ACM SIGPLAN Language Tips



# C++ Toolbox

**Editor:** *G. Bowden Wise, Dept. of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180; wiseb@cs.rpi.edu*

## A Framework for Programming Denotational Semantics in C++

***Nikolaos S. Papaspyrou***
(nickie@softlab.ntua.gr)

National Technical University of Athens
Division of Computer Science, Software Engineering Laboratory
Polytechnioupoli, 15780 Zografou, Athens, Greece

**Abstract.** In this paper, we describe how the denotational semantics of programming languages can be implemented in C++, by exploiting the object-oriented programming paradigm. Such implementations are execution models, extremely useful for the study of programming languages. Although C++ is not a natural choice for this problem domain, compared to functional programming languages such as ML, we suggest a type-safe framework, implemented in pure C++, that integrates functional characteristics such as high-order functions and is capable of naturally expressing denotational descriptions. Finally, by comparing our approach to possible implementations using functional languages, we investigate extensions to C++ that would be valuable in this problem domain.

## Introduction

Denotational semantics, or the mathematical approach to programming language semantics, is a formalism that was introduced by D. S. Scott and C. Stratchey in the late '60s. Since then, it has been widely studied by distinguished researchers and has been used as a means for the study of programming languages. Introductions to the denotational approach, including useful bibliography, can be found in [19] and [11]. For a more in-depth presentation of the underlying theory and the techniques that have been developed, the reader is referred to [18], [6] and [15].

According to the denotational approach, semantics is described by attributing mathematical *denotations* to programs and program segments. Denotations are typically high-order functions over Scott domains, computed for each program segment by appropriately combining the denotations of its subsegments. A denotational description of a programming language defines directly an execution model, i.e. an interpreter, and can be regarded as a complete and unambiguous reference standard, of obviously great value to users and language implementers. It is not always easy to implement such an interpreter by translating the denotational description to a program, written in a general purpose programming language. Using an appropriate target language is very important and can greatly reduce the complexity of this task. Several languages have been suggested and used for this purpose, with more or less success. It seems that functional programming languages are more suitable, as suggested in [20] where ML is used as a meta-language for denotational descriptions. Imperative programming languages have also been suggested, such as ALGOL 68 [12] and Pascal [1], with considerably less success. In the first case, an extension to the language was suggested by the author and in the second it was clear that similar extensions would be extremely valuable.

In this paper, we attempt to apply the object-oriented programming paradigm to the problem of implementing denotational descriptions and we suggest C++ as the implementation language. It is clear that C++ is not a natural choice for this problem domain, lacking lexical closures, expressible functions and fully operational high-order functions. However, we believe that object-orientation is a useful tool in this problem domain and attempt to overcome the many difficulties imposed by C++.

A framework is suggested in this paper that provides a type-safe implementation for the $\lambda$-calculus over Scott domains and can be directly used to implement denotational descriptions. This framework differs from previous approaches in the use of the object-oriented programming

paradigm. Our approach results in denotational descriptions much closer to the original meta-language and therefore more natural than approaches using imperative languages, mentioned above. However, our implementation is not as elegant as one in ML [20]. It is clear that the presence of features characteristic of functional programming languages, such as high-order functions, currying, partial binding and the $\lambda$-notation, determines the suitability of a programming language for implementing denotational descriptions. In order to overcome the drawbacks of C++, we attempt to integrate such features in our framework. But, in contrast to other approaches towards the same goal [14, 4, 9, 10, 13], our approach does not require any extensions to the language. Unfortunately, there is a tradeoff between natural description using pure C++ and performance. In our approach, we resolve this dilemma at the expense of performance, on the grounds that implementations of denotational descriptions are commonly used for the study of programming languages, in a context where performance is of little importance.

## Definition of the problem

In order to formulate a denotational description, a meta-language has to be employed. The meta-language that will be used in this paper is a variation of the $\lambda$-calculus over Scott domains [16], which we will very briefly present in this section. A very good introduction to domain theory can be found in [8]. For a complete definition and applications of domain theory in programming language semantics, the reader is referred to [7].

Domains are mathematical spaces whose elements are partially ordered by a relation of definedness, commonly denoted as $\sqsubseteq$. For each domain $d$ there is a least or *bottom* element, denoted by $\perp_d$, representing an undefined value. Domains that are constructed from normal sets of elements with the addition of a bottom element are called *flat* domains and are denoted by $\{e_1, e_2, \ldots\}^\circ$. The flat domain of integer values is denoted by $\mathbf{N}$, whereas the flat domain $\{true, false\}^\circ$ is denoted by $\mathbf{T}$. For $d$, $d_1$ and $d_2$ domains, other compound domains can be defined as shown in Figure 1. Domain definitions can be recursive, e.g. $\mathbf{Tree} = \mathbf{N} + (\mathbf{Tree} \times \mathbf{Tree})$, but it should be noted that not all equations over domains have non-trivial solutions.

Element expressions are formed as shown in Figure 1, where $x$ denotes a variable whose value belongs in a specific domain and $e$, $e_1$ and $e_2$ are element expressions. Further notational conventions include the omission of subscripts, the omission of variable types in the $\lambda$-notation, the representation of functions as prefix unary and infix binary operators, as well as the omission of **inl** and **inr** whenever they can be deduced from the context. Also, since the distinction between Cartesian and smash products, as well as separated and coalesced sums, is usually clear from the context, it is possible to use the same notation for both. As a last notational convention, $(e \,\mathtt{E}\, d)$ may be used for determining whether the element of a sum domain denoted by $e$ corresponds to an element of $d$, whereas $(e \mid d)$ denotes the corresponding element of $d$. These two can be expressed by using operators **outl**, **outr**, **isl** and **isr** appropriately.

To illustrate the use of the meta-language, we will present a denotational description for a *Pure Functional Language with Control*, which for brevity we will call PFLC. The language features high-order functions, recursion by means of a `fix` operator, advanced control constructs, such as `abort`, `call/cc` and a control delimiter, or *prompt*, written as #. All control features are described in detail in [17]. The complexity of our example serves the purpose of using as many elements of the meta-language as possible, in its semantic description. Figures 2 and 3 present the abstract syntax of PFLC, the syntactic and semantic domains and a subset of the semantic equations, using continuation semantics.

Implementation of a denotational description in C++ requires a means of translating abstract syntax, domain definitions and semantic equations into C++ code.[1] This is the aim of the framework that we suggest in the following sections.

## Untyped version

We will first consider an implementation for an untyped version of our meta-language. In this implementation, we will use the *envelope/letter* idiom [3], in order to achieve method polymorphism and at the same time alleviate the memory-management problems that result from the use of object pointers. An *envelope* class will be used for the representation of Scott domain elements, whereas several *letter* classes will be used for the representation of various operations on such elements. The envelope class `Element` is defined as:

```
class Element
{
private:
  ElementImpl * const impl;

public:
  Element (ElementImpl * const ei): impl(ei) { }
```

[1] A complete interpreter would also require a parser and a translator from concrete to abstract syntax.

Figure 1: An overview of λ-calculus over Scott domains.

| | |
|---|---|
| **Domain notation.** | |
| $d_1 \to d_2$ | The domain of *functions* from $d_1$ to $d_2$. |
| $d_1 \to_s d_2$ | The domain of *strict* functions from $d_1$ to $d_2$. A function $f \in d_1 \to d_2$ is strict iff $f(\bot_{d_1}) = \bot_{d_2}$. |
| $d_\bot$ | The *lifted* domain that is produced by adding a new bottom element under a distinct copy of $d$. |
| $d_1 \times d_2$ | The *Cartesian product* of $d_1$ and $d_2$. |
| $d_1 \otimes d_2$ | The *smash product* of $d_1$ and $d_2$, obtained from the Cartesian product by identifying all pairs containing bottom elements. |
| $d_1 + d_2$ | The *separated sum* of $d_1$ and $d_2$, containing distinguished copies of all elements of the two domains. |
| $d_1 \oplus d_2$ | The *coalesced sum* of $d_1$ and $d_2$, obtained from the separated sum by identifying all bottom elements. |
| **Element expressions.** | |
| $e_1 =_d e_2$ | Denotes *true* if $e_1$ and $e_2$ denote the same element of the flat domain $d$, $\bot_T$ if any of them denotes $\bot_d$, *false* otherwise. |
| $e \to e_1, e_2$ | This construct requires $e$ to denote an element of $\mathbf{T}$; $e_1$ and $e_2$ to denote elements $x_1, x_2 \in d$ respectively. Then, it denotes $x_1$ if $e$ denotes *true*, $x_2$ if $e$ denotes *false*, and $\bot_d$ if $e$ denotes $\bot_\mathbf{T}$. |
| $\lambda x : d_1 . e$ | The function $f \in d_1 \to d_2$ given by $f(x) = e$, where $x \in d_1$. |
| $e_1\ e_2$ | The result of applying function $f \in d_1 \to d_2$ denoted by $e_1$ to the value $x \in d_1$ denoted by $e_2$. |
| $e_1 \circ e_2$ | The composition of functions $f_1$ and $f_2$ denoted by $e_1$ and $e_2$ respectively, i.e. the function $f(x) = f_1(f_2(x))$. |
| $\mathbf{fix}_d$ | The least fixed point operator, mapping a function $f \in d \to d$ to the least element $x \in d$, w.r.t. $\sqsubseteq$, satisfying $f(x) = x$. |
| $\mathbf{strict}_d$ | The function mapping each function $f \in d = d_1 \to d_2$ to the corresponding strict function $f' \in d_1 \to_s d_2$. |
| $\mathbf{up}_d$ | The function mapping each element of $d$ to the corresponding element of $d_\bot$. |
| $\mathbf{down}_d$ | The function mapping each element of $d_\bot$ back to the corresponding element of $d$, and $\bot_{d_\bot}$ to $\bot_d$. |
| $\langle e_1, e_2 \rangle$ | The pair belonging in $d_1 \times d_2$ with components $x \in d_1$ and $y \in d_2$ denoted by $e_1$ and $e_2$ respectively. |
| $\mathbf{fst}_d, \mathbf{snd}_d$ | Functions mapping a pair $\langle x, y \rangle \in d = d_1 \times d_2$ to $x \in d_1$ and $y \in d_2$ respectively. |
| $\mathbf{inl}_d, \mathbf{inr}_d$ | Functions mapping elements of $d_1$ and $d_2$ respectively to the corresponding element of $d = d_1 + d_2$. |
| $\mathbf{outl}_d, \mathbf{outr}_d$ | Functions mapping an element $x \in d = d_1 + d_2$ back to the corresponding element of $d_1$ or $d_2$. |
| $\mathbf{isl}_d, \mathbf{isr}_d$ | Functions mapping an element $x \in d = d_1 + d_2$ to *true* or *false*, depending on whether they belong to $d_1$ or $d_2$. |

```
  Element (const Element & e):
    impl(e.impl->copy()) { }
 ~Element () { delete impl; }
  Element operator () (const Element & arg) const
    { return Element(new ApplicationImpl(
        impl, arg.impl)); }
  friend Element lambda (const Element & exp)
    { return Element(new AbstractionImpl(
        exp.impl)); }
  friend Element arg (int db)
    { return Element(new ParameterImpl(db)); }
  friend Element fix (const Element & exp)
    { return Element(new FixImpl(exp.impl)); }
  ...
  friend Element evaluate (const Element & e)
    { return Element(e.impl->evaluate()); }
};
```

where functions `lambda` and `arg` are used for the creation of λ-abstraction elements, `operator ()` is used for the creation of function applications and function `fix` is used for the implementation of the least fixed point operator. In addition, method `evaluate` is used for the evaluation of elements, by applying the evaluation rules of λ-calculus over Scott domains.

A set of *letter* classes is defined for element implementations, derived from the abstract letter class `ElementImpl`. In order to differentiate between implementation classes in run-time, we need a safe dynamic type casting mechanism. Although *run-time type inference* (RTTI) has long been suggested as part of the draft C++ standard, it is not generally supported by compilers, at least in a portable manner. For this reason, we resort in defining a virtual `whatIs` method and a static `isMember` method for all concrete classes. The definition of these two is facilitated by using two special macros: `RTTI_ABSTRACT` and `RTTI_SIGNATURE`. Class `ElementImpl` is defined as:

```
class ElementImpl
{
  RTTI_ABSTRACT

public:
            ElementImpl () { }
  virtual  ~ElementImpl () { }
  virtual   ElementImpl * copy () const = 0;
  virtual   ElementImpl * subst (int db,
     const  ElementImpl * val) const = 0;
  virtual   ElementImpl * incFV (int t = 0) const = 0;
  virtual   ElementImpl * evaluate () const = 0;
};
```

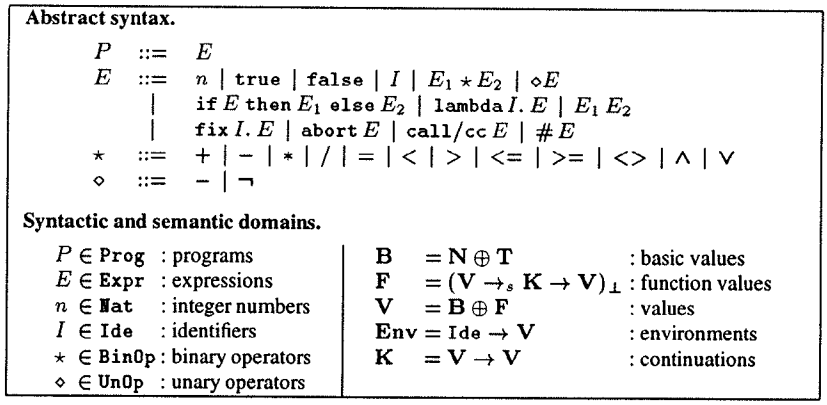
where method `copy` implements the duplication of an object, method `subst` is used for textual substitution, method `incFV` will be explained later and method `evaluate` is used for the evaluation of elements.

Concrete letter classes can be defined for the implementation of domain operations, such as λ-abstractions, function applications or the **fix** operator. A letter class must be defined for bottom elements. Furthermore, a letter class must be defined for the implementation of function parameters. For this purpose we will use *de Bruijn indices* [5] instead of named dummies.[2] De Bruijn indices facilitate the definition and implementation of textual substitution. Of the forementioned letter classes, `AbstractionImpl` can be defined as:

```
class AbstractionImpl: public ElementImpl
```

[2] We will write de Bruijn indices as $\#n$, where $n \geq 1$.

Figure 2: Denotational description of PFLC (abstract syntax and domains).



**Abstract syntax.**

$$
\begin{array}{lcl}
P & ::= & E \\
E & ::= & n \mid \texttt{true} \mid \texttt{false} \mid I \mid E_1 \star E_2 \mid \diamond E \\
 & \mid & \texttt{if}\, E\, \texttt{then}\, E_1\, \texttt{else}\, E_2 \mid \texttt{lambda}\, I.\, E \mid E_1\, E_2 \\
 & \mid & \texttt{fix}\, I.\, E \mid \texttt{abort}\, E \mid \texttt{call/cc}\, E \mid \#E \\
\star & ::= & + \mid - \mid * \mid / \mid = \mid < \mid > \mid <= \mid >= \mid <> \mid \wedge \mid \vee \\
\diamond & ::= & - \mid \neg
\end{array}
$$

**Syntactic and semantic domains.**

| | | | |
|---|---|---|---|
| $P \in \texttt{Prog}$ | : programs | $\mathbf{B} = \mathbf{N} \oplus \mathbf{T}$ | : basic values |
| $E \in \texttt{Expr}$ | : expressions | $\mathbf{F} = (\mathbf{V} \to_s \mathbf{K} \to \mathbf{V})_\perp$ | : function values |
| $n \in \texttt{Nat}$ | : integer numbers | $\mathbf{V} = \mathbf{B} \oplus \mathbf{F}$ | : values |
| $I \in \texttt{Ide}$ | : identifiers | $\mathbf{Env} = \texttt{Ide} \to \mathbf{V}$ | : environments |
| $\star \in \texttt{BinOp}$ | : binary operators | $\mathbf{K} = \mathbf{V} \to \mathbf{V}$ | : continuations |
| $\diamond \in \texttt{UnOp}$ | : unary operators | | |

```
{
  RTTI_SIGNATURE(ElementImpl, "AbstractionImpl")

private:
  ElementImpl * const expression;

public:
          AbstractionImpl (const ElementImpl * exp):
            expression(exp->copy()) { }
  virtual ~AbstractionImpl () { delete expression; }
  virtual  ElementImpl * copy () const;
  virtual  ElementImpl * subst (int db,
             const ElementImpl * val) const;
  virtual  ElementImpl * incFV (int t = 0);
  virtual  ElementImpl * evaluate ();
           ElementImpl * AbstractionImpl::apply (
             const ElementImpl * arg) const;
};
```

Methods `evaluate` and `subst` must be implemented according to a set of evaluation rules for our metalanguage, a subset of which is given in Figure 4. The notation $E \Downarrow V$ represents the evaluation relation whereas $E[\#n := F]$ denotes textual substitution. Method `incFV` implements the adjustment of de Bruijn indices, which is necessary for substituting within $\lambda$-abstractions. According to Figure 4, the implementation of some of these methods is given below:

```
ElementImpl * ApplicationImpl::evaluate () const
{
  ElementImpl * eFun = function->evaluate();
  ElementImpl * eArg = argument->evaluate();
  ElementImpl * result =
    (AbstractionImpl::isMember(*eFun))
    ? ((const AbstractionImpl *) eFun)->apply(arg)
    : new BottomImpl();
  delete eFun; delete eArg; return result;
}

ElementImpl * FixImpl::evaluate () const
{
  ElementImpl * eExp = expression->evaluate();
  ElementImpl * result =
    (AbstractionImpl::isMember(*eExp))
    ? ((const AbstractionImpl *) eExp)->apply(this)
    : new BottomImpl();
  delete eExp; return result;
}
```

```
ElementImpl * AbstractionImpl::apply (
  const ElementImpl * arg) const
{
  ElementImpl * applied = expression->subst(1, arg);
  ElementImpl * result  = applied->evaluate();
  delete applied; return result;
}

ElementImpl * AbstractionImpl::subst (int db,
  const ElementImpl * val) const
{
  ElementImpl * fv = val->incFV();
  ElementImpl * se = expression->subst(db+1, fv);
  ElementImpl * result = new AbstractionImpl(se);
  delete fv; delete se; return result;
}
```

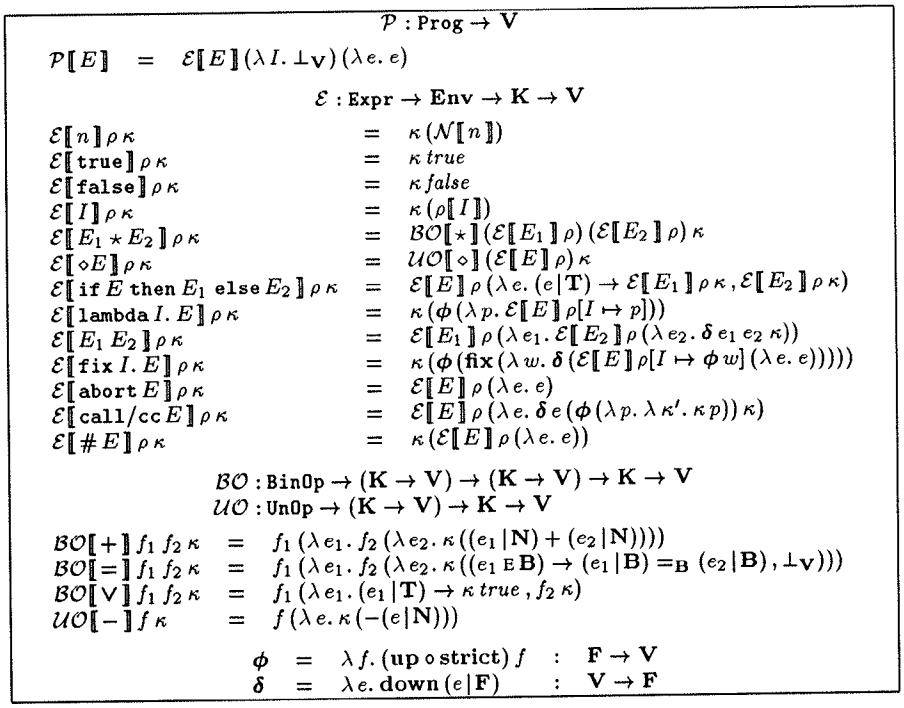
As an example, consider the element expression $(\lambda x.\ \lambda y.\ x\, y)\, (\lambda x.\ x)\, 42$, which evaluates to 42. This expression is written as $(\lambda\lambda \#2\, \#1)\, (\lambda \#1)\, 42$, when using de Bruijn indices. The evaluation of this expression is performed by the following C++ code:

```
Element x = lambda(lambda(arg(2)(arg(1))))
            (lambda(arg(1)))(Integer(42));
cout << x.evaluate() << endl;
```

where we assume that a class `Integer` has been derived from `Element`, an implementation for integer numbers has been written, as well as an `operator <<` for printing elements. Note that, although the evaluated element contains an implementation for the integer number 42, its type is `Element` and not `Integer`, as it would be expected. This is due to the untypedness of this version of our framework and will be corrected in the next section.

It is possible to improve the efficiency of our framework by implementing part of the evaluation process in element constructors, such as `lambda` and `operator ()`, and therefore reducing the size of element implementations. An improved framework contains a `simplify` method of `ElementImpl`, which performs all possible evaluations except least fixed point operations. Thus, invocation of this method will always terminate, in constrast

Figure 3: A subset of the semantic equations for PFLC.



$$\mathcal{P} : \mathbf{Prog} \to \mathbf{V}$$

$$\mathcal{P}[\![E]\!] = \mathcal{E}[\![E]\!](\lambda I.\, \bot_{\mathbf{V}})(\lambda e.\, e)$$

$$\mathcal{E} : \mathbf{Expr} \to \mathbf{Env} \to \mathbf{K} \to \mathbf{V}$$

$$\begin{array}{lcl}
\mathcal{E}[\![n]\!]\rho\kappa & = & \kappa(\mathcal{N}[\![n]\!]) \\
\mathcal{E}[\![\mathtt{true}]\!]\rho\kappa & = & \kappa\, true \\
\mathcal{E}[\![\mathtt{false}]\!]\rho\kappa & = & \kappa\, false \\
\mathcal{E}[\![I]\!]\rho\kappa & = & \kappa(\rho[\![I]\!]) \\
\mathcal{E}[\![E_1 \star E_2]\!]\rho\kappa & = & \mathcal{BO}[\![\star]\!](\mathcal{E}[\![E_1]\!]\rho)(\mathcal{E}[\![E_2]\!]\rho)\kappa \\
\mathcal{E}[\![\diamond E]\!]\rho\kappa & = & \mathcal{UO}[\![\diamond]\!](\mathcal{E}[\![E]\!]\rho)\kappa \\
\mathcal{E}[\![\mathtt{if}\ E\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2]\!]\rho\kappa & = & \mathcal{E}[\![E]\!]\rho(\lambda e.\, (e\,|\mathbf{T}) \to \mathcal{E}[\![E_1]\!]\rho\kappa, \mathcal{E}[\![E_2]\!]\rho\kappa) \\
\mathcal{E}[\![\mathtt{lambda}\, I.\, E]\!]\rho\kappa & = & \kappa(\phi(\lambda p.\, \mathcal{E}[\![E]\!]\rho[I \mapsto p])) \\
\mathcal{E}[\![E_1\, E_2]\!]\rho\kappa & = & \mathcal{E}[\![E_1]\!]\rho(\lambda e_1.\, \mathcal{E}[\![E_2]\!]\rho(\lambda e_2.\, \delta\, e_1\, e_2\, \kappa)) \\
\mathcal{E}[\![\mathtt{fix}\, I.\, E]\!]\rho\kappa & = & \kappa(\phi(\mathbf{fix}(\lambda w.\, \delta(\mathcal{E}[\![E]\!]\rho[I \mapsto \phi\, w])(\lambda e.\, e)))) \\
\mathcal{E}[\![\mathtt{abort}\, E]\!]\rho\kappa & = & \mathcal{E}[\![E]\!]\rho(\lambda e.\, e) \\
\mathcal{E}[\![\mathtt{call/cc}\, E]\!]\rho\kappa & = & \mathcal{E}[\![E]\!]\rho(\lambda e.\, \delta\, e(\phi(\lambda p.\, \lambda\kappa'.\, \kappa p))\kappa) \\
\mathcal{E}[\![\#E]\!]\rho\kappa & = & \kappa(\mathcal{E}[\![E]\!]\rho(\lambda e.\, e))
\end{array}$$

$$\mathcal{BO} : \mathbf{BinOp} \to (\mathbf{K} \to \mathbf{V}) \to (\mathbf{K} \to \mathbf{V}) \to \mathbf{K} \to \mathbf{V}$$
$$\mathcal{UO} : \mathbf{UnOp} \to (\mathbf{K} \to \mathbf{V}) \to \mathbf{K} \to \mathbf{V}$$

$$\begin{array}{lcl}
\mathcal{BO}[\![+]\!]\, f_1\, f_2\, \kappa & = & f_1(\lambda e_1.\, f_2(\lambda e_2.\, \kappa((e_1\,|\mathbf{N}) + (e_2\,|\mathbf{N})))) \\
\mathcal{BO}[\![=]\!]\, f_1\, f_2\, \kappa & = & f_1(\lambda e_1.\, f_2(\lambda e_2.\, \kappa((e_1\,|\mathbf{B}) \to (e_1\,|\mathbf{B}) =_{\mathbf{B}} (e_2\,|\mathbf{B}), \bot_{\mathbf{V}}))) \\
\mathcal{BO}[\![\vee]\!]\, f_1\, f_2\, \kappa & = & f_1(\lambda e_1.\, (e_1\,|\mathbf{T}) \to \kappa\, true, f_2\, \kappa) \\
\mathcal{UO}[\![-]\!]\, f\, \kappa & = & f(\lambda e.\, \kappa(-(e\,|\mathbf{N})))
\end{array}$$

$$\begin{array}{lcll}
\phi & = & \lambda f.\, (\mathbf{up} \circ \mathbf{strict})\, f & : \mathbf{F} \to \mathbf{V} \\
\delta & = & \lambda e.\, \mathbf{down}\,(e\,|\mathbf{F}) & : \mathbf{V} \to \mathbf{F}
\end{array}$$

to `evaluate`. Also, our memory management scheme for element implementations results in a heavy use of operators `new` and `delete`. By overloading these operators and by changing accordingly the `copy` method, it is possible to implement a smarter memory management scheme that would not copy element implementations when not necessary (e.g. by keeping reference counters). Better results would be achieved by using a proper garbage collector for C++.

## Type-safe version

Although the untyped version of the framework that we suggested in the previous section succeeds in implementing the numerous operations on Scott domains in a fairly natural way, it fails to represent the Scott domains themselves. As an attempt to provide the elements with type information, it is possible to derive classes from `Element` but operations on such elements do not propagate the type information. Furthermore, the untyped version is only able to diagnose semantic errors in element expressions (e.g. application to a non-function element) at run-time.

It is possible to create a type-safe version of our framework, implementing a typed version of our meta-language. This version represents Scott domains as classes derived from `Element`, assigns type information to elements and propagates this type information correctly and consistently in operations. In addition, it is able to detect type errors at compile-time, by using the type system of C++. A set of classes and class templates are derived from `Element`, as shown in Figure 5. The various class templates represent domain constructors and are parameterized by the types of their operands. The same figure also shows the complete hierarchy of implementations.

In the type-safe version, functions such as `evaluate` must be replaced by function templates, propagating the correct type information:

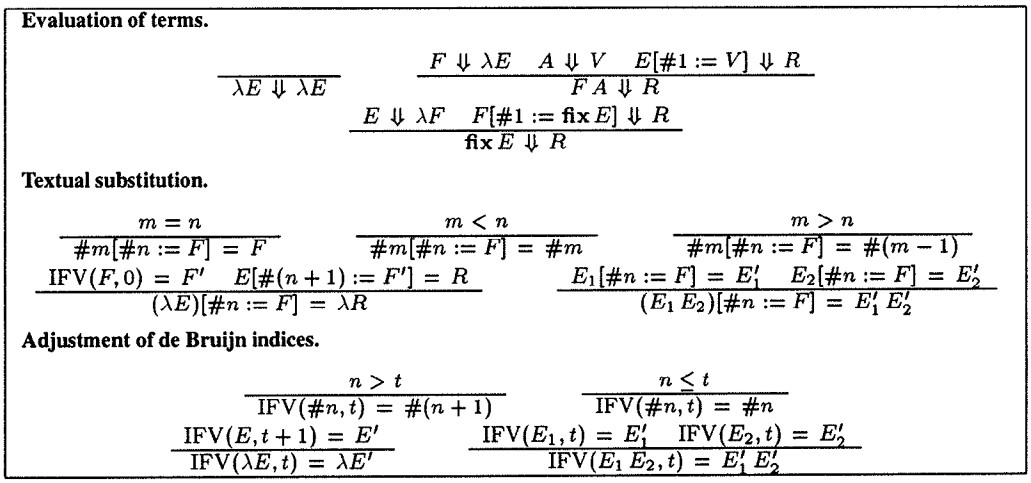
```
template<class T>
T evaluate (const T & e)
{
   return T(e.getImpl()->evaluate());
}
```

The class template `Function<In, Out>` represents function domains. It is defined as:

```
#define FUN(In, Out) Function< In, Out >

template<class In, class Out>
class Function: public Element
{
public:
   Function (ElementImpl * ei): Element(ei) { }
   Function (const FUN(In, Out) & f): Element(f) { }
   Out operator () (const In & arg) const;
};
```

Figure 4: Some evaluation rules for our meta-language with deBruijn indices.




and the corresponding domain operators are defined as:

```
template<class In, class Out>
Out FUN(In, Out)::operator () (const In & arg) const
{
  return Out(new ApplicationImpl(getImpl(),
                                 arg.getImpl()));
}

template<class T>
T fix (const FUN(T, T) & exp)
{
  return T(new FixImpl(exp.getImpl()));
}
```

In the typed version of $\lambda$-notation it is necessary to specify the type of the parameter. Also, in order to make the framework type-safe, it is also necessary to explicitely specify the type of each parameter's instance, since the C++ compiler cannot deduce the type of an expression such as `arg(1)`.[3] Therefore, the $\lambda$-notation that we use in the type-safe version of our framework is not as simple as in the untyped one. The two macros `lambda` and `arg` hide the ugly implementation details from the user. We used `operator |=` for the implementation of `lambda` because it is right associative and has a very low precedence. An empty class template has to be defined (`LambdaOperator<In>`) just to provide the parameter's type to `operator |=`.

```
#define lambda(T) (LambdaOperator< T >(0)) |=
#define arg(T, n) (T(new ParameterImpl(n)))

template<class In>
class LambdaOperator
{
public:
  LambdaOperator (int) { }
};
```

[3] This is a possible source of errors, since there is no way of checking whether the types of parameter instances are consistent with the types specified in the corresponding $\lambda$-expressions.

```
template<class In, class Out>
FUN(In, Out) operator |= (
  const LambdaOperator<In> & l, const Out & exp)
{
  return FUN(In, Out)(new AbstractionImpl(
                              exp.getImpl()));
}
```

As an example, consider again the element expression $(\lambda x : \mathbf{N} \to \mathbf{N}.\, \lambda y : \mathbf{N}.\, x\,y)\,(\lambda x : \mathbf{N}.\, x)\,42$, written as $(\lambda^{\mathbf{N}\to\mathbf{N}}\,\lambda^{\mathbf{N}}\,\#2\,\#1)\,(\lambda^{\mathbf{N}}\,\#1)\,42$, in our typed meta-language, using de Bruijn indices. Note that the type of each parameter's instance is determined by the type specified by the corresponding $\lambda$-abstraction. The evaluation of this expression is performed by the following C++ code. The evaluated element is of the expected type `Integer`.

```
Integer x = (lambda(FUN(Integer, Integer))
               lambda(Integer)
                 arg(FUN(Integer, Integer), 2)
                 (arg(Integer, 1)))
             (lambda(Integer) arg(Integer, 1))
             (Integer(42));
cout << x.evaluate() << endl;
```

We should note at this point that we still have two problems with the type-safe version of our framework, both due to the type system of C++. Type unification in templates does not work as expected in some compilers. In GNU C++, for instance, type unification fails when the formal parameter is a (reference to a) class template and the actual parameter is a subclass of the class template. The same type unification succeeds in Borland C++. Furthermore, defining recursive domains such as $\mathbf{Tree} = \mathbf{N} + (\mathbf{Tree} \times \mathbf{Tree})$ is not a straightforward task. The obvious definition would be something like:

```
class Tree;
typedef SUM(Integer, PROD(Tree, Tree)) Tree;
```

only this does not work, neither in GNU C++ nor in Borland C++. It seems that the only way to overcome this

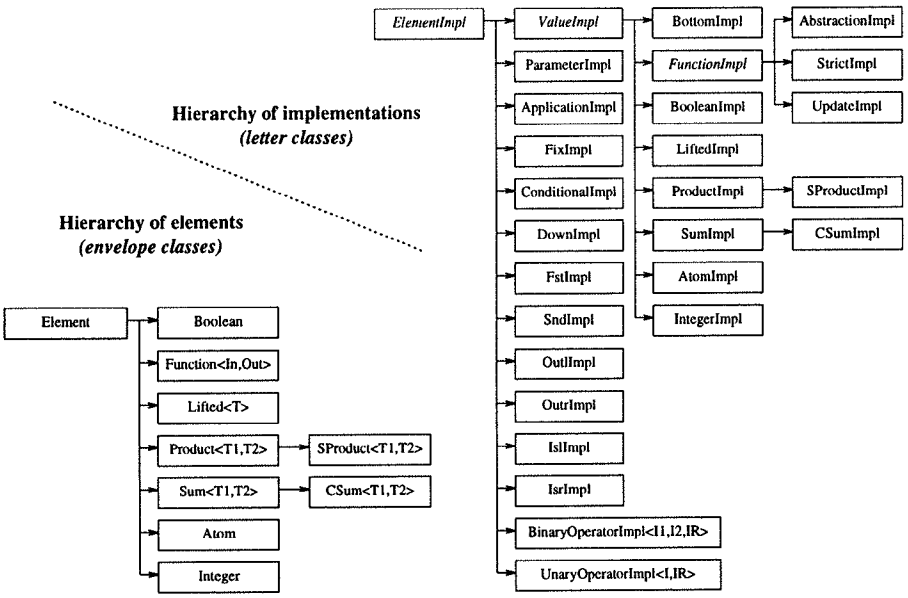



Figure 5: Class hierarchies for domain elements and implementations.

problem is the following definition, which is problematic because of our previous remark:

```
class Tree: public SUM(Integer, PROD(Tree, Tree))
{
public:
  Tree (const SUM(Integer, PROD(Tree, Tree)) & t):
    SUM(Integer, PROD(Tree, Tree))(t) { }
};
```

## Preprocessor

The framework that we suggest in the previous section is capable of expressing denotational descriptions in a fairly natural way. However, element expressions contain redundant information and this is a possible source of errors. The redundant information is the type of parameter instances. Consider the expression $\lambda\, x : \mathrm{N}.\ (\lambda\, x : \mathrm{N}.\ x)\ x$, which is written as:

```
lambda(Integer) (lambda(Integer) arg(Integer, 1))
                (arg(Integer, 1))
```

There are two difficulties with the representation of such an expression: (i) named dummies in $\lambda$-abstractions have to be expressed as de Bruijn indices, and (ii) the types of parameter instances have to be explicitely specified, although this information is redundant. The former is a matter of choice, simplifying the framework and improving its performance; on the other hand, named dummies provide a more natural way of representing $\lambda$-abstractions. The latter, however, is a problem of the implementation, due to the lack of type inference in the type system of C++.

We tried unsuccessfully to overcome these two difficulties by using the C++ preprocessor. It seems that this is not possible, because C++ lacks lexical closures and variable scoping within expressions. It is possible, however, to implement an external preprocessor that will convert $\lambda$-notation with named dummies to de Bruijn indices and explicitely typed parameter instances. We have implemented such a preprocessor consisting of 183 lines in *flex* and *bison*, with a grammar of 8 terminal and 4 nonterminal symbols. By using the preprocessor, the previous element expression can be written as:

```
lambda(x: Integer. lambda(y: Integer. y)(x))
```

No redundant information is given and this notation is much clearer and more natural than the previous one.

## Extensions

It is possible to enhance our framework by using a set of extensions to C++. The set of GNU extensions allows us to get rid of the preprocessor, without affecting the readability of element expressions. The two extensions that are particularly useful are:

- *Statement expressions*: compound statements within parentheses can appear within expressions. The last statement in the compound statement determines the value of the whole construct.
- *typeof*: a compile-time operator, referring to the type of an expression which is never evaluated. It can

be used in any type expression and is very useful in combination with statement expressions.

The first and obvious improvement to our framework, by using the two GNU extensions that were mentioned above, is that our meta-language can now use named dummies instead of de Bruijn indices in $\lambda$-abstractions. The set of evaluation rules must be revised, but the only point that needs special treatment is textual substitution in $\lambda$-abstractions, solved by renaming dummies whenever necessary. A second important improvement is that we can define a class template for domains themselves, namely `Domain<E>`, in addition to classes for domain elements. This allows us to represent domain constructors as C++ operators, and use such *domain expressions* in element expressions, whenever this is required. The type parameter `E` that is used in this class template represents the type of domain elements. We can then define:

```
Domain<Integer> N;
Domain<Boolean> T;
```

The class template `Domain<E>` is defined as:

```
template<class E>
class Domain
{
public:
  Domain () { }
  E * nullInstance () const { return NULL; }
  E bottomInstance () const
    { return E(new BottomImpl()); }
  E operator () (const E & e) { return e; }
};

#define bottom(T) ((T).bottomInstance())
#define OBJ(T) typeof( *((T).nullInstance()) )

template<class E1, class E2>
Domain<FUN(E1, E2)> operator |= (
  const Domain<E1> & d1, const Domain<E2> & d2)
{
  return Domain<FUN(E1, E2)>;
}
```

where `operator |=` represents the domain constructor for functions and `operator ()` simplifies the expression of domain elements, allowing expressions such as `N(42)`. The macro `bottom` returns the bottom element of a domain, whereas the macro `OBJ(T)` returns the type of a given domain's element. Finally, the $\lambda$-notation can be implemented as:[4]

```
#define lambda(v, T, E) (lambdaOperator(#v, T,  \
        ({ OBJ(T) v(new ParameterImpl(#v));     \
           new typeof( E )(E); })))

template<class In, class Out>
FUN(In, Out) lambdaOperator (
  const char * dummy, const Domain<In> &, Out * exp)
{
  FUN(In, Out) result(new AbstractionImpl(
    dummy, exp->getImpl()));
  delete exp; return result;
}
```

[4] We had to use a pointer type for the result of the statement expression because of a bug in GNU C++ statement expressions.

As an example, consider again the element expression $(\lambda x : \mathrm{N} \to \mathrm{N}.\ \lambda y : \mathrm{N}.\ x\,y)\,(\lambda x : \mathrm{N}.\ x)\,42$. This time, the evaluation of this expression is performed by the following C++ code:

```
OBJ(N) x = (lambda(x, N|=N, lambda(y, N, x(y))))
           (lambda(x, N, x))(N(42));
cout << x.evaluate() << endl;
```

A third possible extension, the presence of which would change our framework radically and would much improve its performance, is *unnamed functions* as suggested in [2]. Unnamed functions would render unnecessary the definition of a special class for $\lambda$-abstractions and would much simplify our hierarchy of implementations. They would also significantly narrow the gap between C++ and functional languages. Unfortunately, unnamed functions have not been adopted, to the best of our knowledge, in any popular C++ compiler.

## Example

We have implemented an interpreter for PFLC using the denotational description that was given in Figures 2 and 3. The program consisted of 971 lines of code, including a number of test programs, whereas a similar implementation in ML consisted of only 426 lines of code. A significant part of the C++ implementation was devoted to the implementation of syntactic domains (336 lines, compared to 38 lines in the ML implementation), since we decided not to use coalesced sums and products in order to simplify the equations. From this experience, it is now clear that we need to facilitate the implementation of syntactic domains in our framework.

The following code is a part of the implementation of PFLC's semantic equations, in the form recognized by our preprocessor. Syntactic and semantic domains were defined earlier in the C++ program and the problem of recursive domains, such as V, was handled by defining empty classes, as discussed earlier.

```
V semP (const Expr & E)
{
  return semE(E)(lambda(I: Ide. BOTTOM(V)))
                (lambda(e: V. e));
}

FUN(Env, FUN(K, V)) semE (const Expr & E)
{
  if (E.is("Int'"))
    return lambda(rho: Env. lambda(kappa: K.
      kappa(V::inl(B::inl(semN(E[1]))))));
  else if (E.is("Fix'"))
    return lambda(rho: Env. lambda(kappa: K.
      kappa(phi(fix(lambda(w: FUN(V, FUN(K, V)).
        delta(semE(E[2])(update(rho, E[1], phi(w)))
                        (lambda(e: V. e)))))))));
  else if (E.is("Abort'"))
```

```
    return lambda(rho: Env. lambda(kappa: K.
      semE(E[1])(rho)(lambda(e: V. e))));
  ...
}
```

The scheme for accessing syntactic domains (method `is` and `operator []`) is a simplification of the one that was actually used. In fact, this scheme cannot be implemented in a type-safe way without dynamic type casting in function `semE` and a hierarchy of class templates for syntactic domains. Except for the (rather clumsy) implementation of syntactic domains, we believe that the implementation of an interpreter for PFLC by using the suggested framework was entirely successful and demonstrates the ability of C++ to implement denotational descriptions in a natural way.

## Conclusion

We have presented a type-safe framework for the implementation of denotational descriptions in C++, exploiting the object-oriented programming paradigm. We have demonstrated that the widespread general-purpose C++, although not a natural choice for this problem domain, can be successfully used for the study of programming languages. The suggested framework is implemented by using pure C++, although a set of extensions would be valuable as discussed earlier. It could be translated to other object-oriented languages supporting inheritance, polymorphism and generic types.

The main criteria in the evaluation of our framework are *expressiveness* and *performance*. Concerning the first criterion, our framework provides a natural way of expressing denotational descriptions. Some drawbacks of our approach, imposed by C++, have been discussed in previous sections. Compared with other general purpose programming languages that have been suggested, our framework is inferior to implementations in functional languages such as ML. The object-oriented programming paradigm results in more natural denotational descriptions than possible implementations using imperative programming languages.

On the other hand, we have consciously neglected performance in the implementation of our framework. Performance is reduced because of three factors:

- The memory management of C++ is poor for the requirements of this problem domain. This can be alleviated by using a garbage collector and overloading operators `new` and `delete`.
- $\lambda$-abstractions and high-order functions are managed by the programmer instead of the compiler, resulting in poor optimizations when compared to those that are performed by the compiler of a functional programming language. This can only be solved by extending C++ with an *unnamed function* feature, as discussed earlier.
- A large number of virtual methods are required. This problem though is inherent in object-oriented programming with C++.

Nevertheless, we do not consider performance to be a very significant factor in the evaluation of our framework. Implementations of denotational descriptions are mainly used as experimental execution models for the study of programming languages. In this context, performance is seldom an important issue.

ML is a much more natural choice than C++ for the implementation of denotational descriptions, with respect to both evaluation criteria. There are two main reasons for this: (i) ML is a functional programming language and denotational descriptions are inherently functional; and (ii) ML has a polymorphic static type system, featuring type inference and high-order functions as first-class objects. In comparison, C++ lacks functional features. In the suggested framework we have attempted to integrate some functional features in C++ without extending the language. Furthermore, although C++ has a powerful type system supporting inheritance, polymorphism and generic types by means of templates, implementations in various C++ compilers present significant limitations, probably because its semantics are not yet clearly defined in a generally accepted standard.

## References

[1] ALLISON, L. Programming denotational semantics. *Computer Journal 26*, 2 (1983), 164–174.

[2] BREUEL, T. Lexical closures for C++. In *Proceedings of the USENIX C++ Conference* (Denver, CO, Oct. 1988), pp. 293–304.

[3] COPLIEN, J. O. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[4] DAMI, L. *Software Composition: Towards an Integration of Functional and Object Oriented Approaches*. PhD thesis, Université de Genève, Apr. 1994.

[5] DE BRUIJN, N. Lambda-calculus notation with nameless dummies: A tool for automatic formula manipulation. *Indag. Mat. 34* (1972), 381–392.

[6] GORDON, M. J. C. *The Denotational Descriptions of Programming Languages.* Springer Verlag, Berlin, Germany, 1979.

[7] GUNTER, C. A. *Semantics of Programming Languages: Structures and Techniques.* Foundations of computing. MIT Press, Cambridge, MA, 1992.

[8] GUNTER, C. A., AND SCOTT, D. S. Semantic domains. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., vol. B. Elsevier Science Publishers B.V., 1990, ch. 12, pp. 633–674.

[9] KÜHNE, T. Inheritance versus parameterization. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS Pacific'94)* (London, 1995), C. Mingins and B. Meyer, Eds., Prentice Hall, pp. 235–245. For correct version ask author; proceedings contain corrupted version.

[10] LAÜFER, K. A framework for higher-order functions in C++. In *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)* (Monterey, CA, 26–29 June 1995), pp. 103–116.

[11] MOSSES, P. D. Denotational semantics. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., vol. B. Elsevier Science Publishers B.V., 1990, ch. 11, pp. 577–631.

[12] PAGAN, F. G. Algol 68 as a metalaguage for denotational semantics. *Computer Journal 22*, 1 (1979), 63–66.

[13] RAMSDELL, J. D. CST: C state transformers. *ACM SIGPLAN Notices 30*, 12 (Dec. 1995), 32–36.

[14] ROSE, J. R., AND MULLER, H. Integrating the Scheme and C languages. In *Conference Record of the ACM Symposium on Lisp and Functional Programming* (San Francisco, CA, 1992), pp. 247–259.

[15] SCHMIDT, D. A. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Newton, MA, 1986.

[16] SCOTT, D. S. Domains for denotational semantics. In *International Colloquium on Automata, Languages and Programs* (Berlin, Germany, 1982), vol. 140 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 577–613.

[17] SITARAM, D., AND FELLEISEN, M. Reasoning with continuations II: Full abstraction for models of control. In *Conference Record of the ACM Symposium on Lisp and Functional Programming* (1990), M. Wand, Ed., ACM Press, pp. 161–175.

[18] STOY, J. E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[19] TENNENT, R. D. The denotational semantics of programming languages. *Communications of the ACM 19*, 8 (Aug. 1976), 437–453.

[20] WATT, D. A. Executable semantic descriptions. *Software Practice and Experience 16*, 1 (Jan. 1986), 13–43.

---

*Nikolaos S. Papaspyrou is a Ph.D. candidate in the Department of Electrical and Computer Engineering at the National Technical University of Athens (NTUA), Greece. His doctoral research focuses on the denotational semantics of programming languages and its relation with the software development process. Other research interests include intelligent software agents, distance learning and educational software. He received a M.Sc. in computer science from Cornell University, Ithaca, NY, and a B.Sc. in electrical and computer engineering from NTUA.*