A literate, executable, denotational semantics of simple C++ declarations

Joseph Reynolds

TR #93-15
May, 1993

Keywords: denotational semantics, C++, literate programming, declarations, executable semantics, Standard ML
1992 CR Categories: D.1.m [*Programming Techniques*] Miscellaneous — Literate programming; D.3.1 [*Programming Languages*] Formal Definitions and Theory — Semantics, syntax; D.3.3 [*Programming Languages*] Language Constructs and Features — Data types and structures; F.3.2 [*Logics and Meaning of Programs*] Semantics of Programming Languages — Denotational semantics.

Department of Computer Science
226 Atanasoff Hall
Iowa Sate University
Ames, Iowa 50011-1040, USA

A literate, executable, denotational semantics of simple C++ declarations

Joseph Reynolds

May 12, 1993

Abstract

Denotational semantics are developed for simple C++ declarations and are implemented in a functional language using a literate programming style.

1 Introduction

Computer languages are difficult to learn and understand. Part of the problem is that there is (usually) no formal statement of what various language constructs mean. Denotational semantics [Sch86] provides a formal framework for developing this meaning. Yet there is no denotational semantics for C++.

This paper provides a denotational semantics for some simple C++ declarations.

1.1 Audience

This paper is intended for a general computer science audience interested in denotational semantics and C++ declarations. A familiarity with C++ and and SML is assumed.

Serious students of C++ can see how memory and environments are used. Students of denotational semantics can use this as an example of a semantics of a "real" language. Designers of C++-like languages can use this to guide how new features will work. Finally, C++ implementors and designers of C++ specification languages can use this as a formal definition of C++.

1.2 C++ declarations

The semantics given here model a small but usable subset of C++'s variable and class declarations. Currently they model variable declarations involving primitive types, pointers, arrays, and classes. No modifiers or initializers can

be specified. Class declarations are limited to `public` members only, and members must be objects. Nested class definitions are not allowed, although class members may be class types.

Another restriction is that variables and classes must have unique names. E.g., you cannot say `class foo {} foo;` because then `foo` would denote both a variable and a class.

1.3 Denotational Semantics

In denotational semantics, programs are mapped directly to their meanings as in [Sch86]. In practice this means a pre-digested tree is mapped onto well-understood environment and memory operations. It is the particular mapping that defines the language and provides the semantics.

1.4 Using SML to implement semantics

The semantics in this paper are implemented in Standard ML (SML) [Pau91]. (ML was originally the name of a meta language for a theorem prover system [GMW79].) This language is secure from type subversion and corruption of the run-time environment [Pau91, Section 1.5]. These features should be considered essential in any implementation of a formal system.

SML `signatures` are used to describe the semantic domains. They provide a natural way to modularize the code and separate the system's functionality from implementation details.

Also, many of these signatures contain an SML `datatype`. By putting the datatype in the signature, clients can use the type's constructors and observers without additional effort on the ML programmer's part. For example, in the following we advertise that clients can create values of `Foo_Type` using constructors `Bar` and `Baz`, and can observe the result of `fooey` using SML's pattern matching facilities.

⟨*Example using a SML datatype* 2⟩≡

```
signature Foo_Sig =
    sig
        datatype Foo_Type =
            Bar of Int
          | Baz of Int
        val fooey: Foo_Type * Foo_Type -> Foo_Type
    end
```

Root module (not used in this document).

We say that a function *mutates* a value when that value is an input to the function and a similar value of the same type is an output of the function. For instance, consider the signature of a typical memory update function:

`update: Location * Value * Memory -> Memory`. We say that `update` mutates memory when, in fact, no SML value is mutated. Memory is mutated only in the sense that the new value is (usually) similar to the old value and the previous value is never again referred to. SML's imperative capability is not used in these semantics because of the well-known difficulty in understanding the scope of the effect of an assignment.

1.5 Literate Programming

The program described here is written using a literate programming technique. This technique combines documentation and source code into one coherent document. In this way, you are assured that the documentation is up to date.

The literate programming tool used here is *noweb* [Ram92]. With this tool, a text file contains the literate program which looks very much like the LaTeX input that created this paper. The file can be processed by *noweave* to produce this file. And it can be processed by *notangle* to produce the source code.

Noweb sees this paper as a series of "chunks" embedded in the text of the paper. Each chunk begins with an identifier consisting of a short description in angle braces. Noweb also supplies the page number where the chunk is defined. The body of the chunk is typeset in a monospaced font and may contain references to other chunks by giving those chunks' identifiers.

Noweb forms the completed program by pasting the chunks together heirarchically beginning with the special chunk whose name is simply an asterisk. By recursively following the chunks referred to in that chunk and substituting the text literally, noweb pieces together the entire program.

Noweb chunks in this document are given descriptive names. Pieces of `type`s, `typedef`s, `fun`ctions, `val`ues, `signature`s, and `structure`s are identified by chunks that have the appropriate keyword in their name. Chunks that contain complete SML declarations have names that end with the word "declaration" or "definition." Other chunks contain smaller syntactic units, not top-level declarations, and are used to break up the code into manageable pieces.

1.6 Organization

The rest of this paper is organized around the presentation of the semantics. First, the syntactic and semantic domains are described. Next the semantic domains are implemented. Then the valuation functions are given. These are followed by a short section on testing, related work, conclusions, and acknowledgements.

The overall structure of the SML source code is given by the following. Note here that this follows the order given in this paper—this will not always be the case for other sections. (Recall that the asterisk tells noweb this is the root chunk, i.e., noweb starts here when pasting together the code.)

⟨* 4⟩≡
```
(* Generated code--do not edit *)
```
⟨*misc definitions* 28⟩
⟨*syntactic domain declarations* 4⟩
⟨*semantic domain declarations* 6⟩
⟨*domain implementations* 11⟩
⟨*Valuation function implementations* 29⟩
⟨*memory examiner* 35⟩
Root module (not used in this document).

2 Syntactic Domains

We begin with a discussion of the syntactic domains. There are two. The first allows creation of types. The second allows the C++ programmer to declare variables.

⟨*syntactic domain declarations* 4⟩≡
⟨*Data-Type type declaration* 5⟩
⟨*Declaration datatype declaration* 4⟩
⟨*Data-Type signature declaration* 5⟩
This code is used on page 4.

C++ declarations are given using an abstract syntax [Sch86, Section 1.1]. For example, the C++ declaration `int i;` is represented by the SML expression `var_decl ("i", IntType)`. A list of such declarations is represented by an ordinary SML list.

2.1 Declaration Syntax

The declarations considered here are either simple variable declarations or simple class definitions. By simple class definitions, we mean classes that do not contain protected or private members, member functions, or type definitions. This corresponds to the Classic C notion of a `struct`.

The following SML datatype defines the type of C++ declarations. For a variable, we need only its name and type. For a structure, we need its name and a list of its field names and what types they store: essentially a list of declarations. Note that this definition allows nested class definitions.

The actual data type is broken out here and given separately so that it can be used in the `Data_Type` signature and also in the `Data_Type` structure to be given later. This is an example of where the literate programming technique facilitates code re-use not present in the target language.

⟨*Declaration datatype declaration* 4⟩≡
```
datatype Declaration =
```

```
    var_decl     of Identifier * Data_Type
  | class_decl   of Identifier * Declaration list
```
This code is used on page 4.

2.2 Data Types

This section gives the abstract syntax for C++ data types—the types for C++ variables. Note that the type is implemented as a SML `datatype` which provides the clients with constructors and observers. The only operation is `sizeof` which implements the C++ operator of the same name.

Note that the `ClassOf` constructor takes the *name* of a class, not a description of the class itself. Anonymous classes are not allowed in these semantics. This also means that the environment is needed in the `sizeof` function.

For example `var_decl("i", IntType)` represents `int i;`.

⟨*Data-Type type declaration* 5⟩≡
```
datatype Data_Type =
    CharType
  | IntType
  | FloatType
  | DoubleType
  | PtrToType of Data_Type
  | ArrayOfType of Nat * Data_Type
  | ClassOfType of Identifier
```
This code is used on page 4.

⟨*Data-Type signature declaration* 5⟩≡
```
signature Data_Type_Sig =
    sig
        type Environment
        type Data_Type
        val sizeof: Data_Type * Environment -> int
    end;
```
This code is used on page 4.

3 Semantic Domains

We now turn our attention to the semantic domains. The mundane domains of truth values, identifiers, and numbers are modeled with the built-in capabilities of SML. The environment and denotable values provide the usual sort of environment capabilities [Sch86]. Finally, memory and storable values provide the rich set of capabilities in C++ data structures. These last set of capabilities are most interesting.

This section presents the capabilities only—the interfaces for users of the system. The implementations are given later.

The following is an outline of all the signatures for the semantic domains. These are discussed later in this section.

⟨*semantic domain declarations* 6⟩≡
```
(***** semantic domains *****)
```
⟨*environment signature declaration* 7⟩
⟨*memory signature declaration* 8⟩
⟨*basic value signature declaration* 9⟩
⟨*array signature declaration* 10⟩
⟨*pointer signature declaration* 10⟩
⟨*object signature declaration* 11⟩
This code is used on page 4.

3.1 Denotable Values

For our purposes, a C++ name is either the name of a variable or the name of a class.

For a variable, we are interested in its type and where its value is stored. Since our model of memory keeps track of the type of data stored at each location, the denotation of a variable is simply the `Location` of its value. The types for classes are discussed in the next paragraph.

⟨*Denotable-Value type declaration* 6⟩≡
```
datatype Denotable_Value =
    Variable of Location
  | Class of class_info
```
This code is used on page 12.

The `class_info` type holds information about a class. It consists of the size of a class and a function to construct new objects in memory. These are created when the class is declared and are discussed with the class valuation functions.

⟨*Class type declarations* 6⟩≡
```
type object_constructor_type = Memory -> Location * Memory
type class_info = int * object_constructor_type
```
This code is used on page 12.

3.2 Environment

The environment maps from `Identifier` to `Denotable_Value`. For example, the following C++ declarations would add two new mappings to the environment: the name "`Coord`" to a SML `Class` which represents the equivalent C++

class, and the name "`loc`" to a SML `Variable` which represents the equivalent C++ variable.

```
struct Coord { int x, y; }; Coord loc;
```

Functions `new`, `add`, and `lookup` provide the usual capabilities [Sch86, Chapter 7]. `lookup` raises the exception `symbol_not_found` if the symbol it is looking for is not there.

The operations `enter_block` and `is_local` handle C++'s block structure.

⟨*environment signature declaration 7*⟩≡

```
signature Environment_Sig =
  sig
    type         Environment
    type         Denotable_Value
    exception    symbol_not_found
    val empty_env:      Environment
    val add:            Environment * Identifier * Denotable_Value
                                                   -> Environment
    val lookup:         Environment * Identifier -> Denotable_Value
    val is_in_env:      Environment * Identifier -> bool
    val in_local_scope: Environment * Identifier -> bool
    val enter_block:    Environment -> Environment
  end
```

This code is used on page 6.

3.3 Storable Value

The C++ memory stores several different kinds of values: basic values, arrays, pointers, and objects. These are all alternatives of the `Storable_Value` type, discussed in detail below.

These types are all mutually recursively defined with Memory and the specific storable values. The SML keyword `datatype` is given when all these types are all defined together, not here.

⟨*Storable-Value type 7*⟩≡

```
(* datatype *) Storable_Value =
    BV of Basic_Value
  | AV of Array_Value
  | PV of Pointer_Value
  | OV of Object_Value
```

This code is used on page 12.

There are three functions over all kinds of storable values: `alloc`, `sizeof`, and `copy`. Alloc creates a storable value from a data type. Sizeof yields the size, in bytes, of a storable value. And copy copies a storable value into some memory location.

⟨*alloc function signature* 8⟩≡
```
Data_Type * Memory * Environment -> Location * Memory
```
This code is used on page 16.

⟨*sizeof function signature* 8⟩≡
```
Storable_Value -> int
```
This code is used on page 16.

⟨*copy function signature* 8⟩≡
```
Storable_Value * Location * Memory -> Memory
```
This code is used on page 16.

3.4 Memory

Memory provides the usual sort of mapping from `Location` to `Storable_Value` [Sch86, Chapter 5]. The value `initial` and functions `alloc` and `read` should be self-explanatory. Function `is_same_location` provides an equality test for locations. The only wrinkle is that we require clients to make up values when they allocate cells even if they want to consider a cell uninitialized. (We do this so we will have a type for the cell.) The clients must supply either the value `initialized` or `not_initialized` when they allocate cells.

The other elements of the memory domain is for the convenience of the rest of the program. The location `null_loc` provides the null pointer value in C++. Location `undef_loc` provides the initial value for an uninitialized pointer. The exception `read_out_of_bounds` is raised only when a read or write operation requests a non-existant memory location. Finally, the exception `uninitialized_memory` is raised when attempting to read from a location that does not contain a value.

Our memory is primitive in that that deals only with storing and retrieving information, not how that information is aggregated. The array, pointer, and object domains each have to worry about how they aggregate their values.

⟨*memory signature declaration* 8⟩≡
```
signature Memory_Sig =
    sig
        type     Location
        type     Storable_Value
        type     Memory
        exception   read_out_of_bounds
        exception   uninitialized_memory
        val initialized:     bool
        val not_initialized: bool
        val null_loc:        Location
        val undef_loc:       Location
        val is_same_location: Location * Location -> bool
```

```
        val initial:   Memory
        val alloc:     Memory * Storable_Value * bool -> Location * Memory
        val write:     Memory * Location * Storable_Value -> Memory
        val read:      Memory * Location -> Storable_Value
    end;
```

This code is used on page 6.

3.5 Basic Value

The `Basic_Value` domain holds those C++ types that do not refer to other C++ types. It is implemented as an SML `datatype`. Once again, here, we use noweb to duplicate code. This appears both in the signature and in the structure.

⟨*Basic-Value datatype declaration* 9⟩≡

```
datatype Basic_Value =
    IntVal of int
  | CharVal of int
  | FloatVal of real
  | DoubleVal of real
```

This code is used on pages 9, 12, and 19.

⟨*basic value signature declaration* 9⟩≡

```
signature Basic_Sig =
    sig
        ⟨Basic-Value datatype declaration 9⟩
    end;
```

This code is used on page 6.

3.6 Array Value

The `Array_Value` domain models C++ arrays.

Since C++ uses the indirect model of arrays, so do these semantics. An array is a sequence of `Location`s. This makes it easy to take the address of an element as in `&myarray[3]`. (Recall that C++ arrays are one dimensional, but elements can themselves be arrays.)

The `alloc` function takes a memory, the type of array elements, and the array size. It reserves cells for the array and returns a new memory and the location of where the new array was created. Indexing yields one of the reserved locations. Hence reading or writing an array element involves first indexing to get the location, then manipulating memory directly.

The operation `is_same_array_value` checks if two arrays are the same array. This is used, for instance, to determine if pointer comparison is meaningful.

(Comparing two pointers is meaningful only if the pointers are either both pointing to the same location or both pointing into the same array [ES90, Section 5.9].)

⟨*array signature declaration* 10⟩≡
```
signature Array_Sig =
    sig
        type Array_Value
        type Memory
        type Location
        val alloc: Data_Type * Nat * Memory -> Array_Value * Memory
        val index: Array_Value * Nat -> Location
        val is_same_array_value: Array_Value * Array_Value -> bool
    end;
```
This code is used on page 6.

3.7 Pointer Value

Pointers refer to other storable values. They can be thought of abstractions on `Location`.

Function `create`, when given a type, creates an uninitialized pointer. The function `make_ref`, when given a `Location` and `Memory`, mutates a pointer refer to the new location. Its inverse, `deref`, gets back the location from the pointer. The `Memory` is required as an argument to `make_ref` so type checking can be done. If `make_ref` is passed the location of an entire array it silently forms a pointer to the zeroth element of that array.

The `increment` operation traverses the array forward or backward. If it goes past the beginning or more than one past the end of the array, it raises the exception `pointer_escaped_array`. Any attempt to `deref` a pointer one past the end of an array will raise the exception `deref_outside_array`. Using `increment` on a non-arry pointer will raise the exception `increment_undefined`.

When two pointers are both pointing into the same array or both at the same non-array value, `difference` returns the pointers' distance in terms of array elements. Otherwise it raises the exception `ptrdiff_undefined`.

⟨*pointer signature declaration* 10⟩≡
```
signature Pointer_Sig =
    sig
        type Pointer_Value
        type Memory
        type Location
        exception ptrdiff_undefined
        exception pointer_escaped_array
        exception deref_outside_array
```

```
        exception increment_undefined

        val create: Data_Type -> Pointer_Value
        val make_ref: Pointer_Value * Location * Memory -> Pointer_Value
        val deref: Pointer_Value -> Location

        val difference: Pointer_Value * Pointer_Value -> int
        val increment: Pointer_Value * int -> Pointer_Value
    end;
```

This code is used on page 6.

3.8 Object Value

Objects are instances of C++ classes. When we say C++ classes, we also mean C++ structures since the two are so similar. Recall also that our model only handles the Classic C features of structs.

We model objects using the indirect model: as mappings from field names to locations. The `dot` function selects a member of an object. If the field name is incorrect, the exception `not_member` is raised.

⟨*object signature declaration* 11⟩≡

```
signature Object_Sig =
    sig
        type      Object_Value
        type      Memory
        type      Location
        exception not_member
        val       dot: Object_Value * Identifier -> Location
    end;
```

This code is used on page 6.

The instantiation and copy functions are handled by the object's class definition. They are discussed with the *Class* implementation, not here.

4 Semantic Domain Implementations

In this section we implement the semantic domains. Overall, we have several semantic domains that need to share types. These are declared first. Next, Memory and Environment are implemented in terms of these. Finally, the subdomains and Storable-Values themselves are given.

⟨*domain implementations* 11⟩≡

```
(***** domain implementations *****)
⟨semantic domain implementation type declarations 12⟩
```

⟨*Memory implementation* 17⟩
⟨*Environment implementation* 13⟩
⟨*data type implementation* 12⟩
⟨*basic value structure declaration* 19⟩
⟨*array structure declaration* 20⟩
⟨*pointer structure declaration* 23⟩
⟨*object structure declaration* 27⟩
⟨*Storable-Value implementation* 16⟩
This code is used on page 4.

The outline of all semantic types is given below. The details are discussed with each domain. Most of these have to be in the order given here. As is usual in denotational semantics, there are many mutually-recursive types; these are given here with the **and** and **withtype** keywords.

⟨*semantic domain implementation type declarations* 12⟩≡
```
⟨Location type declaration 17⟩
⟨Basic-Value datatype declaration 9⟩
datatype ⟨Pointer-Value type 24⟩
      and ⟨Storable-Value type 7⟩
withtype ⟨Memory type 17⟩
      and ⟨Array-Value type 20⟩
      and ⟨auxilliary object types 31⟩
      and ⟨Object-Value type 27⟩
⟨Class type declarations 6⟩
⟨Denotable-Value type declaration 6⟩
⟨Environment type declaration 13⟩
```
This code is used on page 11.

4.1 Data Type Implementation

The type `Data_Type` has a simple implementation; the SML `datatype` did most of the work. We simply use that here and implement the `sizeof` function.

⟨*data type implementation* 12⟩≡
```
structure Data_Type_Struct : Data_Type_Sig =
    struct
        type Environment = Environment
        type Data_Type = Data_Type
        ⟨data type sizeof function declaration 13⟩
    end;
```
This code is used on page 12.

The `sizeof` function gives the size of each type. Its second argument, the environment, is necessary to look up a class by name.

⟨*data type sizeof function declaration* 13⟩≡
```
fun sizeof(IntType, _) = 2
  | sizeof(CharType, _) = 1
  | sizeof(FloatType, _) = 4
  | sizeof(DoubleType, _) = 8
  | sizeof(PtrToType(_), _) = 4
  | sizeof(ArrayOfType(size,t), env) =
       size * sizeof(t, env)
  | sizeof(ClassOfType(name), env) =
       let val Class(size, _) = lookup(env, name) in
           size
       end
```
This code is used on page 12.

4.2 Environment Implementation

Our environment implementation consists of the environment structure opened up so its components can be used easily.

⟨*Environment implementation* 13⟩≡
```
⟨environment structure declaration 13⟩
open Environment_Struct;
```
This code is used on page 12.

We implement environments in the usual way: as functions from identifiers to denotable values.

⟨*Environment type declaration* 13⟩≡
```
type Environment = Identifier -> Denotable_Value
```
This code is used on page 12.

To implement scoping, we use an exception, `end_of_block`. Raising this exception means "the operation could not be carried out within the most local block." The exception returns the next-most local environment so further work can be done. Clients will never see this exception; they are all handled within environment functions. See `enter_block` and `is_local` for details.

⟨*environment structure declaration* 13⟩≡
```
structure Environment_Struct : Environment_Sig =
    struct
        type Denotable_Value = Denotable_Value
        type Environment = Environment
        exception symbol_not_found
        exception end_of_block of Environment
```

⟨*environment empty env val declaration* 15⟩
⟨*environment add function declaration* 15⟩
⟨*environment lookup function declaration* 14⟩
⟨*environment is-in-env function declaration* 14⟩
⟨*environment is-local function declaration* 14⟩
⟨*environment enter block function declaration* 15⟩
```
    end;
```
This code is used on page 13.

Looking up a symbol amounts to applying the environment to the symbol. If the symbol is not in the top-level block, there will be an `end_of_block` exception. We handle this by looking up the symbol in the next block down. If the symbol is undefined, the exception `symbol_not_found` is raised for the client to handle.

⟨*environment lookup function declaration* 14⟩≡
```
fun lookup(env,id) =
    env(id)
    handle end_of_block env2 => lookup(env2,id)
```
This code is used on page 14.

Checking if a symbol is defined amounts to looking it up. If found, the function returns true. Otherwise the `symbol_not_found` exception will be raised and the function returns false.

⟨*environment is-in-env function declaration* 14⟩≡
```
fun is_in_env(env,id) =
    let val _ = lookup(env, id) in
        true
    end
    handle symbol_not_found => false
```
This code is used on page 14.

Checking if a symbol has a local definition is similar to looking it up. If it is in the most local block, then it will be found without raising any exceptions and the function returns true. Otherwise an `end_of_block` exception will be raised and the function returns false.

⟨*environment is-local function declaration* 14⟩≡
```
fun in_local_scope(env,id) =
    let val info = env(id) in
        true
    end
  handle end_of_block _ => false
```
This code is used on page 14.

To enter a new block we create a function which raises the `end_of_block` exception and carries with it the lower-level blocks.

⟨*environment enter block function declaration* 15⟩≡
```
fun enter_block(env) =
    fn(id) => raise end_of_block(env)
```
This code is used on page 14.

The empty environment contains one block with no symbols defined.

⟨*environment empty env val declaration* 15⟩≡
```
val empty_env =
    fn(id) => raise end_of_block(fn(id) => raise symbol_not_found)
```
This code is used on page 14.

Adding a binding to an environment is done in the usual way. We simply add the mapping to the function.

⟨*environment add function declaration* 15⟩≡
```
fun add(env,id,info) =
    update(env, id, info)
```
This code is used on page 14.

The function above uses the "update" function to make incremental changes in the behavior of a function. The implementation is given here although it is not part of the environment domain; so this chunk is pasted into the "miscellaneous" section of the program *noweb* builds.

The general-purpose update function takes three arguments: the function to be updated, an element of the domain of the function, and its new value. It constructs a new function such that $update(f, x, y)$ is defined as f updated so that x maps to y.

⟨*general-purpose function update declaration* 15⟩≡
```
fun update(oldfun, domain_element, new_value) =
    fn(x) => if x=domain_element then new_value
             else oldfun(x)
```
This code is used on page 28.

4.3 Storable Value Implementation

Recall that the `Storable_Value` datatype is nothing more than a collection of other types, each with their own implementations. What is left to do here is to tie the `alloc`, `copy`, and `sizeof` function implementations together. These do little more than shunt the work off to the appropriate functions.

4.4 Memory Implementation

To implement memory, we complete the memory structure and make some shorthand notation.

⟨*Memory implementation* 17⟩≡
```
⟨Memory structure declaration 17⟩
val allocmem = Memory_Struct.alloc
val readmem = Memory_Struct.read
val writemem = Memory_Struct.write
val initialized = Memory_Struct.initialized
val not_initialized = Memory_Struct.not_initialized
val is_same_location = Memory_Struct.is_same_location
val undef_loc = Memory_Struct.undef_loc
val null_loc = Memory_Struct.null_loc
```
This code is used on page 12.

Memory is implemented in the usual way [Sch86, Section 5.1]. Locations are represented by integers. Memory is represented by a function from `Location` to `Storable_Value`. We also keep a separate component to generate unique locations, so `Memory` is a tuple containing the function and an integer which represents the next-free location.

⟨*Location type declaration* 17⟩≡
```
type Location = int
```
This code is used on page 12.

⟨*Memory type* 17⟩≡
```
Memory = Location * (Location -> Storable_Value)
```
This code is used on page 12.

Our goal in this section is to implement the memory structure.

⟨*Memory structure declaration* 17⟩≡
```
structure Memory_Struct : Memory_Sig =
    struct
        type Location = Location
        type Storable_Value = Storable_Value
        type Memory = Memory
        exception read_out_of_bounds
        exception uninitialized_memory

        ⟨initialized and not-initialized value declarations 18⟩
        ⟨the null location declaration 18⟩
        ⟨the undefined location declaration 18⟩
        ⟨the memory is-same-location function definition 18⟩
```

⟨*the initial memory value declaration* 18⟩
⟨*the memory alloc function declaration* 19⟩
⟨*the memory write function declaration* 19⟩
⟨*the memory read function declaration* 18⟩
```
end;
```
This code is used on page 17.

The following two values provide mnemonics for when memory is allocated. Recall that a storable value is required when allocating memory. Clients must say whether this value initializes memory or not.

⟨*initialized and not-initialized value declarations* 18⟩≡
```
val initialized = true
val not_initialized = false
```
This code is used on page 17.

In this model, it is easy to create special locations.

⟨*the null location declaration* 18⟩≡
```
val null_loc = 0
```
This code is used on page 17.

⟨*the undefined location declaration* 18⟩≡
```
val undef_loc = 1
```
This code is used on page 17.

The initial memory begins allocating at the location represented by the SML integer 2 and contains the function that rejects every reference.

⟨*the initial memory value declaration* 18⟩≡
```
val initial = (2, fn(loc) => raise read_out_of_bounds)
```
This code is used on page 18.

Checking if two locations are really the same location is as easy as comparing their integer locations.

⟨*the memory is-same-location function definition* 18⟩≡
```
fun is_same_location(loc1, loc2) =
    loc1 = loc2
```
This code is used on page 17.

As a consequence of the memory type, the memory read function is very simple. We have only to apply the location to the memory mapping.

⟨*the memory read function declaration* 18⟩≡
```
fun read((_, mem_fn), loc) = mem_fn(loc)
```
This code is used on page 18.

Writing to some location involves updating the memory mapping so that the location maps to its new value. The next-free counter is not changed.

⟨*the memory write function declaration* 19⟩≡
```
fun write((size,f), loc, new_value) =
    (size, update(f, loc, new_value))
```
This code is used on page 18.

Allocating a new location is only a bit more involved. Here we must return a pair containing the new location and the new memory. The new location is readily available from the old memory. And the new memory is easily constructed by incrementing the next-free counter and constructing the appropriate function.

⟨*the memory alloc function declaration* 19⟩≡
```
fun alloc((size,f), value, is_init) =
    (size,                                (* the new location *)
     (size+1,                             (* the new memory size *)
       fn(loc) => if loc=size             (* the new memory function *)
                  then if is_init
                       then value
                       else raise uninitialized_memory
                  else f loc))
```
This code is used on page 18.

4.5 Basic Value Implementation

To complete the Basic Value Implementation we have only to complete the alloc, sizeof, and copy functions.

As already given in the signature, we model the C++ basic values as SML values. We simply repeat that datatype here.

⟨*basic value structure declaration* 19⟩≡
```
structure Cxx_Basic : Basic_Sig =
    struct
        ⟨Basic-Value datatype declaration 9⟩
    end
```
This code is used on page 12.

Finding the sizeof a basic value is trivial. Note that the sizes used here are implementation dependent. For instance, some machines have 4 byte integers. The language standard does not specify a size, only that `sizeof(char) == 1`.

⟨*Basic-Value sizeof function* 19⟩≡
```
    (BV(IntVal(_))) => 2
```

```
  | (BV(CharVal(_))) => 1
  | (BV(FloatVal(_))) => 4
  | (BV(DoubleVal(_))) => 8
```
This code is used on page 16.

Allocating memory for a basic value is as simple as allocating a chunk of memory. We have to make up a dummy value of the correct type to store in the memory. And we tell Memory that the cell is `not_initialized`. Recall that this code is just part of the unified `alloc` function.

⟨*Basic-Value alloc function* 20⟩≡
```
  (* val rec alloc = fn *)
    (CharType, mem, env) =>
          allocmem(mem, BV(CharVal(99)), not_initialized)
  | (IntType, mem, env) =>
          allocmem(mem, BV(IntVal(99)), not_initialized)
  | (FloatType, mem, env) =>
          allocmem(mem, BV(FloatVal(99.0)), not_initialized)
  | (DoubleType, mem, env) =>
          allocmem(mem, BV(DoubleVal(99.0)), not_initialized)
```
This code is used on page 16.

Copying a basic value to some location is as simple as writing its value there.

⟨*Basic-Value copy function* 20⟩≡
```
  (BV(x), loc, mem) =>
      writemem(mem, loc, BV(x))
```
This code is used on page 16.

4.6 Array Value Implementation

Array values are basically mappings from integers to locations. Along with this function we also store the number of elements in the array, the type of the elements, and the `sizeof` the array.

⟨*Array-Value type* 20⟩≡
```
  Array_Value = Nat * (Nat -> Location) * Data_Type * Nat
```
This code is used on page 12.

Our goal in this section is to complete the `Array_Struct` and to write the alloc, copy, and sizeof functions for arrays.

⟨*array structure declaration* 20⟩≡
```
  structure Array_Struct =
      struct
```

```
        type Array_Value = Array_Value
        type Memory = Memory
        type Location = Location
        exception array_range

        ⟨array index function declaration 21⟩
        ⟨is-same-array-value function declaration 21⟩
    end;
```
This code is used on page 12.

The index function is straightforward given our array types.

⟨*array index function declaration* 21⟩≡
```
fun index((_,index_fn,_,_),index) = index_fn(index)
```
This code is used on page 21.

The function `is_same` returns true if and only if its inputs are the same array. This is true exactly when the locations of the zeroth elements coincide. (Recall that the indexing function does not change after the array is created.)

⟨*is-same-array-value function declaration* 21⟩≡
```
fun is_same_array_value((_,index_fnA,_,_), (_,index_fnB,_,_)) =
    is_same_location(index_fnA(0), index_fnB(0))
```
This code is used on page 21.

Allocating an array involves allocating a location for each element, an additional location for the array value itself, and storing the array value there. This uses the auxillary function, `private_array_alloc`, to allocate space for the elements. The environment is used only to pass to the `sizeof` function. The size of an array is simply the number of elements times the size of each element.

⟨*Array-Value alloc function* 21⟩≡
```
(ArrayOfType(size,elemtype), mem, env) =>
    let val (arr_fn, mem') =
        private_array_alloc(size, 0, fn(_) => raise Array_Struct.array_range,
                            elemtype, mem, env)
    in
        let val (location, mem'') =
            allocmem(mem', BV(IntVal(99)), not_initialized)
                                (* any value will do *)
        in
            let val mem''' =
                writemem(mem'', location,
                         AV(size, arr_fn, elemtype,
```

```
                                size * Data_Type_Struct.sizeof(elemtype, env)))
            in
                (location, mem''')
            end
        end
    end
```

This code is used on page 16.

The function `private_array_alloc` is essentially a `for` loop to allocate successive array locations and collect them in the array indexing function. It is mutually recursive with the other alloc functions. The environment is passed through unchanged.

This function iterates over the array elements. On each iteration it calls `alloc` and builds up the array indexing function with its result, then recurses.

⟨*special array alloc stuff* 22⟩≡

```
private_array_alloc :
        (int * int * (Nat -> Location) * Data_Type * Memory * Environment) ->
                                                ((Nat -> Location) * Memory) =
    fn (wanted_size, old_size, old_fn, elemtype, mem, env) =>
    if wanted_size=old_size
        then (old_fn,mem)
    else
        let val (new_loc,new_mem) = alloc(elemtype,mem,env) in
            let val new_fn = fn(index) =>
                if index=old_size
                    then new_loc
                else old_fn(index)
            in
                private_array_alloc(wanted_size, old_size+1,
                                    new_fn, elemtype, new_mem, env)
            end
        end
```

This code is used on page 16.

Finding the size of an array value involves simply retrieving it from the array value.

⟨*Array-Value sizeof function* 22⟩≡

```
(AV(_, _, _, sizeof)) =>
    sizeof
```

This code is used on page 16.

Copying an array involves using a `for` loop to run through all the elements and copy each one individually. The state propagated by the `for` function

includes the memory and the two indexing functions. The update function given to `for` simply forms a tuple containing the mutated memory and the unchanged indexing functions.

⟨*Array-Value copy function* 23⟩≡
```
(AV(n1, src_f, t, charsize), loc, mem) =>
let val AV(n2, dest_f, t2, charsize2) = readmem(mem, loc) in
    let val (mem, _, _) =
        for(0, n1, (mem, src_f, dest_f),
            fn(i, (mem, src_f, dest_f)) =>
              (copy(readmem(mem, src_f(i)), dest_f(i), mem),
               src_f,
               dest_f))
    in
        mem
    end
end
```
This code is used on page 16.

The functions above use the "for" function to loop through a series of values like the traditional `for` loop. The implementation of the `for` function is given here. Noweb will paste this into the "miscellaneous" section of the program.

The `for` function operates much like a C `for` loop. The index runs from `current` to `range`. On each iteration, the `info` is updated by the `update_fun`.

In C++, this would go something like this:
```
for ( ; current<range; current++)
    info = update_fun(current, info);
```

⟨*the for function declaration* 23⟩≡
```
fun for(current, range, info, update_fun) =
    if current = range then info
    else for(current+1, range, update_fun(current, info), update_fun)
```
This code is used on page 28.

4.7 Pointer Value Implementation

The goal of this section is to complete the pointer structure and implement the alloc, copy, and sizeof functions for pointer values.

⟨*pointer structure declaration* 23⟩≡
```
structure Pointer_Struct : Pointer_Sig =
    struct
        type Pointer_Value = Pointer_Value
        type Memory = Memory
        type Location = Location
```

```
        type Data_Type = Data_Type
        exception ptrdiff_undefined
        exception pointer_escaped_array
        exception deref_outside_array
        exception increment_undefined

        ⟨create function declaration 24⟩
        ⟨make-ref function declaration 25⟩
        ⟨deref function declaration 25⟩
        ⟨difference function declaration 26⟩
        ⟨increment function declaration 26⟩
    end;
```

This code is used on page 12.

A pointer value is basically a location. We also keep track of the type of value the pointer points at. Finally, if the pointer points into an array, we need to know which element it points at. (Recall that C++ arrays are all one dimensional. To use this for "multi-dimensional" arrays, the client must keep track of the dimensions separately.)

C++ does not make a distinction between pointers that can point to cells and pointers that can point to array elements. At any given time, a C++ pointer may point to either. However, we have to know so we can do pointer arithmetic and array bounds checking.

⟨*Pointer-Value type* 24⟩≡

```
(* datatype *) Pointer_Value =
    cell_ptr of Data_Type * Location
  | arr_ptr of Data_Type * Array_Value * Nat
```

This code is used on page 12.

To allocate memory space for a pointer, we need its type. Then we just store the pointer in a new memory cell.

⟨*Pointer-Value alloc function* 24⟩≡

```
(PtrToType(t), mem, env) =>
    allocmem(mem, PV(Pointer_Struct.create(t)), initialized)
```

This code is used on page 16.

To create a new pointer, we need only its type. We use the undefined location because the pointer is not yet pointing anywhere.

⟨*create function declaration* 24⟩≡

```
fun create(t) = cell_ptr(t, undef_loc)
```

This code is used on page 24.

To copy a pointer value, we assume the pointer types are of compatable types, then we just write the value into the memory cell. The client is responsible for type checking.

⟨*Pointer-Value copy function* 25⟩≡
```
(PV(v), newloc, mem) =>
    writemem(mem, newloc, PV(v))
```
This code is used on page 16.

The size of all pointers in this implementation is constant, although the standard allows for pointers of various sizes.

⟨*Pointer-Value sizeof function* 25⟩≡
```
(PV(_)) => 4
```
This code is used on page 16.

When making a pointer refer to some cell, we require the client to ensure the types are compatable. If they want to make the pointer point to an array, this forms an `arr_ptr` to the first element. Otherwise, it forms a `cell_ptr`.

We want it to do this regardless of whether it used to be pointing to an array or not. So once again literate programming gives us a way to name just the body of a function and to use that body in two places.

⟨*make-ref function declaration* 25⟩≡
```
fun make_ref(cell_ptr(t,_), newloc, mem) : Pointer_Value =
    ⟨make-ref body 25⟩
  | make_ref(arr_ptr(t,_,_), newloc, mem) =
    ⟨make-ref body 25⟩
```
This code is used on page 24.

⟨*make-ref body* 25⟩≡
```
    let val referenced_value = readmem(mem, newloc) in
        (case referenced_value of
             AV(array_value) => arr_ptr(t, array_value, 0)
           | _ => cell_ptr(t, newloc))
    end
```
This code is used on page 25.

Dereferencing a pointer involves pulling out its location. For pointers to arrays, this means indexing the array.

⟨*deref function declaration* 25⟩≡
```
fun deref(cell_ptr(_,loc)) = loc
  | deref(arr_ptr(_,arrval,indx)) = (Array_Struct.index(arrval,indx)
    handle array_range => raise deref_outside_array)
```
This code is used on page 24.

Pointer subtraction is done by cases. We return 0 if the pointers refer to the same cell. If the pointers point into the same array, we perform subtraction. Otherwise, the `ptrdiff_undefined` exception is raised. In all cases the result is the C++ type `ptrdiff_t`, represented by an ML integer. [ES90, Section 5.7].

⟨*difference function declaration* 26⟩≡
```
fun difference(cell_ptr(_,loc1), cell_ptr(_,loc2)) =
    if is_same_location(loc1,loc2) then
        0
    else
        raise ptrdiff_undefined
  | difference(arr_ptr(_,arrval1,index1), arr_ptr(_,arrval2,index2)) =
    if Array_Struct.is_same_array_value(arrval1, arrval2) then
        index1 - index2
    else
        raise ptrdiff_undefined
  | difference(_, _) =
        raise ptrdiff_undefined
```
This code is used on page 24.

Pointer arithmetic makes sense only when applied to a pointer into an array. For a pointer into an array, simply perform the increment and check if the bounds of the array were violated. If they were, or if the pointer was not pointing into an array, the exception `increment_undefined` is raised.

Note that this section has to be intimately familiar with how array values are represented so it can know if the array bounds are violated.

⟨*increment function declaration* 26⟩≡
```
(* fun increment: Pointer_Value * int -> Pointer_Value *)
fun increment(arr_ptr(t,(array_size,indx,typ,sizeof), old_index), increment) =
    let val new_index = old_index + increment in
        if new_index >=0 andalso new_index <= array_size
           (* Allow pointer to go one past the end *)
        then
            arr_ptr(t, (array_size,indx,typ,sizeof), new_index)
        else
            raise increment_undefined
    end
  | increment(_, _) =
    raise increment_undefined
```
This code is used on page 24.

4.8 Object Value Implementation

The objective of this section is to finish the object structure and to implement the alloc, copy, and sizeof functions for object values.

⟨*object structure declaration* 27⟩≡
```
structure Object_Struct : Object_Sig =
    struct
        type Object_Value = Object_Value
        type Memory = Memory
        type Location = Location
        exception not_member
        fun ⟨dot fun implementation 27⟩
    end;
```
This code is used on page 12.

An object value contains a mapping from identifier to location. This mapping is used in the C++ dot (`.`) and arrow (`->`) operators, and is referred to here as the "dot" operator.

⟨*dot function signature* 27⟩≡
```
Identifier -> Location
```
This code is used on page 31.

An object value is constructed when an object is created; it is never modified, only the cells it refers to. A "dot" function is created fresh for each new object. The copy and sizeof functions are the same for all objects of the same class. These functions are implemented by the *Class* functions, described in a later section.

⟨*Object-Value type* 27⟩≡
```
Object_Value = dot_type * object_copier_type * int
```
This code is used on page 12.

To implement the `dot` operator, we have only to invoke the "dot" function which maps from identifier to location.

⟨*dot fun implementation* 27⟩≡
```
dot((dot_fn, _, _), id) = dot_fn(id)
```
This code is used on page 27.

To allocate an object, we need information from its class. Essentially, each class instantiates objects. This function was constructed when the class was declared. We have only to retrieve that function and invoke it.

⟨*Object-Value alloc function* 28⟩≡
```
(ClassOfType(name), mem, env) =>
    let val Class(size, alloc_fun) = lookup(env, name) in
        alloc_fun(mem)
    end
```
This code is used on page 16.

The function to copy the object is constructed when the object is created. We have only to invoke the result on the current memory and say where we want it copied to.

⟨*Object-Value copy function* 28⟩≡
```
(OV(dot_fn, copy_fn, size), mem, loc) =>
    copy_fn(loc, mem)
```
This code is used on page 16.

The size of an object is computed when its class is created and was stored in the object value for easy access. It is used here.

⟨*Object-Value sizeof function* 28⟩≡
```
(OV(_, _, size)) => size
```
This code is used on page 16.

4.9 Miscellaneous details

And now we have only miscellaneous details to attend to. These include some shorthand types and some general-purpose functions.

⟨*misc definitions* 28⟩≡
```
type Identifier = string
type Nat = int
⟨general-purpose function update declaration 15⟩
⟨the for function declaration 23⟩
```
This code is used on page 4.

5 Valuation Functions

The only valuation function, `EvalDeclList`, accepts a `Declaration list` and produces a declaration continuation transformer. The function is written in a continuation passing style (CPS) [Sch86, Section 9.1]. Its signature is given here.

⟨*EvalDeclList signature* 28⟩≡
```
Declaration list -> DeclCont -> DeclCont
```
This code is used on page 29.

Our continuations are functions that take an `(Environment * Memory)` and produce a new `(Environment * Memory)`. This continuation promises to produce a new `(Environment * Memory)` according to the declaration list used to create it and the `(Environment * Memory)` it was handed.

⟨*DeclCont type definition* 29⟩≡
```
type DeclCont = Environment * Memory -> Environment * Memory
```
This code is used on page 29.

The remainder of this section gives the order chunks of code are pasted together. This is different than the order presented in this paper.

⟨*Valuation function implementations* 29⟩≡
⟨*DeclCont type definition* 29⟩
⟨*Class declaration stuff* 31⟩
⟨*fun EvalDecl definition* 29⟩
⟨*fun EvalDeclList definition* 29⟩
This code is used on page 4.

5.1 Valuation function definitions

To evaluate a declaration list, we use straightforward CPS recursion. At each element we use `EvalDecl` to produce a new continuation transformer, then apply this to the continuation which evaluates the rest of the list.

⟨*fun EvalDeclList definition* 29⟩≡
```
val rec EvalDeclList : ⟨EvalDeclList signature 28⟩ =
    fn (decl :: decllist) => (fn (declk) =>
      EvalDecl(decl)
        (EvalDeclList(decllist)
          declk))
     | (nil) => (fn (declk) =>
       declk)
```
This code is used on page 29.

Function `EvalDecl` evaluates a single declaration and produces a declaration continuation transformer. Its design is driven by the `datatype Declaration` given in the section on C++ declarations. There are two types of declarations: variables and classes. The functions which process each of these declarations are given in their respective sections.

⟨*fun EvalDecl definition* 29⟩≡
```
fun ⟨eval variable-declaration function 30⟩
  | ⟨eval class-declaration function 30⟩
```
This code is used on page 29.

5.1.1 Variable Declarations

Here we say how to evaluate a variable declaration. The process involves first making sure the variable was not previously declared, allocating memory for it, and recording the result into the environment. Note that this uses the SML shorthand notation for curried functions.

⟨*eval variable-declaration function* 30⟩≡
```
EvalDecl(var_decl(id, d_type)) (declk) =
    (fn (env, mem) =>
        if is_in_env(env, id)
        then (output (std_out, "Identifier "^ id ^" redeclared\n");
              declk(env, mem))
        else
            let val (loc, mem') = alloc(d_type, mem, env) in
                let val env' = add(env, id, Variable(loc)) in
                    declk(env', mem')
                end
            end)
```
This code is used on page 29.

5.1.2 Class Declarations

For a class definition, we process the fields and store that information in the environment. Memory is unchanged.

⟨*eval class-declaration function* 30⟩≡
```
EvalDecl(class_decl(class_name, fields)) (declk) =
    fn (env, mem) =>
        declk(add(env, class_name, make_class_info(fields, env)), mem)
```
This code is used on page 29.

6 Class Declarations

The implementations of classes and objects are intimately related. The critical relationship is that when a class is declared, we create a function to create objects of that class. When this function is invoked, it allocates memory for the object's members and creates the dot function which maps from member names to the newly allocated locations.

To get started, we will review the types we are dealing with. The information stored for a class has type `class_info` which was defined as part of the denotable values. The information stored for an object has type `Object_Value` which was defined as part of the storable values.

Two types were not defined elsewhere. These are the type of the dot function, and the type of the function to copy an object.

⟨*auxilliary object types* 31⟩≡
```
      dot_type = ⟨dot function signature 27⟩
  and object_copier_type = Memory * Location -> Memory
```
This code is used on page 12.

6.1 Creating class info

We now turn to the function `make_class_info` which transforms a list of fields into a denotable value to represent the class. Its type is given here.

⟨*make-class-info function signature* 31⟩≡
```
  Declaration list * Environment -> Denotable_Value
```
This code is used on page 31.

The `make_class_info` function is fairly involved and needs some auxilliary definitions. These include the general-purpose field iterator, a function to transform a list of declarations into a list of `(Identifier * Type)` pairs, and the initial dot function. These are discussed later in this section.

⟨*Class declaration stuff* 31⟩≡
```
  ⟨create-field-iterator function definition 33⟩
  ⟨create-field-list function definition 32⟩
  ⟨initial dot function definition 35⟩
  ⟨make-class-info function definition 31⟩
```
This code is used on page 29.

We now discuss how the `make_class_info` function works. It first creates the list of fields from the list of members, then creates a general-purpose function to iterate over the fields and collect information about them. Next it uses this "field iterator" function to find out the size of the class and objects of that class.

Then it packages its work with the `Class` constructor which takes the pre-calculated size and the not-yet-calculated instantiation function and produces a `Denotable_Value` which is returned as the value of the function.

The instantiation function takes a memory. When called, this instantiates the fields of an object, creates a function to copy the object, and wraps these up with the size of the object to create the object information.

The details are discussed in the following subsections.

⟨*make-class-info function definition* 31⟩≡
```
  val make_class_info : ⟨make-class-info function signature 31⟩ =
    fn (members, env) =>
      let val fields = create_field_list(members)
      in
          let val field_iter =
              create_field_iterator(fields)
```

```
        in
            let val object_size =
                ⟨object size guts 33⟩
            in
                Class(object_size,
                      fn(mem) =>
                      let val (mem, dot_fn) =
                          ⟨object instantiation guts 35⟩
                      in
                          let val copy_fn =
                              ⟨object copy guts 33⟩
                          in
                              ⟨wrap up object 35⟩
                          end
                      end)
            end
        end
    end
```

This code is used on page 31.

6.2 Create field list

To create a list of fields from a list of members, we have only to keep "variable" declarations, and ignore nested class declarations. Note that we strip the `var_decl` constructor and put only the identifier and type into the list. This function performs the usual recursion on lists.

⟨*create-field-list function definition* 32⟩≡

```
fun create_field_list([]) =
    []
  | create_field_list(var_decl(id, var_type) :: rest_of_list) =
    (id, var_type) :: create_field_list(rest_of_list)
  | create_field_list(class_decl(id, decl_list) :: rest_of_list) =
    create_field_list(rest_of_list)
```

This code is used on page 31.

6.3 General purpose field iterator function

This `create_field_iterator` function creates the field iterator using the usual recursion on lists. The function it produces, the iterator function, takes two arguments: a state, and a function to update the state for each field. This update function takes the current state, the current field, and returns a new state. We will see examples of the field iterator function's use in the next sections.

Note that the normal use of this function is to supply only the list of fields, i.e., this really is a curried function. Supplying the first argument, the list of fields, produces the iterator. Supplying the second set of arguments, the state and the update function, uses the iterator to do work.

⟨*create-field-iterator function definition* 33⟩≡
```
fun create_field_iterator ([]) (state,f) =
    state
  | create_field_iterator (first_field :: other_fields) (state,f) =
    let val state' = f(state,first_field) in
        create_field_iterator (other_fields) (state', f)
    end
```
This code is used on page 31.

6.4 Object size

To find the size of an object we have only to sum the sizes of the members. This is a straightforward use of the iterator function. The state we keep is the cumulative size of the fields seen so far. The update function has only to add that to the size of the current field. One small wrinkle is that an object with no members has a size greater than zero [ES90, Section 5.3.2, Section 9]; here we arbitrarily choose 4 as the size of such an object.

⟨*object size guts* 33⟩≡
```
let val size =
    field_iter(0, fn(size_so_far, (field_name, field_type))
                  => size_so_far + Data_Type_Struct.sizeof(field_type, env))
in
    if size = 0 then 4
    else             size
end
```
This code is used on page 32.

6.5 Object copy

We next turn to the problem of copying objects of this type to memory locations occupied by other objects of this same type. To do this, we create a function that copies arbitrary objects, and arrange to call it with only objects of the correct type. This function takes a memory to mutate and a location to copy the object to.

⟨*object copy guts* 33⟩≡
```
fn(mem, dst_loc) =>
  ⟨ocg' 34⟩
```
This code is used on page 32.

We first read the destination object from memory and pull out its dot function. The content of the cell is required to be an object value, otherwise this code will raise the SML `match` exception. It is the responsibility of the user to make sure this functions is used correctly.

⟨*ocg'* 34⟩≡
```
let val OV(dst_dot_fn, _, _) = readmem(mem, dst_loc) in
    ⟨ocg" 34⟩
end
```
This code is used on page 33.

Once we have the dot functions of both the source and destination objects, we can iterate over the fields and perform the copy member-by-member using the iterator function. The state includes the two dot functions and the current memory image. The update function simply performs the copy and passes the dot functions to the next iteration unchanged. Afterward, we discard the source and destination dot functions and return only the memory.

Here we use the values `field_iter` and `dot_fn` from the quadruply nested `let`s in the section where this code goes.

⟨*ocg"* 34⟩≡
```
let val (mem, src, dst) = field_iter
    ((mem, dot_fn, dst_dot_fn),
     fn((mem, src, dst), (field_name, field_type)) =>
         let val value = readmem(mem, src(field_name)) in
             let val mem' = copy(value, dst(field_name), mem) in
                 (mem', src, dst)
             end
         end)
in
    mem
end
```
This code is used on page 34.

6.6 Object allocation

We now turn to the problem of allocating object memory. Allocating the memory is a straightforward use of the field-iter function. The state used by the field iterator includes the memory we are mutating and the dot function we are creating. On each iteration we simply allocate memory for the new field and update the dot function to record the new location. The result of all of this is the mutated memory and the dot function.

Here we use the values `field_iter` and `mem` from the quadruply nested `let`s in the section where this code goes.

⟨*object instantiation guts* 35⟩≡
```
field_iter ((mem, initial_dot_fun),
  fn((mem, dot), (field_name, field_type)) =>
  let val (loc, mem') = alloc(field_type, mem, env) in
      (mem', update(dot, field_name, loc))
  end)
```
This code is used on page 32.

The initial dot function does not recognise any member names.

⟨*initial dot function definition* 35⟩≡
```
exception bad_member
val initial_dot_fun = fn(name) => raise bad_member
```
This code is used on page 31.

6.7 Wrapping up objects

Once we have all the information we need for an object, we wrap it all up and store it in memory. This simply fills out an `Object_Value` and stores that in a primitive memory cell.

⟨*wrap up object* 35⟩≡
```
let val object_value = (dot_fn, copy_fn, object_size)
in
    allocmem(mem, OV(object_value), initialized)
end
```
This code is used on page 32.

7 Testing

What follows is not part of the semantics itself. This section describes tools to examine the environment and memory for testing purposes. These consists primarily of a "show-cell" function to describe the contents of memory and a "show-id" function to describe the meaning of identifiers.

⟨*memory examiner* 35⟩≡
⟨*fun format-Data-Type definition* 36⟩
⟨*Storable-Value printer function definition* 36⟩
⟨*fun show-cell definition* 36⟩
⟨*show-id function definition* 37⟩
This code is used on page 4.

The primary interface is a function to display the content of one memory cell.

⟨*fun show-cell definition* 36⟩≡
```
fun show_cell(loc, mem) =
    let val item = readmem(mem, loc)
    in
        (print_storable_value(item);
         print "\n  ")
    end
  handle Memory_Struct.read_out_of_bounds =>
           (print "(read out of bounds)\n  "; ())
       | Memory_Struct.uninitialized_memory =>
           (print "(cell does not yet contain a value)\n  "; ())
```
This code is used on page 35.

The next function formats and prints a `Storable-Value`; it returns unity. The design of this function is driven by `datatype Storable-Value`.

⟨*Storable-Value printer function definition* 36⟩≡
```
fun print_storable_value(BV(IntVal(x))) =
    (print ("integer "^ makestring(x)); ())
  | print_storable_value(BV(CharVal(x))) =
    (print ("character "^ makestring(x)); ())
  | print_storable_value(BV(FloatVal(x))) =
    (print ("float "^ makestring(x)); ())
  | print_storable_value(BV(DoubleVal(x))) =
    (print ("double "^ makestring(x)); ())

  | print_storable_value(AV(size:int,index_fn,elem_type,charsize)) =
    (print ("array["^makestring(size)^"] of " ^ format_Data_Type(elem_type));
     ())
  | print_storable_value(PV(_)) =
    (print "pointer (not specified)"; ())
  | print_storable_value(OV(_)) =
    (print "object (not specified)"; ())
```
This code is used on page 35.

Finally, `format_Data_Type` returns a string representing the `Data_Type`.

⟨*fun format-Data-Type definition* 36⟩≡
```
fun format_Data_Type(CharType) = "char"
  | format_Data_Type(IntType) = "int"
  | format_Data_Type(FloatType) = "float"
  | format_Data_Type(DoubleType) = "double"
  | format_Data_Type(PtrToType(t)) =
        "pointer to " ^ format_Data_Type(t)
  | format_Data_Type(ArrayOfType(n,t)) =
```

```
        "array[" ^ makestring(n) ^ "] of " ^ format_Data_Type(t)
  | format_Data_Type(ClassOfType(id)) =
        "class " ^ id
```

This code is used on page 35.

We next describe a tool for getting information from the environment. Given a C++ symbol it prints everything it can find out about it.

⟨*show-id2 function definition* 37⟩≡

```
fun show_id2(Variable(loc), env, mem) =
    (print ("a variable in cell "^makestring(loc)^" of type ");
     show_cell(loc, mem);
      ())
  | show_id2(Class(_,_), env, mem) =
    (print "the name of a class";
     ())
```

This code is used on page 37.

Finally the `show-id` function looks up the `id` and passes its value to `show-id2` for display.

⟨*show-id function definition* 37⟩≡

```
⟨show-id2 function definition 37⟩
fun show_id(id, env, mem) =
    let val thing = lookup(env, id) in
        (print (id^" is ");
         show_id2(thing, env, mem))
     end
  handle symbol_not_found => (print ("Unknown identifier: "^id^"\n  "); ())
```

This code is used on page 35.

8 Related Work

Sethi published an article on the denotational semantics of C programs [Set80]. There is a paper on denotational semantics implemented in SML [Mas91]. Compared to these, the techniques used in this paper offers a greater explanation of the denotational semantics technique and a better partitioning of the semantic values into domains. There is work in separating the concerns of the compile-time system from the run-time system [LP87]. This makes compilers derived from a semantic specification much more efficient. The clear separation between the domains in this paper makes it amenable to this type of optimization.

9 Conclusions

We wrap up the paper with a discussion of what was learned and a discussion of possible extensions to this work.

9.1 What was learned

Using literate programming to implement denotational semantics in SML is not commonly done. Here we point out some of the problems and benefits of the combination of techniques.

As programmers, we solve problems and code solutions. The code is there for all to see, but the solutions are usually lost and must be re-created from the code. Literate programming encourages us to document what we were thinking when we solved our coding problem. This, of course, requires extra effort. But this extra effort is rewarded when we have to look back to the code to understand or modify it. It really does.

Another feature of literate programming is that it extends the target language by allowing us to name pieces of our program smaller than the language allows. For instance, this paper defines functions piecemeal (as is normal in denotational semantics), but SML does not allow this. Literate programming gives us the flexibility to do this by naming pieces of the program smaller than the language normally allows. Another example of where literate programming helps is in naming just the body of a function for use in two places. Of course, this can be done with an auxilliary function, but literate programming allows us to both modularize our code and arrange the pieces for efficient execution.

ML is well suited to implementing denotational semantics. ML's first-class functions provide the expressive power necessary to directly model semantic specifications. ML's module capability (signatures and structures) provides a convenient implementation of semantic domains. ML's pattern matching and syntactic support for curried functions makes the implementation look more like a traditional semantic specification.

Despite its advantages, ML is not ideal for implementing denotational semantics. Its typechecking is a blessing and a pain. On the one hand, it catches most errors before programs can be used. On the other hand, it limits the flexibility of functions written in continuation-passing style (CPS). In short, we still have the control advantages of CPS but must use a SML `datatype` (and explicit injection into a disjoint union) to return an error condition. This is commonly done in a direct semantics.

Finally, ML does not have syntactic sugar for defining a function and giving its complete type. For example, we sometimes give the signature for a function when discussing what it does. Then we give the function itself later and would like to include its signature in th source code to prove that we were not lying about it. The problem is that SML does not allow the complete signature of a function to be given by the `fun` syntax—only its return type can be specified

easily. The primitive `val rec f : TYPE = fn (args) => body` is currently used to document the type.

9.2 Future directions

This work could be extended in several directions. The first and most obvious extension is to remove the restrictions from the declarations it can deal with by adding the usual C++ features. These features include typedefs, private and protected members, nested classes, and inheritance.

Functions and expressions could be added, along with all the other features of C++. A parser could be added to ease testing. Finally, the semantics could be separated more cleanly into compile-time and run-time responsibilities. This would help language implementors and those concerned about execution efficiency.

10 Acknowledgements

I gratefully acknowledge the help of my major professor, Gary Leavens, who initially suggested the project, provided innumerable references, papers, encouragement, and insights. I would also like to thank my program-of-study committee for asking the right questions which helped to sharpen my thinking about this project.

References

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Mass., 1990.

[GMW79] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, N.Y., 1979. The second author is listed on the cover as Arthur J. Milner, which is clearly a mistake.

[LP87] Peter Lee and Uwe Pleban. A realistic compiler generator based on high-level semantics: Another progress report. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich*, pages 284–295. ACM, January 1987.

[Mas91] Dave Mason. Denotational semantics and an ML interpreter for a functional programming language. obtained from the author: dmason@plg.uwaterloo.ca. 1991.

[Pau91] Laurence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, NY, 1991.

[Ram92] Norman Ramsey. Literate-programming tools need not be complex. Technical report, Princeton, August 1992.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., Boston, Mass., 1986.

[Set80] Ravi Sethi. A case study in specifying the semantics of a programming langauge. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada*, pages 117–130. ACM, January 1980.

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Tech Report: TR 93-15
Submission Date: May 12, 1993