVal a (Class scnm) (p ^ p')) se)
```

The second hypothesis is strictly redundant in the dynamics; along with the base's accessibility, this will be checked before the dynamics gets a chance to

```
nonclass_conversion s (v1,ty1) (v2,ty2) =
  let ty1 = strip_const ty1
  and ty2 = strip_const ty2
  in
    (integral_type ty1 ∧ integral_type ty2 ∨
     ∃ty0. (ty1 = Ptr ty0) ∧ (ty2 = Ptr Void)) ∧
    (∃i. (INT_VAL ty1 v1 = SOME i) ∧
         (SOME v2 = REP_INT ty2 i))
            (* includes null pointer conversion *)
       ∨
    (strip_ptr_const ty1 = strip_ptr_const ty2) ∧ (v1 = v2)
       ∨
    (∃c1 c2 addr pth1 pth2.  (* this is an upcast *)
        (ty1 = Ptr (Class (LAST pth1))) ∧
        (ty2 = Ptr (Class c2)) ∧
        (SOME v1 = ptr_encode s addr (Class c1) pth1) ∧
        (s,{}) |- c2 casts pth1 into pth2 ∧
        (SOME v2 = ptr_encode s addr (Class c1) pth2))
       ∨
    (∃ty0 base derived p fld.
        (ty1 = MPtr base ty0) ∧ (ty2 = MPtr derived ty0) ∧
        (s,{}) |- path derived to base unique ∧
        (derived, p) ∈ rsubobjs (s,{}) ∧
           (* rsubobjs ensures base is not virtual *)
        (LAST p = base) ∧
        (v2 = v1) ∧
        ((SOME v1 = encode_offset base fld) ∨
         (v1 = null_member_ptr)))
       ∨
    (∃ty0 c.
        (* null pointer conversion for pointers to member *)
        (ty1 = Signed Int) ∧
        (SOME v1 = REP_INT (Signed Int) 0) ∧
        (ty2 = MPtr c ty0) ∧ (v2 = null_member_ptr))
```

Figure 29: The `nonclass_conversion` relation for implicit type conversions. Based on the standard's §4.

execute. The third hypothesis is where the extended path from the dynamic type up to the new base is constructed.

In the other direction (back down a class hierarchy), dynamic casts can fail, and require the source type to be polymorphic (*i.e.*, the source class should have at least one virtual function). The rule governing this behaviour appears in Figure 30. Note how the dynamic type of the l-value does not

```
(* RULE-ID: dyncast-base-other-ref *)
     (strip_const dty = Class dcnm) ∧
     (src_dynty = Class scnm) ∧
     polymorphic s (LAST p) ∧
     (s,{}) |- path scnm to dcnm via p' ∧
     (result =
      if (s,{}) |- path scnm to dcnm unique then
        (* should also check accessible, though I think this
           could be done statically *)
        LVal a src_dynty p'
      else
        EThrow (SOME (New (Class bad_cast_name) NONE)))
   ⇒
     mng (s, EX (DynCast (Ref dty) (LVal a src_dynty p)) se)
         (s, EX result se)
```

Figure 30: Performing a Polymorphic `dynamic_cast`

change: it remains as `src_dynty`. Instead the accompanying path value adjusts: shifting from a path that leads up as far as `LAST p` (the original static type), to one that leads to the class `dcnm`. In this way, a `dynamic_cast` can move the static type from one branch of the ancestry tree to another; in one step, one can do more than just make a down-cast.

When there isn't a class of the desired type available, a reference cast causes a `bad_cast` exception to be thrown. The accompanying rule for dynamic-casting of pointer values has the pointer converted to a null pointer if the destination type is unreachable.

5.8 Reference Types

Reference types pose problems in the contexts where they are distinctive:

- passed as parameters to functions;

- returned from functions;
- when they are initialized.

Otherwise, it is obvious how the existing semantics should treat references: they are l-values. References are only created from l-values, and in the reverse direction, l-values can turn into r-values as required. They will do the right things when of `class` types because they will have the Wasserrab style paths attached. In other words, a reference to a base class may actually be an l-value referring to a derived class.

The remaining sub-sections explain what happens in the three interesting situations.

5.8.1 References as Parameters

When a formal parameter is of reference type, the actual parameter needs to stay an l-value rather than reduce to an r-value. When the called function's environment is being established in its local stack frame, the binding for the formal name can be directly to the actual l-value's address and type. In other words, nothing is allocated in memory to represent the reference. This is not very likely in an actual environment, which will probably have an address in memory somewhere. (The only way to detect this (and this would require the use of undefined behaviour) would be know where local variables were allocated, and to scan this area byte-by-byte, presumably starting at the address of some other, non-reference, local parameter.)

This requires a change to the existing semantics, to remove function arguments from the l-value context (as defined by `valid_lvcontext`). But then, in order to allow some function arguments to decay into r-values, a new rule is introduced:

```
(* RULE-ID: fnapp-lval2rval *)
     lval2rval (s0,e0,se0) (s,e,se) ∧
     fn_expects_rval s0
       (case s0.thisvalue of
           SOME (ECompVal bytes (Ptr ty)) -> SOME ty
        || _ -> NONE)
       f
       (LENGTH pfx)
   ⇒
     mng (s0, EX (FnApp f (pfx ++ (e0 :: sfx))) se0)
         (s, EX (FnApp f (pfx ++ (e :: sfx))) se)
```

The variables pfx and sfx are lists of expressions corresponding to the other actual parameters being passed to the function f. The predicate fn_-expects_rval examines the type of f to determine whether or not the argument at the given position (LENGTH pfx here) is required to be an l-value.

The rule that determines that a function application's sequence point has been reached (*[function-call-sqpt]*) must also change. Previously, this rule checked that the function and all of the arguments had been fully evaluated, and being "fully evaluated" meant "had been evaluated to a value (consisting of a list of bytes)". The new rule checks that every parameter is either a byte-list value (when the function doesn't expect a reference type), or an l-value:

```
(* RULE-ID: function-call-sqpt *)
     (fty = Function retty argtys) ∧
     EVERYi (λi e. if ref_type (EL i argtys) then
                      ∃a t p. e = LVal a t p
                    else ∃v t. e = ECompVal v t)
            params ∧
     is_null_se se
   ⇒
     mng (s, EX (FnApp (FVal fnid fty eopt) params) se)
         (s, EX (FnApp_sqpt NONE (FVal fnid fty eopt) params)
                base_se)
```

The EVERYi function (from HOL:utils) is of type

```
: (num -> 'a -> bool) -> 'a list -> bool
```

and checks whether or not every element of a list satisfies the given predicate, but where the predicate is also given access to the element's index in the list.

5.8.2 References Returned from Functions

In order to return a reference from a function, the model must not force the "l-value to rvalue" conversion that would normally turn l-value expressions in return statements into r-values. But it must also allow that conversion to occur when appropriate. This in turn requires the model to be able to recognise when a function is due to return a reference as opposed to a normal value. The model achieves this by encoding this expectation in the continuation that accompanies every statement evaluation.

The rules for function calls (see, for example *[function-call]* in Figure 19 (page 53)) construct the continuation with a call to `return_cont`. The continuation type is defined in HOL:`statements`:

```
conttype = RVC of (CExpr -> CExpr) => se_info
         | LVC of (CExpr -> CExpr) => se_info
```

meaning that a continuation stores both the side effect record that is to accompany the re-established expression, and a function which will construct it when given the result of the function call.

The definition of `return_cont` is

```
return_cont se ty = if ref_type ty then LVC I se
                    else RVC I se
```

where `I` is the identity combinator (the function equal to $(\lambda x.\ x)$).

In the continuations, the "tags", `RVC` and `LVC`, allow other rules to determine what is expected. To return to the example of the "l-value to r-value" conversion, this is done by the following rule:

```
(* RULE-ID: return-lval2rval *)
     lval2rval (s0,e,se0) (s,e,se) ∧
     (HD s.rvstk = NONE)
   ⇒
     mng (s0, ST (Ret (EX e0 se0)) (RVC c ret_se))
         (s, ST (Ret (EX e se)) (RVC c ret_se))
```

The continuation that accompanies every statement is the second argument of the `ST` tag. Also note that the first argument of a `return` might itself be another statement. If the original expression contained a function call, the body of the called function, a statement, would eventually become the top of the expression. In this rule, the use of the inner `EX` tag precludes this possibility.

The second hypothesis in *[return-lval2rval]* prevents the l-value to r-value conversion occurring if the function is returning a value of class type (when the top of the `rvstk` component will be a `SOME` value). This is because the value will be passed to a copy constructor for the class being returned, and the copy constructor may be expecting a reference.

There are two rules allowing a `return` statement to pass its expression-value to the continuation. The rule for returning normal values is *[return-rvalue]* (we saw this rule earlier, in Section 5.5):

```
(* RULE-ID: return-rvalue *)
     is_null_se se0
   ⇒
     mng (s, ST (Ret (EX (ECompVal v t) se0)) (RVC c se))
         (s with rvstk updated_by TL, EX (c (ECompVal v t)) se)
```

The rule for returning a reference *[return-lvalue]* is similar:

```
(* RULE-ID: return-lvalue *)
     is_null_se se0
   ⇒
     mng (s, ST (Ret (EX (LVal a t p) se0)) (LVC c se))
         (s with rvstk updated_by TL, EX (c (LVal a t p)) se)
```

5.8.3 Declaring References

When a reference is declared, it must also be initialized, and the initializing expression must be an l-value. But normal initializations need to be able to turn l-values into r-values so there is an "lvalue-to-rvalue" rule for variable declarations that are accompanied by initializations (this rule earlier appeared in Section 5.5.3):

```
(* RULE-ID: decl-vdecinit-lval2rval *)
     lval2rval (s0,e0,se0) (s,e,se) ∧
     ¬ref_type ty ∧
     ((f = CopyInit) ∧ ¬class_type (strip_const ty) ∨
      (f = DirectInit))
   ⇒
     declmng mng (VDecInitA ty loc (f (EX e0 se0)), s0)
                 ([VDecInitA ty loc (f (EX e se))], s)
```

A variable declaration of reference type is ready to "fire" when its initializing expression has reduced to an l-value, and when there are no remaining side effects in the side effect record. At this point, the variable address map in the state is updated to point at the l-value's address, and the type map is also updated to make the type of the name the same as the type of the l-value. This step is controlled by the rule *[decl-vdecinit-finish-ref]*, for which see Figure 31.

```
(* RULE-ID: decl-vdecinit-finish-ref *)
(* if isSome, aopt is the address of a containing class *)
     is_null_se se ∧
     ((f = CopyInit) ∨ (f = DirectInit)) ∧
     (if class_type ty1 then
        (s0,{}) |- dest_class ty1 casts p into p'
      else (p' = p)) ∧
     (s = new_addr_binding refnm aopt (a,dest_class ty2,p') s0) ∧
     (s.stack = (env,thisv,amap,[]) :: rest)
   ⇒
     declmng mng
             (VDecInitA (Ref ty1)
                        (RefPlace aopt refnm)
                        (f (EX (LVal a ty2 p) se)), s0)
             ([], s with <| stack := rest; allocmap := amap |>)
```

Figure 31: How References are Initialized

5.9 Polymorphism & Multiple Inheritance

As already suggested, multiple inheritance has been modelled by following the approach described in Wasserrab *et al* [15]. Most of the dynamic rules have already been presented, so that the modelling of multiple inheritance is best understood by considering the auxiliaries supporting those rules. Most of these auxiliaries are defined in HOL:`class_info`.

At the top level, the rule for calculating the function that will be called dynamically is *[function-member-select]*, which appears in Figure 26, on page 75. This rule describes how the call to method `fld` is resolved for an object located at address `a`, with dynamic type `C`, and where the static type of the object is the last element of the list `Cs`. The list `Cs` is also a path through the class hierarchy, starting at the dynamic type and ending at the current static type. (Note that an object's dynamic type is determined on object creation, and persists for an object's entire life-time. In contrast, an object's static type is the type ascribed to it by a particular piece of code. Different pieces of code may well "see" the same object as having different types. In this sense, an object's dynamic type is unchanging, but it will have a variety of static types across the text of a program. Confusing, but true!)

The first premise ("has-least-method") examines the static type of the object for which the method will be called, `LAST Cs`. Starting at that point

in the hierarchy it looks upwards (*i.e.*, towards base classes) for the nearest base class that provides an implementation of the desired method. This base-class might be `LAST Cs` itself, in which case the path found (`Ds`) will be the singleton consisting of just `LAST Cs`. There must also be a unique closest ancestor providing the desired method. If this isn't the case, then there will have been a compile-time error, as the call will be statically ambiguous.

The second premise ("selects-via") then determines the dynamic location of the desired method. There are two cases. The simple case (reiterating the discussion in Section 5.7.2) is when there is a unique best method for the dynamic type, `C`. Imagine, for example, that there is a four-element singly-linked inheritance graph, from base `B0` down to most-derived `B3`, with implementations of the method `fld` at `B0` and `B2`. If the object is actually of type `B3`, but is statically seen as type `B1`, then `C` is `B3`, and `Cs` will be `[B3,B2,B1]`. The first premise ("has-least-method") determines that, starting at `B1` (the static type) there is an implementation of `fld` at path `[B1, B0]`. This will be the instantiation of the rule's variable `Ds`.

The simple case for "selects-via" checks whether or not there is also a least method for the dynamic type (ignoring, for the moment, the path giving the static type). Thus (from HOL:`class_info`)

```
(* RULE-ID: selects-simple *)
     s |- C has least method mname -: minfo via Cs'
   ⇒
     s |- (C,Cs) selects mname -: minfo via Cs'
```

In the simple example, we will thus conclude that

```
   s |- (B3, [B3,B2,B1,B0]) selects fld -: info via [B3,B2]
```

so that the call will be to the implementation of `fld` in `B2`, and the `this` pointer will be adjusted so that the type and path information associated with its value will be `(B3,[B3,B2]`, *i.e.*, a dynamic type of `B3` (as always) and a static type of `B2`.

The same rule applies in a much more complicatd seeming situation, where multiple inheritance and shared base objects come into play. Consider the program in Figure 32. When the call to `cref.f()` is made, the type and path associated with the reference will be `(D,[D,C1])`. Statically, the reference is a `C1` value, but dynamically, it's really of class `D`. The first premise in rule *[function-member-select]* finds that there is a path (`Ds` in the rule) which is appropriate for `f`. This path is `[B]`. The fact that the path does not include the derived object's name, and is just a bare reference to a

```
#include <iostream>

class B {
public:
  virtual int f() { std::cout << "B's f\n"; return 3; }
  virtual ~B() { }
};

class C1 : virtual public B {
};

class C2 : virtual public B {
public:
  virtual int f() { std::cout << "C2's f\n"; return 4; }
};

class D : public C1, public C2 {
};

int dosomething(C1 &cref)
{
  return cref.f();
}

int main()
{
  D d;
  return dosomething(d);
}
```

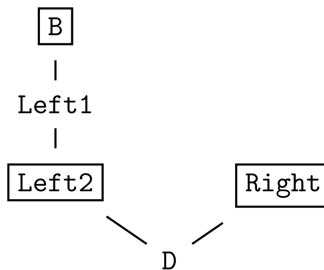
Figure 32: Multiple inheritance with shared base objects. (This program is in the `notes` directory with name `diamond-multinherit.cpp`.)

class indicates that it is a path to a shared base. When such a path is the second argument of path concatenation (the ^ operator in the second premise), the result is just the second argument, so that the second hypothesis in the rule becomes

```
s |- (D,[B]) selects f -: info via Cs'
```

The first, simple, rule for "selects-via" resolves this. Ignoring the path [B], the simple rule checks whether or not there is a unique least path to an f from D. There is such a path, and it is [D,C2]. So, the call to cref.f() ends up being a call to the f in class C2, in an "unrelated" part of the object hierarchy.

For those situations where there is not a unique path from the dynamic type to a best selection, there is another more complicated rule defining "selects-via". In Figure 33, the inheritance hierarchy looks like



where the boxed class names are those that implement the function f. The question is which f will be called when the dynamic type is D, and when the static type is Left1. There is no unique, least implementation of f visible from the dynamic type (D), so the simple rule does not apply. Instead, the notion of *overrider* is introduced via the rule

```
(* RULE-ID: selects-with-overrider *)
     (∀minfo Cs'.
       ¬(s |- C has least method mname -: minfo via Cs')) ∧
     s |- (C,Cs) has overrider mname -: minfo via Cs'
   ⇒
     s |- (C,Cs) selects mname -: minfo via Cs'
```

where the static type of the value again plays a role. In this case, the f that gets called is that in Left2. For further details on exactly how this is defined, see either HOL:class_info, or Wasserab *et al* [15].

```
#include <iostream>

class B {
public:
  virtual void f() { std::cout << "B's f\n"; }
  virtual ~B() { }
};

class Left1 : public B { };

class Left2 : public Left1 {
public:
  virtual void f() { std::cout << "Left2's f\n";  }
};

class Right {
public:
  virtual void f() { std::cout << "Right's f\n"; }
  virtual ~Right() { }
};

class D : public Left2, public Right { };

void dosomething(Left1 &l1ref) { l1ref.f(); }

int main()
{
  D d;
  // d.f();  would be statically ambiguous
  dosomething(d);
  return 0;
}
```

Figure 33: A lop-sided 'V' inheritance hierarchy. Available as `notes/lopsided-v.cpp`.

5.10 Object Lifetimes

For comments specific to the lifetimes of class r-values, see Section 5.11 below.

Constructors Handling constructors has easily wrought the greatest change to the simple C model. The basic approach taken has been to encode as much as possible with "evolving syntax". Just as `while` can be modelled by having

$$\texttt{while}\ (G)\ \mathit{body}$$

become

$$\texttt{if}\ (G)\ \{\mathit{body}\texttt{;}\ \texttt{while}\ (G)\ \mathit{body}\}$$

so too do calls to C++ constructors create new programs "in place". The advantage of this approach is that there is less need for relatively complicated state to be recorded in yet more fields in the big record type `state` (see Figure 11 on page 38). Instead, programs can unfold into more elaborate forms directly. The disadvantage of this approach is that the original syntax may not support enough forms, requiring new special syntax to be created, or for existing forms to be extended with new parameters.

The existing handling of block-statements is an example of this latter disadvantage. In particular, the abstract syntax has a constructor `Block` with type

```
: bool -> decl list -> stmt list
```

where the boolean flag indicates whether or not the block has been entered yet. In this way, the abstract syntax has values in it that can't be written down in the concrete syntax. (Simiarly, the original thesis model for C included the `RVR` constructor and the `FnApp_sqpt` intermediate form (written $\hat{f}$) for function calls.)

Constructors are intimately tied up with declarations and initialization. In the simple C world, a variable comes into being in two stages. First space in memory is allocated for the variable (whether on the stack or heap), and the variable name is associated with that space for the span of the variable's life. Then there is an optional initialization phase, when the piece of memory associated with the variable space is filled in with some value.

The C++ model is similar. All new objects (but not references) must be associated with some new space. Then they may or may not be initialized. Additionally, in C++, an object that is only declared, and which does not appear to be initialized, will actually have its default constructor called.

The abstract syntax supporting this is all defined in HOL:`statements`:

Constructor	Argument Types	Description
`VDec`	$\mathit{name}, \mathit{type}$	no initialization
`VDecInit`	$\mathit{name}, \mathit{type}, \mathit{initializer}$	initialization (unallocated)
`VDecInitA`	$\mathit{varlocation}, \mathit{type}, \mathit{initializer}$	initialization (space allocated)

For example, when a class is declared with no explicit initialization of any sort, meaning that the default constructor will be called, the syntax moves through all three stages. The abstract syntax corresponding to something like

```
{
   classname c;
   ...
}
```

will be `VDec "c" classname`. Because this is a class type, rule *[decl-vdec-class]* will fire (and can do so immediately), and the declaration will become `VDecInit "c" classname init`, where the form of `init` will encode the fact that a direct initialization is being performed, and that there are no arguments. Then the rule *[decl-vdecinit-start-evaluate-direct-class]* fires. Side conditions of this rule cause space to be allocated (at address `a`), the state's maps from names to addresses be updated, and for the construction of the class to be recorded so that it can be destroyed later.

The syntax also evolves to become

```
VDecInitA classname (ObjPlace a) init'
```

where the new initializer records that a function call to a constructor is about to happen, and where that construction will happen (*i.e.*, `init'` includes a reference to `a`).

There are three forms of initializer. The first two are `DirectInit0` and `DirectInit` and correspond to direct initialization [5, §8.5 paragraph 12]. The `DirectInit0` constructor takes as arguments a list of expressions. In this way, concrete syntax such as

```
{
  Classname c(x,&y,z+1);
}
```

is directly modelled. (When the rule *[decl-vdec-class]* fires, the newly created `DirectInit0` constructor takes an empty list of arguments.)

The `DirectInit` constructor takes one argument, an "extended expression", which will initially be an expression constructed by an application of the special constructor `ConstructorFVal` to the same argument list. This form needs to be an extended expression so that the body of the constructor (a statement) can be entered.

The other form of initializer is constructed by the function `CopyInit`. This corresponds to the syntax

```
{
  type varname = expression;
  ...
}
```

which is a copy-initialization [5, *ibid*]. When the type of the new object is not a class, there is no difference between copy-initialization and direct-initialization, reflected in the rule *[decl-vdecinit-start-evaluate-direct-nonclass]*, which moves from a `DirectInit0` to a `CopyInit` initializer.

When a `CopyInit` initializer completes its evaluation, yielding a value, that value be copied across into the space earlier allocated for the object. For non-class types this is done with the same `val2mem` helper function that is used to apply side effects. For class types, this copying must be performed by a call to the copy constructor.

Constructor Calls The expression form corresponding to a constructor call uses the expression form for function applications, but with a special form in the place of the function value. This allows the normal evaluation of function applications (with the unspecified order of evaluation of arguments, for example). The function value position is filled by a new abstract syntactic form `ConstructorFVal`. This takes four parameters:

- a boolean indicating whether or not the constructor is being called for a most-derived object or not;
- a natural number recording the allocation level of the creation. This allocation level is the same for the class and any of its sub-classes, and is a count from the *bottom* of the stack;
- the address of the space which the constructor is to operate over; and
- the name of the class that is being constructed

So, if a declaration is made of the form

```
{
   class v(x);
}
```

then, once sufficient space is allocated for the new object `v` (at address `a`, say), the abstract syntax will look like

```
VDecInitA (Class (Base "class"))
          (ObjPlace a)
          (DirectInit
             (EX (FnApp (ConstructorFVal T n a (Base "class"))
                        [Var "x"])
                 base_se))
```

where the number `n` records the allocation level for this creation. Note how the name `v` has disappeared, subsumed into the allocated address at `a`.

The `ObjPlace` constructor is used to distinguish this from the situation where a reference is being initialized. The `EX` constructor specifies that the current form is an expression (as opposed to the statement that will be in this position once the constructor body is entered). The `base_se` value is the empty side-effect record. This will evolve as references to and updates of memory occur.

Once the parameters to the constructor have been evaluated, the constructor body can be entered.This happens in rule *[constructor-function-call]*, for which see Figure 34. The basic rule is complicated enough, and there is more complexity hidden behind the auxiliary functions and relations. The first auxiliary is the relation `find_constructor_info`, which appears in the rule's first hypothesis. This relation treats its first three parameters (`s0`, `cnm` and `args`) as inputs. These are the current state, the name of the class being constructed, and the actual arguments being passed to the constructor. The remaining arguments to the relation are "outputs". The variable `params` is the list of formal parameters (names and types). The variable `mem_inits` is the list of "mem-initializers" (see [5, §12.6.2]) associated with the constructor, and `body` is the constructor's body. The `find_constructor_info` auxiliary is responsible for resolving which constructor needs to be called, based on the types of the actual arguments.

The second hypothesis of the rule constructs the sequence of variable declarations corresponding to the parameters, using standard functional programming auxiliaries `MAP` and `ZIP`. Parameter passing is just like vari-

```
(* RULE-ID: constructor-function-call *)
     find_constructor_info s0 cnm args params mem_inits body ∧
     (pdecls = MAP (λ((n,ty),a).
                        VDecInit ty (Base n)
                           (CopyInit (EX (NoScope a) base_se)))
                   (ZIP (params, args))) ∧
     (SOME this = ptr_encode s0 a (Class cnm) [cnm]) ∧
     (cpfx = construct_ctor_pfx s0 mdp (LENGTH s0.stack) a
                                cnm mem_inits) ∧
     (newstmt =
        if is_catch body then
          let (bod,handlers) = dest_catch body
          in
            Block T pdecls
              [Catch (Block F cpfx [bod])
                     (MAP (λ(e,st).
                             (e, Block F [] [st; Throw NONE]))
                          handlers)]
        else Block T (pdecls ++ cpfx) [body])
   ⇒
     mng (s0, EX (FnApp_sqpt (ConstructorFVal mdp subp a cnm)
                             args) se0)
         (s0 with <|
           thisvalue := SOME (ECompVal this (Ptr (Class cnm)));
           stack updated_by
                 (CONS (s0.env, s0.thisvalue, s0.allocmap, []));
           rvstk updated_by (CONS NONE);
           env := empty_env
          |>,
          ST newstmt (RVC (λe. ConstructedVal subp a cnm)
                          se0))
```

Figure 34: Making a Call to a Class Constructor

able declaration[8], and so the model's existing treatment of declarations can be re-used to set up the binding between formal names and actual values. Note that the expressions that initialize the parameters have already been fully evaluated, so that there will be no expression evaluation done when the declarations come to be evaluated (except for any class construction that may be called for).

The third hypothesis calculates a value for the `this` value. The dynamic and static type of the `this` pointer will be the same (as the pair C and $[C]$ are passed to `ptr_encode`), and thus there will not be any polymorphic dispatch to functions in derived classes if any virtual functions are called in the constructor bodies.

The fourth hypothesis constructs a `cpfx` of declaration calls to initialize class members and bases. This is all done in the complicated function `construct_ctor_pfx` (defined in `dynamicsScript`). This constructs a sequence of declarations to initialize the non-static members of the new class, and the class's immediate bases. The mem-initializers are consulted to see what initializers should be provided. (If a mem-initializer is not provided for a given member or base, then that object will be value- or default-initialized; see [5, §12.6.2, paragraphs 3–4].)

For example, in Figure 35, before class `C`'s constructor body is even entered, the parameters `cptr` and `i` need to be declared and initialized with actual values. Subsequently, all of `C`'s immediate bases (just `B` in this case) need to be constructed, followed by its members (`ptr` and `sz`). Note that while the parameters need to have space allocated for them, the bases and members do not (because the space for the entire object was allocated at the `VDecInit` stage).

Assume that the constructor has been called with parameters `x` and `y`, and that these have evaluated to values `xval` and `yval`. The sequence of declarations that are constructed to precede the constructor body is given in Figure 36. The first two declarations are of the parameters. The next constructs the base `B`, and the last two construct the non-static members. The body of `construct_ctor_pfx` is responsible for calculating the offsets of the members (given as `Boff`, `ptroff` and `szoff` in the figure.

Note how the first argument of the `ConstructorFVal` form in the construction of the base `B` is false; this is because `B` is not the most-derived object. If there were any shared bases in the example, the most-derived object would be "responsible" for constructing them (see [5, §12.6.2, para-

[8]This does away with the Cholera approach which had a number of auxiliary relations effectively duplicating what occurred in variable declaration.

```
#include <cstring>

class B {
  int x;
public:
  B(int i) : x(i) {}
};

class C : public B {
  char *ptr;
  int sz;
public:
  C(char *cptr, int i)
    : B(cptr[i]), ptr(cptr), sz(strlen(cptr)) { }
};
```

Figure 35: C++ Constructors with *mem-initializers*. (Available as `notes/mem-inits.cpp`.)

graph 5]). If `C` had any non-static members of class type, then these would be constructed with their `mdp` and `subobj` flags both set to true.

When the constructor for the base class `B` comes to be called, it will in turn initialize its members. The constructor for `B` is called with an argument (`cptr[i]`) that needs to be evaluated in the context where the parameters are in scope, so it is clear that the declarations for the parameters must come before the base and member initializations.

Object Destruction When an object of class type is first declared (with a `VDecInit` form), it has memory allocated sufficient to contain the new class in its entirety (including sub-objects). This allocation is reflected in the state's `allocmap`. When the block in which this declaration was made is left, this allocation is forgotten (the `stack` component of the state is popped). At the same time, the destructor for the object must be called. These required object destructions are recorded in the `stack` as well.

Each element of the `stack` is a 4-tuple. The fourth component of these tuples is a list of objects to destroy, encoded as a list of addresses and class names. Potentially, one object may appear on the `stack` twice, both at the topmost level and also deeper in the stack. This can occur when sub-objects are created within a constructor for an enclosing class. If an object appears

```
VDecInit (char *) cptr (CopyInit (NormE xval base_se))
VDecInit int      i    (CopyInit (NormE yval base_se))

VDecInitA B (ObjPlace (a + Boff))
            (DirectInit
               (NormE
                  (FnApp (ConstructorFVal F 1 (a + Boff) B)
                         [Deref(Plus (Var "cptr")
                                     (Var "i"))])
                  base_se))

VDecInitA (char *) (ObjPlace (a + ptroff))
                   (CopyInit (NormE (Var "cptr") base_se))
VDecInitA int      (ObjPlace (a + szoff))
                   (CopyInit (NormE (FnApp (Var "strlen")
                                           [Var "cptr"])
                                    base_se))
```

Figure 36: The variable declarations constructed to precede the body of `C`'s constructor (from Figure 35). Assume that the enclosing class `C` is being created at allocation level 1. This in turn makes `B` be created at that level too (witness the second parameter of the `ConstructorFVal` constructor).

twice, its presence at the top of the `stack` is ignored, except when the block is being exitted through an exception (for more on this, see Section 5.6.4). In that circumstance, objects and sub-objects alike must be destroyed in the reverse order of their construction.

The rule for exitting from a block can only occur if the current scope's list of objects needing destruction is empty. As long as it is not empty, the destructor corresponding to the object on the top of the stack is set up to be called. This is done in rule *[block-exit-destructors-to-call]*:

```
(* RULE-ID: block-exit-destructors-to-call *)
(* normally-constructed objects at this level are always
   destroyed here *)
     (sel4 (HD s0.stack) = destroy_these) ∧
     ¬(destroy_these = []) ∧
     final_stmt st c ∧
     ((destcalls, s) =
        realise_destructor_calls (exception_stmt st) s0)
   ⇒
     mng (s0, ST (Block T [] [st]) c)
         (s, ST (Block T [] (destcalls ++ [st])) c)
```

(The `sel4` function returns the fourth component of a tuple.) This is another example of evolving syntax: the block that the flow of control is about to leave, has this departure deferred with the insertion of new statements before the block's final statement. The rule allowing an exit to eventually occur is *[block-exit]*, for which see Figure 22 (page 61).

Actually calling a destructor is straightforward because there are no parameters, nor anything comparable to the mem-initializers. The requirement in the standard that sub-objects be destroyed before the body of a destructor is entered is handled by the fact that the sub-obje1ct information is entered into the `stack` ahead of the final encompassing object.

Using the Heap Classes, and other types, can be allocated on the heap with the `new` operator. For non-classes the rule is *[new-nonclass]*, in Figure 37. When the address is found for the new object (using the `malloc` auxiliary), the allocation is recorded in the state's `hallocmap` field, rather than the `allocmap`, which is used for local objects. The `hallocmap` is not affected by moving in and out of blocks.

When a class is to be allocated, a constructor must be called. This is reflected in the rule of Figure 38. The hypotheses are very similar to the non-class rule, but the conclusion differs because the class's constructor must be

```
(* RULE-ID: new-nonclass *)
     ¬class_type (strip_array ty) ∧
     malloc s0 ty a ∧
     (result_ty = case strip_const ty of
                     Array bty n -> bty
                  || somety -> ty) ∧
     sizeof T (sizeofmap s0) ty sz ∧
     (s = s0 with <|
            hallocmap updated_by (UNION) (range_set a sz) ;
            constmap := if const_type ty then
                           s0.constmap UNION range_set a sz
                        else s0.constmap DIFF range_set a sz
          |>) ∧
     (SOME ptrval = ptr_encode s0 a result_ty [])
   ⇒
     mng (s0, EX (New ty NONE) se)
         (s, EX (ECompVal ptrval (Ptr result_ty)) se)
```

Figure 37: Allocating a Non-Class Object on the Heap with `new`

called before the pointer to the object can be returned. The use of allocation level 0 in this rule ensures that the object's creation will never be recorded in the `stack`.

5.11 Class R-Values

As discussed earlier, r-values of class type are harder to model than r-values of other types. (Note that the standard also talks of "temporary values" in this context.) For example, it is legitimate to call member functions on such values, which in turn means that such values must continue to have addresses so that `this` can have a value in the bodies of those member functions. But once class r-values are given residence in memory, they have to be allocated, constructed, destroyed and de-allocated properly. This opens up many cans of worms.

The first saving grace is that class r-values can only occur as a result of function calls. There is no way of creating a class r-value within a normal expression except by calling a function that returns a value of the given type.[9] Once created, class r-values can be

[9]Compound literals can be used to initialize POD classes, but these are not general r-

```
(* RULE-ID: new-simple-class *)
(* The T 0 parameters to the ConstructorFVal constructor indicate
    1. the object produced is most-derived, and
    2. it is not stack-allocated
*)
     (Class cnm = strip_const ty) ∧
     malloc s0 ty a ∧
     sizeof T (sizeofmap s0) ty sz ∧
     (s = s0 with <|
             hallocmap updated_by (UNION) (range_set a sz) ;
             constmap := if const_type ty then
                            s0.constmap UNION range_set a sz
                        else s0.constmap DIFF range_set a sz
          |>) ∧
     (SOME ptrval = ptr_encode s0 a ty [cnm])
   ⇒
     mng (s0, EX (New ty (SOME args)) se)
         (s, EX (CommaSep (FnApp (ConstructorFVal T 0 a cnm) args)
                          (ECompVal ptrval (Ptr ty)))
                se)
```

Figure 38: Allocating a Class Object on the Heap

- passed as parameters to other functions;
- used to initialize newly declared variables; and
- returned from functions

It is also possible for an r-value to be the "result" of an expression-statement (the `Standalone` constructor in the `stmt` type). For example, in

```
class C { ... };
C f(int x);
int g(int y) { f(y + 1); ... }
```

the body of `g`, a "standalone" `C` r-value is created and then does nothing (presumably there were interesting side effects incident on its creation).

The rest of this section describes the modelling of five modes of being: creation, the three bullet-points above, and eventual destruction. In fact, r-value destruction is straightforward because it fits into the general object lifetime framework as already described. When a full lexical expression finishes its evaluation, the various rules for that situation cause the equivalent of a block exit to occur, which first requires the destruction of objects from that allocation level.

5.11.1 R-Value Creation and Function Return

R-values must live in memory, so it is important to know how the memory they consume comes to be allocated. Thankfully, we know that all r-values are created by function calls, so the model can arrange for memory to be allocated when a function is about to be entered. This memory is allocated at the level of the caller rather than the callee, which is important because the temporary lives at the level of the caller.

Allocation is done in the rule *[allocate-rvrt]*, in Figure 39. If the first argument slot of the `FnApp_sqpt` constructor is `NONE` (as it is initially, thanks to rule *[function-call-sqpt]*), then this rule can fire, and update it. The call to `find_best_fnmatch` is used to calculate the type of value returned by the function. If that type is a class type, then space is allocated for a value of the right size, and this type and the space's address and the allocation level is recorded in the first argument slot. (The allocation level information ensures that the object will have its destructor called at this level.) Naturally, if

values as found inside arbitrary expressions. Moreover, this model doesn't handle that form of initialization in any case.

```
(* RULE-ID: allocate-rvrt *)
(* allocates space for a function call so that it can return an
   object r-value *)
     find_best_fnmatch s0 fnid (MAP valuetype args)
                       rtype params body ∧
     (strip_const rtype = Class cnm) ∧
     malloc s0 rtype a ∧
     sizeof T (sizeofmap s0) rtype sz
   ⇒
     mng (s0, EX (FnApp_sqpt NONE
                            (FVal fnid ftype thisobj)
                            args)
                 se0)
         (s0 with allocmap updated_by (UNION) (range_set a sz),
          EX (FnApp_sqpt (SOME
                           (LENGTH s0.stack, a,
                            dest_class (strip_const rtype)))
                         (FVal fnid ftype thisobj)
                         args)
             se0)
```

Figure 39: Allocating Memory into Which Class R-Values will be Constructed

the return type of the function call is not a class type, then the first argument slot will remain `NONE`.

We have already seen how *[function-call]* (p53) subsequently transfers the information from the `FnApp_sqpt` constructor onto the state's `rvstk` stack when the function call enters the body of the function. The last step of r-value creation comes when a `return` statement is executed.

At the base level, all r-values must be created by returning an l-value in a function such as[10]

```
C f(...)
{
  C x;
  ...
  return x;
}
```

If this is the case, we will eventually see the following situation in the abstract syntax:

```
ST (Block T [] [Return (EX (LVal a t p) se)]) (RVC c)
```

This tells us that we have an l-value due to be returned but that the statement continuation is expecting an r-value. The last thing we want happening at this point is for *[block-exit-destructors-to-call]* to fire: that may well invalidate the l-value. For this reason, the definition of `final_stmt` (p55) checks to see that the expression being returned is not a class l-value when there is an `RVC` continuation.

Instead, a fresh object needs to be constructed, calling a constructor to do so. This is done in the rule *[ret-construct-rvalue]*:

```
(* RULE-ID: ret-construct-rvalue *)
    (HD s.rvstk = SOME (alvl, a,cnm)) ∧
    ¬(e0 = ConstructedVal alvl a cnm)
  ⇒
    mng (s, ST (Ret (EX e0 se0)) (RVC c se))
        (s, ST (Ret (EX (FnApp (ConstructorFVal T alvl a cnm)
                               [e0])
                        se0))
               (RVC c se))
```

[10]There are many possible variations on this of course: the l-value need not be an automatic variable but might be some global; if returning an automatic, the block where it is allocated need not be the body of the whole function; etc

Calling a constructor will eventually return a `ConstructedVal` value that has been allocated to fill in the appropriate space, at address `a'`. Note that the type of the value `e0` may not be exactly the same as the value being returned. Indeed it does not need to be of class type at all. For example, the rule above would fire in the program

```
class C {
public:
  C(int x) { ... }
  ...
}

C f(...) { int x; ... return 3; }
```

When the call to the constructor returns it will be with a `ConstructedFVal` value at the given address. This then allows *[ret-class-rvalue]* to fire:

```
(* RULE-ID: ret-class-rvalue *)
     (HD s.rvstk = SOME (alvl,a,cnm)) ∧
     (e0 = ConstructedVal alvl a cnm)
   ⇒
     mng (s, ST (Ret (EX e0 se0)) (RVC c se))
         (s with <|
            rvstk updated_by TL;
            stack updated_by record_creation alvl a cnm
          |>,
          EX (c e0) se)
```

The function `record_creation` records the creation of a class in the `stack` component of the state so that it can be destroyed at the appropriate point later. The function is defined in HOL:`declaration_dynamics`:

```
  record_creation alvl a cnm stk =
    let stk' = update_nth_rev (LENGTH stk)
                              (upd4 (CONS (a,cnm))) stk
    in
      if alvl < LENGTH stk ∧
         ¬MEM (a,cnm) (sel4 (REV_EL alvl stk))
      then
        update_nth_rev alvl (upd4 (CONS (a,cnm))) stk'
      else
        stk'
```

In all cases, the creation of the object is recorded at the top of the stack. The utility function `update_nth_rev` updates a list at the given index, where the elements are counted with the first element being #1 (rather than the typical 0), and from the end of the list (the bottom of the stack). Then, if the allocation level of the object is less than the length of the stack, this indicates that the object really "belongs" to an earlier allocation level. This will happen if the object is a sub-object of some larger object.

Eliding Unnecessary Copy Constructions In the standard's §12.8, paragraph 15, it is said

> when a temporary class object that has not been bound to a reference (12.2) would be copied to a class object with the same cv-unqualified type, the copy operation can be omitted by constructing the temporary object directly into the target of the omitted copy

Such a situation arises when the expression attached to a return statement is itself a call to a function returning an object of the same class type that is being returned. We can model this possible optimisation with the rule *[ret-pass-rvrt]*:

```
(* RULE-ID: ret-pass-rvrt *)
     (fnc = FVal fnid ftype thisobj) ∧
     find_best_fnmatch s fnid (MAP valuetype args)
                        (Class cnm) params body ∧
     (HD s.rvstk = SOME (alvl, a, cnm))
   ⇒
     mng (s, ST (Ret (EX (FnApp_sqpt NONE fnc args) se)) c)
         (s, ST (Ret (EX (FnApp_sqpt (SOME(alvl,a,cnm)) fnc
                                      args)
                         se))
                 c)
```

As per the standard's language, this rule only fires if the type returned by the inner function is the same as the type of the function enclosing the `return` statement.[11] Nor does this rule stop *[allocate-rvrt]* from firing. If the latter happens, an unnecessary copy will be performed, but this reflects the standard's permission rather than requirement of the optimisation.

[11] If the expected return type is an ancestor of the type of the object actually being returned, then a copy constructor will be called and the return value will be subjected to "slicing".

Creation of R-Values By Explicitly Calling Constructors C++ allows constructors to be called "as normal functions" within expressions (actually this is known as *explicit type conversion (functional notation)*, see [5, §5.2.3]). For example

```
class C {
public:
  C(int x) { ... }
  ...
};
int g(C);
int f(int y)
{
   return g(C(y + 3));
}
```

The model does not explicitly model this in the abstract syntax, but assumes instead that for every class constructor there is an accompanying function with arguments of the same type and declared as returning an object of the given class type. When an explicit type conversion appears for a class, a call to this accompanying function can be substituted.

5.11.2 R-Values as Parameters and Initializers

We have already seen that when a value is passed as a parameter, that is treated identically to the copy-initialization of an automatic variable with the parameter's name. Thus, if we have

```
int f(C parameter) { body }
```

and this function `f` is called

```
... f(expression) ...
```

then this becomes the equivalent of

```
{
  C parameter = expression;
  {
    body
  }
}
```

Thus, we can handle class r-value parameters as initializers. The only difference between the two scenarios is that the initializing expression will already be fully evaluated when it has been created as a parameter (because function arguments must be fully evaluated before execution of a function can begin).

If a class r-value has been passed as a parameter, the abstract syntax will look like:

```
VDecInit ty nm (CopyInit (EX (ConstructedVal alvl a cnm) se))
```

If the `ty` is equal to `Class cnm`, then the space already allocated for the r-value can be used for the local variable. This behaviour is captured in rule *[decl-parameter-copy-elision]*:

```
(* RULE-ID: decl-parameter-copy-elision *)
     T
   ⇒
     declmng mng
       (VDecInit (Class cnm) nm
          (CopyInit (EX (NoScope (ConstructedVal alvl a cnm))
                     base_se)),
        s0)
       ([], new_addr_binding nm NONE (a,cnm,[cnm])
              (new_type_binding nm (Class cnm) s0))
```

There is another scenario where unnecessary copying can be avoided: if the local variable has had space allocated for it, and the expression to be evaluated is about to perform a function call returning an r-value of the same type, then this function call can construct its result in the space allocated for the local variable, and *[allocate-rvrt]* can again be avoided. The rule for this behaviour is *[decl-fncall-copy-elision]*:

```
(* RULE-ID: decl-fncall-copy-elision *)
     (fnc = FVal fnid ftype thisobj) ∧
     find_best_fnmatch s0 fnid (MAP valuetype args)
                       (Class cnm) params body
   ⇒
     declmng mng
       (VDecInitA (Class cnm) (ObjPlace a)
                  (CopyInit (EX (FnApp_sqpt NONE fnc args) se)),
        s0)
       ([VDecInitA
           (Class cnm) (ObjPlace a)
           (CopyInit
              (EX (FnApp_sqpt (SOME(LENGTH s0.stack,a,cnm))
                              fnc
                              args)
                  se))],
        s0)
```

If it hasn't been short-circuited by *[decl-parameter-copy-elision]*, a class copy-initialization eventually finishes when the expression under the `CopyInit` constructor is a `ConstructedVal` with the same address as was allocated. This is rule *[decl-class-copy-finishes]*:

```
(* RULE-ID: decl-class-copy-finishes *)
     is_null_se se ∧
     (e = ConstructedVal alvl a cnm) ∧
     (s.stack = (env,thisv,amap,[]) :: rest) ∧
     (s' = s with <| stack := rest; allocmap := amap |>)
   ⇒
     declmng mng
       (VDecInitA (Class cnm) (ObjPlace a)
                  (CopyInit (EX e se)), s)
       ([], s' with stack updated_by record_creation alvl a cnm)
```

Note how the `alvl` variable is not constrained, but we can argue that it will in fact be the length of the stack once it is popped of the information that pertained to the initializer expression. This is because copy-initialisation is not used for sub-object construction, and this rule does not get a chance to fire if the allocation level is from ealier in the stack (as happens in *[decl-parameter-copy-elision]*).

6 Validation

There are at least two possible forms of mechanical validation possible for a semantics such as the one presented in this report: sanity theorems, and evaluation of the specification on ground programs. In addition, there are informal, social, routes to validation. This section of the report describes the (limited) steps already taken along these three paths, and discusses what might still be done.

Sanity theorems A sanity theorem is a result that we expect to be true of a specification, should be relatively easy to prove, and whose falseness would indicate a serious flaw in the specification. (Such theorems are a class of "formal challenge", as discussed in Rushby [10].)

Throughout the HOL sources, there are a number of sanity theorems. These are identified by ML comments of the form `(* SANITY *)`. There are 39 of these. There is also one slightly more substantial result in the theory HOL:`sanity`, and its proof pales beside those of transitivity, reflexivity and anti-symmetry of name instantiation in HOL:`instantiation`. These are painful to establish because of the complexity of the abstract syntax that can be instantiated, but they are important because they indicate that Siek and Taha's basic framework [12] is still usable in the more complicated setting of this C++ semantics.

My earlier work on C [6], which included a number of simple sanity theorems, also included proofs of the "standard" properties beloved of programming language theorists: type safety and progress. These properties demonstrate respectively that expressions of a given type continue to posses that type as they evaluate, and that an operational semantics never "blocks" but only finishes program evaluation in a valid final state (the return of a value, or some sort of abort state, say).

Such results for C++ would likely be considerable work. In particular, type safety would be difficult because the current semantics has an inadequate description of the type system. (The focus of this report is the dynamic semantics, which only uses the type system in a very small number of places, mostly where it is needed due to the requirements of object-oriented polymorphism. I'm confident the report's treatment of polymorphim is correct; elsewhere, I am far less confident.)

Execution of Concrete Programs The file HOL:`concrete_tests` includes more theorems, but of a slightly different nature: exploring the behaviour

of the model when applied to concrete examples. The last test (`t6`) is name resolution on the following, small program, featuring a local class.

```
int x;
int f()
{
  struct s {
    int g() { return x; }
    int x;
  };
  s val;
  val.x = x;
  return val.g();
}
```

The test confirms that this pair of declarations is interpreted as if it were

```
int ::x;
int ::f()
{
  struct s {
    int g() { return s::x; }
    int x;
  };
  s val;
  val.s::x = ::x;
  return val.s::g();
}
```

Even this much is a great deal of work to do by hand, as the length of the proof leading to the `t6` result demonstrates. There is even a little proof automation involved in getting this result, but any tool for working with larger programs would require considerably more engineering effort.

The deliverable also includes a directory `holsrcs/testfiles`, where there is some preliminary work towards the creation of a symbolic evaluator to demonstrate that programs in the model can behave as one might expect. This work builds on the ideas in [1], allowing symbolic exploration of a semantic definition that features non-deterministic branching. For the moment, the code only handles a sequence of external declarations not requiring any expression evaluation, which is very minimalist indeed. Again, more in this vein would require a significant investment of work.

Tools of this form really also require some sort of parser for C++ source code, as having to write out abstract syntax trees by hand is a major annoyance. For example, the `t6` program has to be presented to the tool in the form given in Figure 40. There, one sees that the program is a sequence of external declarations (the `ext_decl` type is defined at the bottom of theory HOL:`statements`). There are two declarations, one for the variable `x`, and one for the function `f`. The body of `f` is a `Block` with a list of two declarations followed by a list of two statements. The first of the two declarations is the desclaration of the struct `s`, with its two member declarations nested inside that.

The abstract syntax for C++ programs is given in theory files HOL:`types`, HOL:`expressions` and HOL:`statements`. One route into the HOL model's abstract syntax might be via the parse-trees generated by `g++`'s front-end.

Social Process The treatment of multiple inheritance in this report owes a great deal to the article by Wasserrab *et al* [15]; similarly, the treatment of templates is informed by Siek and Taha [12]. Both of these resources have appeared in the academic literature, and build in turn on earlier work in their respective areas. I feel confident therefore that my adoption of these formal models here is well-justified. My work on modelling C expressions was also published academically [7], so the underlying C semantics has met with some degree of social examination.

Clearly, it would be ideal to now put the C++ semantics before the public. The academic community's thirst for novelty, and the current absence of any "deep" proofs, might make it difficult to publish the whole semantics academically. My feeling is that the most interesting material for this audience might be name resolution (Section 3). This is the sort of thing that is usually thought trivial, but which is in fact quite complex in C++. The treatment of object lifetimes in the presence of exceptions and object r-values is also probably academically novel.

Publication in other venues would also be desirable. The ISO committee behind the standardisation process is currently obsessing over the next revision to the standard, adding a great deal more complexity to the existing language and aiming to do so before 2010. It seems unlikely therefore that the committee would want to pay much attention to new material that would, to them, appear to have come "out of left field". Simply making the report (and HOL source code) available on an accessible web-page would perhaps be a good first step. As its commissioners, QinetiQ (or the UK Ministry of Defence?) should perhaps decide how to best publicise the work in

```
[Decl (VDec (Signed Int) (Base "x"));
 FnDefn (Signed Int) (Base "f") []
   (Block F
      [VStrDec (Base "s")
         (SOME <|
            ancestors := [];
            fields := [
              (CFnDefn F (Signed Int) (IDName "g") []
                       (SOME
                          (SOME (Ret (EX (Var (Base "x"))
                                         base_se)))),
               F, Public);
              (FldDecl (IDName "x") (Signed Int),
               F, Public)
            ]
           |>);
       VDec (Class (Base "s")) (Base "val")]
      [Standalone
         (EX
          (Assign NONE
                  (SVar (Var (Base "val")) (Base "x"))
                  (Var (Base "x")))
          base_se);
       Ret (EX (FnApp (SVar (Var (Base "val")) (Base "g"))
                      [])
               base_se)])]
```

Figure 40: The Abstract Syntax Corresponding to Test Program t6

these non-academic venues.

7 Omissions and Possible Fixes

The most significant omission in this semantics is a treatment of overloading. This is a complicated feature of the language, but one that is purely syntactic, and one that is checked and resolved entirely by the compiler. If this semantics were to handle overloading, it would be done in Phase 1 (Name Resolution) and Phase 2 (Templates). There a call to a bare `f` would ultimately turn into a call into the `f` whose parameters' types best matched the types of the actual arguments. This resolved call would then be to an exact name and type combination, so that `f` might become `::ns::f(int,char)` for example.

Similarly, there is no treatment of operator overloading. Again, any modelling of this feature would naturally occur in Phases 1 and 2, where calls to operators such as `+` would be resolved into calls to functions over particular types, in particular namespaces and classes.

Two other large omissions in the realm of statics are `const`, and protection statuses (including `friend` functions). In general, the `const`-ness of an expression influences the selection of particular functions to call (more name resolution), and can prevent certain expressions from being written at all. These latter constraints are an important part of the practice of programming with C++, but again, are checked by the compiler. The semantics *does* model the fact that it is undefined behaviour to update memory that has been declared as `const`.

Protection statuses (*i.e.*, the designation of certain fields or base classes as being `public`, `protected` or `private`), are similarly a static mechanism, and have almost no impact on the dynamics of a program. (They make a difference to the behaviour of `dynamic_cast`, and to the dynamic type-matching that is done in exception handlers.) I feel that all of these omissions are justified given the commission to prefer treatment of dynamics rather than static issues.

The semantics does not handle references to functions. Supporting these would require more rules in the dynamic semantics, but these rules will be directly analogous to the rules presented below: wherever an `LV` constructor (which is for l-values of object type) appears, there will need to also be a rule for `FV`, which is for function values. The semantics also doesn't handle initialization of `const` references from r-values.

The model also omits `delete`, placement-`new`, and some of the con-

straints on what may or may not be done with constructors (for example, that in[5, §12.1, para 15]). Other language features completely ignored (mainly on the grounds that they are of less interest) are: enumerated types, `typedef` declarations, unions, bit-field members and the `goto` and `switch` statements (these omissions have all been inherited from the original C model).

A Mechanised Sources

The deliverable consists of a compressed `tar`-file, that when unpacked consists of a directory called `qinetiq-cpp`, which in turn contains four directories

- `holsrcs`, containing the HOL source files of the mechanisation. These files will build with the version of HOL4 present in the Subversion repository at SourceForge, with date `2007-10-13`. See Section A.1 below for instructions on how this version of HOL can be retrieved, and how the deliverable's HOL source files can then be built and checked.

- `talks`, containing the LaTeX source and a PDF for the talk presented at the DARP meeting in Newcastle in April 2006. The source assumes that the LaTeX packages `latex-beamer` and `PSTricks` are available.

- `docs`, containing LaTeX sources and a PDF version of this document, as well as sources for the notes on the earlier deliverables (nos. 1–4).

- `notes`, some C++ source files that illustrate various aspects of C++ behaviour. An accompanying text file explains some of the behaviours.

A.1 Building HOL Source-Files

HOL4 builds on Windows XP, MacOS X and Linux. Though untested on a recent Solaris, we expect it should also build there. The only dependency is on the Moscow ML compiler and interpreter [9]; so HOL should build and run on all the platforms where Moscow ML builds and runs.

Getting HOL From SourceForge To get a particular, dated, version of the HOL4 sources from the Subversion (`svn`) repository, one must issue the command

```
svn co -r date-spec https://hol.svn.sf.net/svnroot/hol/HOL
```

where *date-spec* is the desired date (best specified as an ISO 8601 string) enclosed in braces. The whole should in turn be enclosed in quotes in order to avoid having the braces confuse the command-line shell. For example, `"{2007-10-13}"`. When the `svn` command is issued, the source code is downloaded from SourceForge and put into a directory called `HOL`. The source code (and all the accompanying `svn` meta-data) fits into 200MB.

Once a copy of the sources have been downloaded, further commands can be used to update this copy to correspond to different dates. The commands need to be issued from within the `HOL` directory. The update command is

```
svn update -r date-spec
```

Installing HOL Once the sources have been downloaded, the installation instructions from the page at `http://hol.sourceforge.net` should be followed to build a copy of HOL. An installation of the Moscow ML compiler (v2.01) will also be required.

Building Deliverable Sources When HOL4 has been installed, the `Holmake` program (found in the `HOL/bin` directory) can be run in the `holsrcs` directory of the C++ deliverable to create and check the logical theories.

B Annotated Bibliography

These sources were either used directly in the development of the C++ model, or provide useful background reading.

[1] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *POPL'06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 55–66, New York, NY, USA, 2006. ACM Press.

This article is relevant to C++ only inasmuch as it suggests a technique that might allow reasonably efficient evaluation of a branching, non-deterministic operational semantics.

[2] Clive Feather. A formal model of sequence points and related issues: Working draft. ISO Standards Document JTC1/SC22/WG14 N925, September 2000. Available at `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n925.htm`.

This proposal gives a rigorous description of how C's expression evaluation should work, with particular reference to the possible undefinedness of expressions caused by the sequence point rules.

[3] M. J. C. Gordon and T. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

This book is the ancestor to [4]; it is a complete description, including much reference material, for the (now outdated) HOL88 system.

[4] *The HOL System: Description*, 2007. Available at `http://hol.sourceforge.net`.

A complete description of the HOL4 system, with details about its ML implementation, as well as its logical theories.

[5] *Programming Languages—C++*, 2003. ISO/IEC 14882:2003(E).

The canonical reference for all of the semantics, this is the latest official ISO standard.

[6] Michael Norrish. *C formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998. Also published as Technical Re-

port 453, available from `http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-453.pdf`.

The starting point for the C++ mechanisation. This semantics for C has an awkward treatment of statements, attempting to give statements a big-step semantics at the same time as expressions are evaluated in a small-step style. The new statement model for C++ is described in Section 5.5. The thesis's treatment of the sequence point rule allowing references to variables that are to be updated is also rather complicated. In preference, Clive Feather's semantics has been adopted for the C++ model (see the discussion of assignment in Section 5.4 on page 50).

[7] Michael Norrish. Deterministic expressions in C. In S. Doaitse Swierstra, editor, *Programming languages and systems, 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 147–161. Springer, March 1999.

Essentially a presentation of the main result from Chapter 4 of my thesis [6]; that expressions without sequence points can only evaluate to one possible result.

[8] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

An introductory textbook describing the functional programming language SML. Logical specifications in HOL often resemble functional programs. Additionally, many of the functions used in HOL theories have counterparts in SML. (SML is commonly known as "ML".)

[9] Sergei Romanenko, Claudio Russo, Niels Kokholm, Ken Friis Larsen, and Peter Sestoft. Moscow ML 2.01. Available from `http://www.dina.dk/~sestoft/mosml.html`.

A light-weight implementation of SML, including an interpreter and compiler. The SML implementation required for HOL4 (and also the earlier hol98).

[10] John Rushby. Formal methods and the certification of critical systems. Technical Report CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA, November 1993. Available at `http://www.csl.sri.com/papers/csl-93-7/`.

A well-written introduction to the nature and use of "formal methods" at a variety of levels (ranging from "no use of FM" to "full use of mechanically checked proofs"). Rushby introduces the notion of formal challenge

to a specification, which notion includes what Section 6 calls a sanity theorem.

Rushby's §*2.3 discusses validation of formal specifications generally. Because HOL, cited by Rushby, uses the definitional approach everywhere, "internal consistency" is easy to establish in the context of this C++ project. On the other hand, what Rushby calls "external fidelity" (*§*2.3.2) is much harder, and is validation in the sense: "the specification means what we want/expect/think".*

Rushby's §*3 includes an exhaustive, and by no means out-dated, checklist of features that formal methods tools should exhibit if they are to be practically useful in large verification projects.*

[11] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: effective tool support for the working semanticist. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional programming*, pages 1–12, New York, NY, USA, 2007. ACM.

A recent paper describing a general (prototype) tool allowing a standardised ASCII notation for semantic descriptions to be turned into pretty LaTeX output as well as definitions suitable for consumption by theorem provers such as HOL4, Isabelle/HOL and Coq.

[12] Jeremy Siek and Walid Taha. A semantic analysis of C++ templates. In Dave Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 304–327. Springer, 2006.

A elegant, though somewhat simplistic, presentation of template instantiation in C++. Discussed in Section 4.

[13] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

A description of the history of C++'s development. Some of the discussion is quite useful in describing why certain features of the language are the way they are.

[14] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 2006.

A very complete introduction to C++, explaining how to use the language effectively and in a principled way. Detailed, ISO standard style discussion of semantics is present haphazardly.

[15] Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *OOPSLA '06: Object oriented programming, systems, languages, and applications*. ACM Press, 2006. Available from `http://isabelle.in.tum.de/~nipkow/pubs/oopsla06.html`.

This article describes an elegant method of modelling multiple inheritance in C++, without having to explicitly describe the construction of vtables. Instead, references to objects are accompanied by paths*, allowing one to simultaneously identify the static and dynamic type of an object.*

The article very carefully lays out the way polymorphic names (those associated with `virtual` functions) are resolved at run-time, referring to objects' dynamic and static types.

Index

' (finite map application), 7
++ (list operation), 6
:= (record update), 7
^ (path concatenation), 89
|+ (finite map update), 7

Addr, 46
algebraic types, 8
Assign, 50

bad_cast exception, 81
Base function, 14
base_se, **45**
Block, 59, 91
break, 56
Break (stmt constructor), 56

CApBinary, 44, 51
CApUnary, 44
case expressions in HOL, 9
Catch, 67
Cchar, 40
CIf, 57, 65
cinfo, 48
class r-values, 100–109
    as initializers, 107–109
    as parameters, 107–108
    creation and return of, 102–107
class_part, 76
ClearExn, 68, 70
CLoop, 56
Cnum, 40
CommaSep, 45
conditional expressions, 6
CONS (list operation), 6
const, 114
ConstructedVal (expr constructor), 46, 105
ConstructorFVal, 93
constructors, of algebraic types, 8
Cont (stmt constructor), 56
continue, 56
CopyInit, 62
CPP_ID type, 14
current_exns (state field), 68, 69

declmng relation, 59, 61
Deref, 47
derive_objid, 50
DirectInit0, 62
dynamic semantics, 13, 36–100
dynamic_cast, 79

ECompVal, **39**
EL (list operation), 6
EmptyStmt, 55, 60
env (state field), 21, 59
EThrow, 64
EX, 37, 61
exceptions, 56, 60, 64–71
explicit type conversion, 107
ExtE, 37

FDOM, 7
Feather, Clive, 50
final_stmt, **55**
final_value, **45**, 55
finite maps, 7
FnApp, 52
FnApp_sqpt, 52, 102
FOLDL (list operation), 6
FST, 6
FVal, 41

genv (state field), 21

HD (list operation), 6
HOL definitions, 8–12

IDComp, 14
IDConstant, 14
identifiers, 14–15
if-then-else in HOL, 6
inductive definitions in HOL, 10–12
initmap (state field), 64
is_exnval, 66
is_null_se, **45**
is_qualified, 76

LAST (list operation), 6
let expressions, 5
lookup_type, 41
loops, 56
LVal, **39**, 46
    and dynamic types, 73
lval2rval, 43, 84

MAP (list operation), 6
MEM (list operation), 6
MemAddr, 46
member functions
    calling, 52
    non-static, 76
    overriders, 89
    pointers to, 48–50
    static, 76
    virtual, 73
mk_exn, 66
mk_member, 75
multiple inheritance, 86–89

name resolution, 12, 19–28
    validation of, 111
New, 100
nonclass_conversion, 64, 80
NONE (option constructor), 7

object slicing, 106
OffsetDeref, 49
overloading, 36, 43, 114

phase1_expr, 21
phase1_fndefn, 28

record_creation, 105
records (in HOL), 7
Ret, 55
return_cont, 52, 83
rule (declaration dynamics)
    decl-class-copy-finishes, 109
    decl-fncall-copy-elision, 108
    decl-init-start-eval-copy, 62
    decl-init-start-eval-dnonclass, 62
    decl-parameter-copy-elision, 108
    decl-vdec-nonclass, 62
    decl-vdecinit-evaluation, 63
    decl-vdecinit-finish, 64
    decl-vdecinit-finish-ref, 86
    decl-vdecinit-lval2rval, 63, 85
rule (dynamic)
    addr-lvalue, 46
    allocate-rvrt, 103
    and-false, 45
    and-true, 45
    assign-completes, 51
    assign-op-assign, 51
    bare-throw-fails, 70
    bare-throw-succeeds, 69
    binop-computes, 44
    binop-fails, 44
    block-declmng, 59
    block-declmng-exception, 66
    block-entry, 59
    block-exit, 61

block-interrupted, 60, 65
block-stmt-evaluate, 60
catch-all, 67
catch-normal-stmt-passes, 67
catch-specific-type-matches, 68
catch-specific-type-nomatch, 69
catch-stmt-empty-hnds, 69
catch-stmt-evaluation, 67
char-literal, 40
clear-exn, 70
comma-progresses, 45
constructor-function-call, 94–97
deref-fnptr, 47
deref-objptr, 47
deref-objptr-fails, 47
dyncast-base-other-ref, 81
dyncast-derived-base-ref, 79
econtext-expr, 42
econtext-undefinedness, 42
expression-throw-none, 69
expression-throw-some, 64
fcontext, 42
fnapp-lval2rval, 82
function-call, 52–54, 77
function-call-sqpt, 83
if-eval-guard, 57
if-exception, 65
if-false, 57
if-true, 57
loop, 56
lvcontext, 43
mem-addr-nonstatic, 48
mem-addr-static-nonfn, 46
new-nonclass, 100
new-simple-class, 101
noscope, 40
nstatic-data-field-select, 78
nstatic-fn-member-select, 77
number-literal, 40
offset-deref, 49
offset-deref-fails, 50
ret-class-rvalue, 105
ret-construct-rvalue, 104
ret-pass-rvrt, 106
return-eval-under, 55
return-lval2rval, 84
return-lvalue, 85
return-rvalue, 54, 84
standalone-evaluates, 57
standalone-finishes, 57
static-fn-member-select, 76
this, 41
trap-break-catches, 58
trap-break-pass-continue, 58
trap-continue-catches, 58
trap-continue-pass-break, 58
trap-emptystmt-passes, 58
trap-exn-passes, 65
trap-ret-passes, 58
trap-stmt-evaluation, 58
unop-computes, 44
unop-fails, 44
var-to-fvalue, 10, 41
var-to-lvalue, 41
virtual-fn-member-select, 75
rvstk (state field), 54, 84, 104

sanity theorems, 110
sel4 function, 99
selects-via relation, 87, 89
slicing, 106
SND, 6
SOME (option constructor), 7
ST, 37, 61
stack (state component), 97
stack (state field), 59
Standalone, 56

static_type, 46

templates, 12, 28–36
temporaries, *see* class r-values
THE (HOL function), 7
This, 41
this, 52
Throw, 55, 64
TL (list operation), 6
TL (list operation) function, 55
Trap, 56

UndefinedExpr, 39, 42
updated_by (record update), 8

val2mem, 63
valid_econtext, 42
valid_fvcontext, 42
valid_lvcontext, 43, 82
Var, 23, 41
VDec, 61
VDecInit, 61
VDecInitA, 62
vdeclare, 62
vtables, 73, 79

with (record update), 7

ZIP (list operation), 6