

Automatic Generation of Parallelizing Compilers for Object-Oriented Programming Languages from Denotational Semantics Specifications

Prakash K. Muthukrishnan and Barrett R. Bryant

Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, Alabama 35294-1170, U. S. A.
{prakash, bryant}@cis.uab.edu

Abstract. The denotational semantics of object-oriented programming languages (OOPL's) are used to derive the control and data dependencies that exist within programs and this information is then used to produce parallel object code by a compiler. This approach is especially suited for OOPL's because of the concurrency inherent in their semantics. A denotational semantics of an OOPL called SmallC++ is developed with some specialized operations that facilitate the automatic generation of a parallelizing compiler for SmallC++. The result is a compiler which generates code for shared or distributed memory multi-processors, thereby achieving a language implementation which realizes fully transparent parallel object-oriented programming.

1 Introduction

The object-oriented (OO) programming paradigm provides linguistic support for objects, classes and inheritance [18]. It supports the software engineering principles of data abstraction, information hiding, modular design, and code reuse, and it allows the modeling of real world objects in a natural fashion. Since objects are independent entities that communicate through message passing, the object-oriented paradigm lends itself naturally to the development of distributed systems. This has been pursued primarily by developing object-oriented languages with explicit concurrency constructs called concurrent object-oriented programming languages. For surveys of general principles, the reader is referred to [17]. The various concurrent object-oriented programming systems provide some explicit concurrency constructs which the programmer has to use to obtain parallelism. The development of concurrent object-oriented programming systems have been complemented by efforts in defining the formal semantics of such systems [20, 16]. There has been only very little work done in automatic parallelization of OO languages [19, 4, 2].

We use formal semantics specification in deriving the parallelism that is available in object-oriented programs, which are inherently parallel, and propose a tool which a software developer can use to develop parallel applications by just writing sequential object-oriented code. Such technology has not been applied

for OOP languages. We propose a methodology that extends existing techniques in automatic compiler generation [9] and parallelizing compilers [21]. One immediate benefit of such a tool is that it could be used to enable an existing object-oriented programming language, such as Smalltalk[5] or C++ [15], to be directly compiled into parallel code.

The paper is structured as follows. In Section 2 we present SmallC++ [11], a subset of C++ that we believe to be tractable for developing a prototype of a parallelizing compiler. In Section 3 we present the background information on Denotational Semantics and how it can be used in generating compilers. This is illustrated with an example using the semantics of SmallC++ in Section 4, where we present the semantics of SmallC++. The use of denotational semantics to produce a parallelizing compiler and the generation of the parallel target code is explained in Section 5. And finally in Section 6 we present the summary of our project and future work.

2 SmallC++

SmallC++ is a distilled version of the object-oriented programming language C++. The object-oriented features of SmallC++ include classes, objects, message-passing and inheritance with dynamic binding. In our subset, we can create classes and instances of a class (objects). A user programming with our subset can declare a member (data or function) to be *private*, *protected* or *public*. Since we do not support pointers, dynamic creation of objects is not supported in our language. Our subset allows only public derivation; public derivation in C++ models the *is-a* relationship and thus supports inheritance. Like Smalltalk, we do not support multiple inheritance. We have retained the keyword *virtual* in our subset and also the reference feature of C++. Combining these two features, polymorphism and late binding is achieved. Figure 1 is an example program written in SmallC++.

3 Formal Semantics of Programming Languages and Semantics-Directed Compilers

A programming language has two main characteristics, namely *syntax*, which describes the appearance and structure of its sentences, and *semantics*, which describes the assignment of meanings to the sentences. The syntactic specification of languages has been standardized using the Backus-Naur Form (BNF), or context-free grammar, and there are many automated tools which support the syntax analysis phase of a compiler (e.g. lex and Yacc [1]). In semantic specification, there is no such standard, although of the various methods for defining the formal semantics of programming languages, denotational semantics is the most commonly used because of its unique combination of mathematical properties and formalization of execution models [14].

A denotational semantics specification of a programming language typically consists of the following components:

```

class Point
{
    private:
        int x;
        int y;
    public:
        Point (int a, int b) { x = a; y = b; }
        int GetX () { return x; }
        int GetY () { return y; }
};

void main ()
{
    Point obj1 (3,4);
    Point obj2 (5,6);
    int d;
    // Note that all four method invocations can execute in parallel
    d = sqrt(square(obj1.GetX() - obj2.GetX()) +
             square(obj1.GetY() - obj2.GetY()));
    cout << d;
}

```

Fig. 1. SmallC++ example program

1. *Abstract syntax* gives the abstract syntactical details of the language, which can be derived from its *concrete syntax*, the syntax described by the BNF grammar. It is typically defined as a set of *syntax domains*, each of which describes an abstract syntax representation of the corresponding concrete syntax construction.
2. *Semantic algebra* describes the various *semantic domains* and the operations associated with the elements of those domains. Basic domains include numbers, Boolean values, lists, etc. More complex domains may be structured in three ways:
 - (a) *Product domains* are domains of tuples whose components consist of other semantic domains. These are typically denoted by $D_1 \times D_2 \times \dots \times D_n$, where D_1, D_2, \dots, D_n are the component domains. The main product domain operation is a *selector* to extract a component, for example, by its position in the tuple.
 - (b) *Sum domains*, also called disjoint unions, are unions of sets of values, each value being tagged with the type of the domain it belongs to. These are typically denoted by $D_1 + D_2 + \dots + D_n$, where D_1, D_2, \dots, D_n are the component domains. Sum domain operations include *injection* operations to insert a value into the sum with the appropriate type tag, and

projection operations to extract a value which is tagged with a certain type.

- (c) *Function domains* are mappings of one domain into another, expressed as $D_1 \rightarrow D_2 \rightarrow \dots \rightarrow D_n$, which denotes a function over domain D_1 whose range is $D_2 \rightarrow \dots \rightarrow D_n$. That is, the \rightarrow operator associates to the right. Function domains are usually modelled using λ -calculus and may be used to represent mappings such as the mapping of an identifier into its denotation, or the mapping of a memory location to its value. Examples of λ -calculus expressions are $addone = \lambda y.y + 1$ and $apply = \lambda f.\lambda x.fx$, equivalent to the more standard function notations $addone(y) = y + 1$ and $apply(f, x) = f(x)$, respectively. A detailed discussion of λ -calculus is beyond the scope of this paper (see [6] for an introduction).
3. *Semantic functions* express the denotation of each abstract syntax phrase in terms of the denotations of its components. By applying the semantic functions, it is possible to elaborate the denotation of a complete program as the composed denotation of its various components. Through this process, all syntax domains are translated into semantic domains. At the highest level of this mapping, programs are translated into their denotations, in the form of a λ -calculus expression that maps input into output. It is this mapping which allows compilers to be constructed from denotational semantics specifications.

The process of compiling a program according to a denotational semantics specification is called *semantics-directed* compiling and the theory behind this process dates back over thirty years. The typical process is to use the semantic functions of the denotational semantics to translate a program P into a functional form, e.g. in λ -calculus, which maps input I to output O . That is, a denotational semantics may be regarded as a function over domain $P \rightarrow I \rightarrow O$. If we have P , then we can get $I \rightarrow O$. If we have both P and I , we can get O . Since a denotational semantics defines both *static* and *dynamic* semantics, i.e. type checking as well as run-time behavior, it is usual for such a compiler to have at least two phases, one to evaluate the semantic functions involved with the static semantics at compile-time, and the other to perform the translation into the final functional form which represents only the functions which should be executed at run-time. We shall refer to the functional form produced by this process as “denotational code.” This code may be executed by any number of different functional programming language interpreters, such as ML [10], or it may be compiled into actual machine code using functional programming language implementation techniques [13]. This process is illustrated in Figure 2.

Since this approach may be used to produce a compiler from a language specification, it is called *automatic compiler generation*. The most usual approach for producing target code in such systems is not to compile the denotational code as a functional program but rather to consider it as representing a fixed set of operations called *combinators*. A *combinator* is a λ -expression without free variables, essentially a function without global variables (*addone* and *apply*

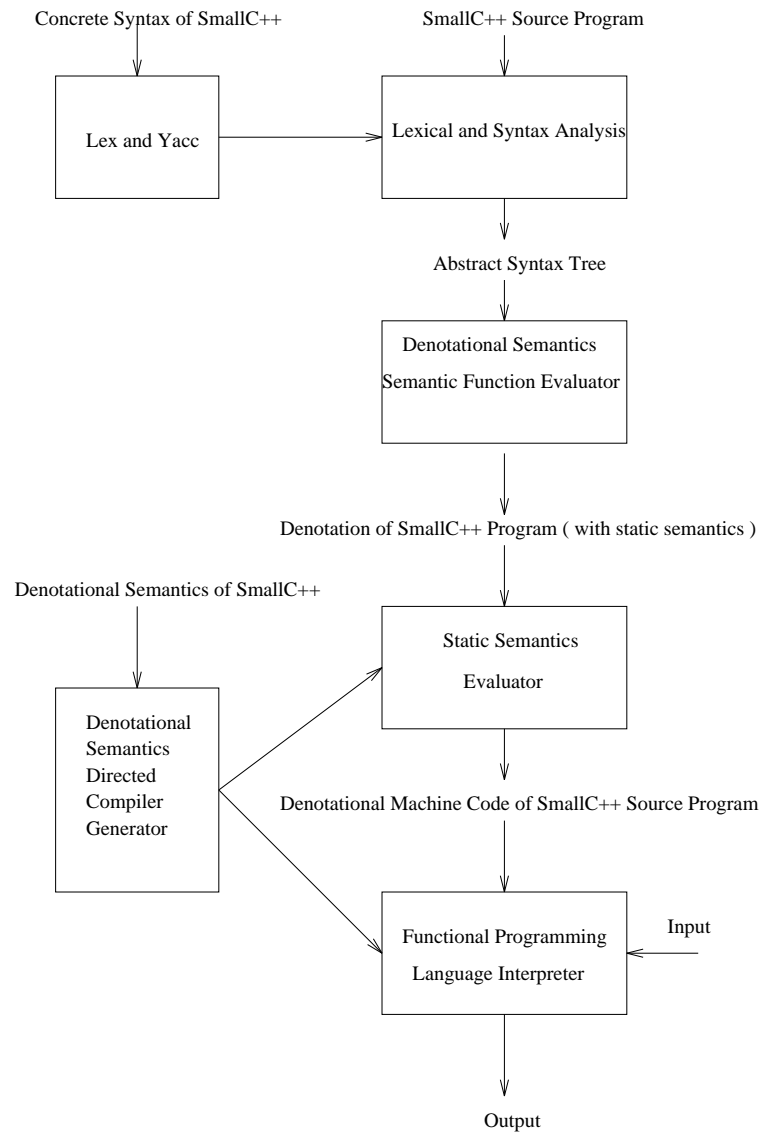


Fig. 2. Overview of semantics-directed compilation process

mentioned previously are examples of combinators). Combinators generated as denotational code are typically denotational expressions which have direct representations in conventional target machine languages (e.g. arithmetic and logical operations, conditional and iterative constructions, etc.). This greatly facilitates the automatic generation of target machine code.

4 Semantics of SmallC++

Denotational semantics functions typically have two styles, *direct* and *continuation*. Direct semantics models the semantic representation of a program as a standard composition of functions, e.g. $f_n(\dots(f_2(f_1))\dots)$, where f_1 is the first function denoted by the program, f_2 the second, etc. Continuation semantics, on the other hand, models this composition as a sequence, each function in the sequence taking the “rest of the sequence” as an argument, e.g. $f_1(f_2(\dots(f_n)\dots))$, where $f_2(\dots(f_n)\dots)$ is the *continuation* of f_1 . Not only is this approach more like a sequence of machine instructions when each function is a combinator, but also it is useful in modelling more powerful control-flow constructions than direct semantics, such as function call and return, loop exits, etc. We present the denotational semantics of SmallC++ using the continuation style approach.

4.1 Abstract Syntax

We begin our denotational semantics specification with the abstract syntax for SmallC++ shown in Figure 3. Each of the indicated syntax domains may be thought of as a model of an abstract syntax tree, whose children are themselves abstract syntax trees defined by other domains. For example, an Assignment Statement, AS, is a syntax tree with operator =, and two children, defined by VAR and E, respectively, each of which denotes an additional abstract syntax tree defined by their abstract syntax rules. From our SmallC++ example, it can be seen that the single statement in `main` matches the AS syntax domain, with `d` being the VAR and the `sqr` expression corresponding to E. The syntax domain rules for E break this down further. VAR syntax domain models identifiers that could be class-identifiers, object-identifiers, identifiers to denote ground types or function identifiers.

4.2 Semantic Algebras

In this section we present some of the semantic algebras used in modeling our language. The basic semantic domains are Boolean Truth Values, Identifiers, which are strings of characters, and Natural numbers. These basic domains may be composed into larger semantic domains to represent denotations of program entities. The Denotable-Values domain models the values that are denoted by identifiers in a program. SmallC++ identifiers may denote memory locations (Loc) if they are call-by-reference formal parameter variables, procedures (Proc) if they are

```

Prog := P
P    := D1, D2, D3, ..., Dn
D    := SD | CD | FD
SD   := int I1 | Ic I1 | int I1 [n1]...[nn] | Ic I1 [n1].[nn]
CD   := class Ic ML | class Ic : public Ic ML
FD   := TS If AL BD
TS   := int | void | Ic | int Ic :: | void Ic :: | Ic Ic ::
ML   := AS ML1, ML2, ..., MLn | SD | FD | virtual FD = 0
AS   := private | public | protected
AL   := AL1, AL2 | int I1 | Ic I1 | int & I1 | Ic & I1
BD   := SD CSL | SD CSL

CSL  := C1;C2
C    := AS | IF | FOR | OUT | IN | RET
AS   := VAR = E
IF   := if E CSL else CSL
FOR  := for AS1 E AS2 CSL
OUT  := cout << E
IN   := cin >> E
RET  := return E

EL   := E1, E2
E    := I | N | If (EL) | UOP VAR | UOP N | UOP If(EL) | E1 BOP E2
      | E1.E2 | OID.If (EL) | this → If (EL)
VAR  := I | VAR.I | this → I | VAR [EL]

```

Fig. 3. Abstract syntax of SmallC++

functions, as well as classes, arrays, system variables (like **cin** and **cout**), objects, and error values (e.g. if the identifier doesn't have a declaration). Since these are all different types, a sum domain is used to model the union of the various types. The Storable-Values domain is the set of values that can be stored in locations in the store, either integers or files, in the case of **cin** and **cout**, and Expressible-Values is the domain that is used to model the values produced by expression evaluations.

Denotable-Values Domain $d \in \mathbf{Dv} = \text{Loc} + \text{Proc} + \text{Class} + \text{Array} + \text{System-Var} + \text{Object} + \text{ErrValue}$

Storable-Values Domain $v \in \mathbf{Sv} = \text{Nat} + \text{File}$

Expressible-Values Domain $e \in \mathbf{Ev} = \text{Nat} + \text{Tr} + \text{Dv}$

The symbol table which models bindings of identifiers is defined using the Environment domain with basic symbol table operations. The environment is regarded as a mapping of identifiers to denotations, so is modelled as a function. Some of the operations defined on this domain are *initialenv* to set up the initial environment for a local scope, *accessenv* which accesses an identifier in the scope, *updateenv* to update the symbol table with a new definition, and a composition

operator to combine symbol tables. Environment is a static semantics domain which would not be present in the denotational machine code.

Environment Domain $r \in \mathbf{Env} = \text{Id} \rightarrow \text{Dv}$

The principal dynamic semantics domain of any object-oriented language is the memory store, which is modelled after the memory store of a conventional stored-program computer, organized as a stack of activation records, where every address is computed from a relative location within a stack frame. Like Environment, Store consists of a mapping function which associates locations with the Storable-Values that are stored in the locations. A Location is simply an address, so may be modelled by natural numbers. The store is denoted by a triple where the first component represents the stack frame, the second component the top of the stack and the third component the map of Location to Storable Values.

Store Domain $s \in \mathbf{Store} = \text{Loc} \times \text{Loc} \times (\text{Loc} \rightarrow \text{Sv})$

Operations:

$access : \text{Store} \rightarrow \text{Loc} \rightarrow [\text{Sv} + \{\text{Errvalue}\}]$

$access = \lambda (sp, top, map) . \lambda l . l < top \rightarrow map(l) \parallel \text{ErrValue}$

$update : \text{Store} \rightarrow \text{Loc} \rightarrow \text{Sv} \rightarrow [\text{Store} + \{\text{errorStore}\}]$

$update = \lambda (sp, top, map) . \lambda l . \lambda v . l < top \rightarrow ([\mapsto v](map, top)) \parallel \text{errorStore}$

$mark-locn : \text{Store} \rightarrow (\text{Loc} \times \text{Loc} \times \text{Store})$

$mark-locn = \lambda (sp, top, map) . (sp, top, (sp+top, 0, map))$

$alloc-locn : \text{Store} \rightarrow \text{Nat} \rightarrow (\text{location} \times \text{Store})$

$alloc-locn = \lambda (sp, top, map) . \lambda n . (sp+top, (sp, locn + 1, map))$

$dealloc-locn : \text{Store} \rightarrow \text{Loc} \rightarrow \text{Loc} \rightarrow \text{Store}$

$dealloc-locn = \lambda (sp, top, map) . \lambda l_1 . \lambda l_2 . (l_1, l_2, map)$

$initial-store : \text{File} \rightarrow \text{Store}$

$initial-store = \lambda f . \text{let } s' = \text{empty-store} \text{ in } ([\text{cin} \mapsto f]s')$

$empty-store : \text{Store}$

$empty-store = (0, 0, \lambda l . \text{inErrValue}())$

A continuation is formally a function that maps intermediate results (that are expected by the “rest of the program”) to their final answers. We use three standard continuations in our formal semantics: Command continuations, Expression continuations and Declaration continuations. A command modifies a store and passes this modified store to the rest of the program following it. So we define the command continuations as follows:

Command Continuations Domain $c \in \mathbf{Cc} = \text{Store} \rightarrow \text{Ans}$

A Declaration modifies the Environment and possibly the store. Since Declarations pass the modified environment and the store to the rest of the program following them we define Declaration Continuations as:

Declaration Continuations Domain $u \in \mathbf{Dc} = \text{Env} \rightarrow \text{Store} \rightarrow \text{Ans}$

Since Expressions pass their values and the modified store to the rest of the program following them, we define Expression Continuations as following:

Expression Continuations Domain $k \in \mathbf{Ec} = \text{Ev} \rightarrow \text{Store} \rightarrow \text{Ans}$

4.3 Semantic Functions

We illustrate the semantic functions for SmallC++ using the example program presented earlier. The top-level function is **PROG** which maps a program, *Prog*, into a functional mapping an input file, *File*, into an Answer, *Ans*. The domain *Ans* is the final answer which will be a finite string of values (of domain *Nat*) ending either with a **stop** (in case of a successful completion) or **error** (in case of a failed execution).

PROG : *Prog* \rightarrow *File* \rightarrow *Ans*
PROG \llbracket *Prog* $\rrbracket = \lambda f. \mathbf{P} \llbracket$ *Prog* \rrbracket *initialenv initial-store* *f*

P : *ListOfDefinitions* \rightarrow *Env* \rightarrow *Cc*
P \llbracket *D*₁, *D*₂, ... *D*_{*n*} \rrbracket *r* :=
 $\mathbf{D} \llbracket$ *D*₁ \rrbracket *r* ($\lambda r_1 . \lambda s_1 . \mathbf{D} \llbracket$ *D*₂ \rrbracket *r*[*r*₁] (...
($\lambda r_{n-1} . \lambda s_{n-1} \mathbf{D} \llbracket$ *D*_{*n*} \rrbracket *r*_{*n*} ($\lambda r_n . \lambda s_n .$
 $\mathbf{R} \llbracket$ *main* \rrbracket (*r*_{*f*} ($\lambda e_{final} . \lambda s_{stop} . \text{result } s_{stop} \text{ "stop" }) s_n)$
 $s_{n-1}) \dots) s_1)$
where $r_n = r[r_1[\dots[r_{n-1}]\dots]]$

A program is a list of declarations and the denotation of a program is a function to construct an environment from the list of declarations and then invoke the function **main** to start the execution. The static semantics of the declarations will be simplified during static semantics evaluation and we will be left with a denotation as a pure dynamic semantic function. For our example program, there is only a declaration of the **Point** class and the **main** function.

Of the various static semantic functions, we would like to concentrate our discussion on the semantics of a SmallC++ function definition, since it must also define the dynamic semantics of a function. A function definition binds the denotation of a function to the function identifier *I_f* in the current environment *r*. The denotation of the function is a component of the Procedure domain, a type for the function and its denotation, and is injected into that domain using *in-Proc*. The denotation of the function is expressed as a λ -expression which takes a Command Continuation *c*, memory store *s*, and set of Expressible Values (the actual parameter values), and returns the result of evaluating the body of the function, *BD*, with the formal parameter list, *AL*, bound to the values of the actual parameters. This denotation also contains the environment *r*, since the body may contain declarations that statements that require resolving the semantics of identifiers from the environment (in the semantics of *BD*, these declarations are elaborated by the semantic function **SD** which we have not shown here). This environment would also be simplified in static semantics processing, leaving the purely dynamic semantics part of the denotation. Only at run-time will this denotation actually be executed, once the function is actually called and the actual parameter values are known. Here we merely associate this denotation with the function identifier in the environment. It is the capability of λ -calculus to denote such functions that makes it extremely suitable for our semantic representation.

FD : FunctionDefinition \rightarrow Env \rightarrow Dc \rightarrow Cc

FD \llbracket TS I_f AL BD \rrbracket r u :=
 let $t_1 =$ TS \llbracket TS \rrbracket in
 let $r_1 =$ *updateenv*(I_f , inProc (t_1 , p), r) in u r_1
 where p = λ c. λ s. λ $e_1 \dots e_n$.
 BD \llbracket BD \rrbracket (**AL** \llbracket AL \rrbracket (r s ($e_1 \dots e_n$)) c)

BD : Body \rightarrow Env \rightarrow Store \rightarrow Cc \rightarrow Ans

BD \llbracket SD CSL \rrbracket r c := **SD** \llbracket SD \rrbracket r (λ r_1 . λ s_1 . **CSL** \llbracket CSL \rrbracket r_1 (λ s_2 . c s_2) s_1)

After processing the declarations for our example program, we will have an environment consisting of the **Point** class and the **main** function. The declarations inside each will have been processed in a similar manner. Let us turn our attention to the body of **main**. The semantics of a CommandsList is defined as taking an environment and a command continuation and returning a command continuation.

CSL : CommandsList \rightarrow Env \rightarrow Cc \rightarrow Cc

CSL \llbracket $C_1;C_2$ \rrbracket r c := **CSL** \llbracket C_1 \rrbracket r (λ s' . **CSL** \llbracket C_2 \rrbracket r c s')

CSL \llbracket C \rrbracket := **C** \llbracket C \rrbracket

The first command in the CommandsList, C_1 , is evaluated with the current environment r and the continuation is the evaluation of the second command, C_2 , which may actually be another CommandsList. Individual statements are evaluated using the **C** function.

C : Command \rightarrow Env \rightarrow Cc \rightarrow Cc

C \llbracket I = E \rrbracket r c := **L** \llbracket I \rrbracket r (λ l. λ s_1 . isLoc l \rightarrow
R \llbracket E \rrbracket r (λ e. λ s_2 . c (*update* s_2 l e)) s_1 \llbracket error \rrbracket)

C \llbracket cout \llcorner E \rrbracket r c := **R** \llbracket E \rrbracket r (λ e. λ s_1 . c (*updatecout* s_1 e))

Here we show the semantics of the assignment and output statements which are used in the example program. An assignment denotation requires the location of the identifier, I, on the left side of the assignment, which is achieved by evaluating the **L** function which looks this identifier up in the environment and returns its denotation. If this denotation is not a location, then we have an error denotation. Note that the way this works is to pass an expression continuation along with the environment to **L**. This expression continuation contains a call to the expression evaluation function **R** which takes an expression E, environment and expression continuation for updating the location of I with the value of E. The output function similarly evaluates its expression using **R**, passing an expression continuation to update the output stream component of the store. The **L** and **R** rules are further evaluated below.

R : E \rightarrow Env \rightarrow Ec \rightarrow Cc

R \llbracket I \rrbracket r k := let d = *accessenv*(I , r) in isLocn (d) \rightarrow deref k d \llbracket error \rrbracket

$\mathbf{L} : E \rightarrow \text{Env} \rightarrow \text{Ec} \rightarrow \text{Cc}$

$\mathbf{L} \llbracket I \rrbracket r k := \text{let } d = \text{accessenv}(I, r) \text{ in } k d$

The expression `obj1.GetX() - obj2.GetX()` in our program is evaluated using the following semantic function where E_1 represents the expression `obj1.GetX()` and E_2 represents the expression `obj2.GetX()`. Here first we evaluate Expression E_1 to get the value e_1 and the modified store s_1 . We then evaluate E_2 using the new store and get the value e_2 and the modified store s_2 . Values e_1 and e_2 are passed to the BOP function which applies the binary operation on the values and returns a new values which is passed along with the store s_2 to the continuation k .

$\mathbf{R} \llbracket E_1 \text{ BOP } E_2 \rrbracket r k :=$

$$\begin{aligned} & \mathbf{R} \llbracket E_1 \rrbracket r (\lambda e_1 . \lambda s_1 . \text{isNat}(e_1) \rightarrow \\ & \mathbf{R} \llbracket E_2 \rrbracket r (\lambda e_2 . \lambda s_2 . \text{isNat}(e_2) \rightarrow \\ & k \mathbf{BOP} \llbracket \text{BOP} \rrbracket (e_1, e_2) s_2 \llbracket \text{error} \rrbracket s_1 \llbracket \text{error} \rrbracket) \end{aligned}$$

The semantics of function calls are defined by the following equation where we first obtain the denotation of the function (main in our example program), create a new stack frame, then evaluate the arguments, and execute the body of the function in this new store. At the end of execution of the body the return value is stored on top of the stack and is returned to the caller of the function.

$\mathbf{R} \llbracket I_f(E_1, E_2, \dots E_n) \rrbracket r k :=$

$$\begin{aligned} & \text{let } d = \text{accessenv}(I_f, r) \text{ in let } l_s \ l_t \ s_{old} = \text{mark-locn } s \text{ in} \\ & \mathbf{R} \llbracket E_1 \rrbracket r (\lambda e_1 . \lambda s_1 . \mathbf{R} \llbracket E_2 \rrbracket r (\dots \\ & \lambda e_{n-1} . \lambda s_{n-1} . \mathbf{R} \llbracket E_n \rrbracket r (\lambda e_f . \lambda s_f . \\ & d \downarrow 2 ((\lambda s_{ret} . \text{let } s_{new} = \text{dealloc-locn } l_s \ l_t \ s_{ret} \text{ in} \\ & k (\text{access } s_{new} (s_{new} \downarrow 1 + s_{new} \downarrow 2) s_{new}), \\ & s_f, (e_1 \dots e_n)))) s_{n-1}) s_1) \end{aligned}$$

The result of elaborating these denotations would be a λ -expression which maps any input file (there being no input in the example program) to the distance between the two points, expressed as an integer.

5 Semantics Based Parallelizing Compilation

In this section we outline the process of generating distributed code for SmallC++ using semantics based compilation techniques. We define a two step process to generate the parallel code. First SmallC++ is translated into a sequential combinator language which provides a sequential interpretation. Next we perform semantics preserving transformations on the combinator representation that replace all potential parallelism by parallel combinators and communication primitives. We use the Tuple Space model of concurrency [3] as the underlying model for object interaction. The result of the transformations is an executable parallel program with well understood semantics.

5.1 Sequential Semantics

A set of sequential combinators is developed using the semantics of SmallC++ defined in Section 4. We have defined a combinator for every major semantic action such as looping, conditionals, arithmetic and boolean operations, memory access and type checking. Figure 4 summarizes the set of sequential combinators we designed to implement SmallC++. Of these combinators *message-send* and *create-object* are directly related to the semantics of objects. An object is encoded as a store-like function that returns the denotations of instance variables and methods for that object. Figure 5 is the denotation of the SmallC++ program introduced in Section 2 after compilation. For simplicity sake we do not change the identifier names into their actual denotations.

<code>access-array, update-array</code>	access and update array elements
<code>access-field, update-field</code>	access and update components of structures
<code>access-member, update-member</code>	access and update instance variables of objects
<code>create-object</code>	create an object
<code>copy-block</code>	copy structures and arrays
<code>assign</code>	define an assignment operation
<code>compose</code>	sequential composition
<code>if</code>	conditional (must have “then” and “else” parts)
<code>for</code>	for-loop (must have an index, initial value, termination condition and loop body)
<code>call</code>	procedure call
<code>message-send</code>	send a message to an object (similar to “call”)
<code>return</code>	return a value
<code>fix</code>	the fix-point combinator, represents recursively defined functions
<code>ref</code>	call-by-reference parameter
<code>deref</code>	access a call-by-value parameter
<code>plus, minus, times, slash, equal, not-equal, less, less-equal, greater, greater-equal, and, or, not</code>	standard arithmetic, relational, and logical operators

Fig. 4. Sequential Combinators

5.2 Parallel Semantics

The second step in defining the semantics is applying a set of semantics preserving parallelizing transformations that encode SmallC++ by a parallel combinator language augmented with communication primitives. This encoding is executable on a distributed platform. The transformation is performed using flow

```

(compose (create-object Point obj1 3 4)
         (create-object Point obj2 5 6) (create-object int d)
         (assign d (call sqrt (plus
                               (call square (minus (message-send obj1 GetX)
                                                    (message-send obj2 GetX)))
                               (call square (minus (message-send obj1 GetY)
                                                    (message-send obj2 GetY))))))
         (write d))

```

Fig. 5. Denotation of the example program in sequential combinators

analysis, dependence analysis, and parallel code generation. We describe the parallel combinators, the Tuple Space model of communication, and the proposed transformations below.

5.3 Parallel Combinators

Parallelism is encoded using parallel combinators. Parallel combinators were introduced by [7] to define the granularity of parallelism, and [8] to generate parallel functional code. Our approach is most like [8] in that the combinators themselves represent the basic instruction set of the parallel machine model being used, in our case the Tuple Space model. Some examples of parallel combinators are given in Figure 6. The set of parallel combinators is still under development. So, only a high-level set is illustrated here.

parallel	argument expressions can be evaluated in parallel
sequence	argument expressions must be evaluated sequentially
distribute	argument objects can be distributed
cluster	argument objects should not be distributed
for-all	parallel for-loop (must have an index, initial value, termination condition and loop body)

Fig. 6. Parallel Combinators

5.4 Tuple Space

We use the *generative communication model of distributed computing* or Tuple Space (TS) for the implementation of interprocess communication. TS represents

communication among distributed processes by tuples. Formally, a tuple is an $n + 1$ -tuple in which the first element of the tuple is a name, and the remaining n elements are parameters, either formal or actual. The actual parameters represent data to be sent and the formal parameters represent place holders. The model is dynamic, with the tuple space of a program changing as the program executes. For example, if A and B are objects, A sending a message to B will generate a new tuple in the space. When B is ready to receive this message it removes the tuple and executes. These actions are encoded by the functions **out**, which puts a new tuple in the space; **in**, which removes a tuple from the space; and **read**, a non-destructive **in**.

TS provides a good interprocess communication model for an OO environment for several reasons. First, tuples, like objects, can have both data members and functions. Thus they provide a natural representation of objects. Second, a tuple can consist of a name and a function, facilitating the creation of a process which can execute in parallel. We call such tuples *message tuples* and they can be encoded using the **eval** operator that was introduced in [12]. Message tuples can return a value in the TS, that can be used by other processes, at the end of its execution. Thus, a method and data member can be treated equally in our implementation. Finally, using **eval**, a process may request the execution of a program (a method, for example) on a specific node or allow the system to select the node within the TS network which has the least number of active processes thus providing an automatic load balancing mechanism.

5.5 Generating the Parallel Semantics and Code

The basic structure of a parallelizing compiler for an object-oriented programming language is similar to that for more conventional languages [21]. However, there are some major differences in the nature of the components, caused by object-orientation. Apart from the regular loop-parallelism that is available in Fortran-like languages, there are at least three other forms of concurrency possible within an object-oriented system. Inter-object concurrency refers to different objects carrying out different activities at the same time. Intra-object concurrency refers to a single object executing several methods simultaneously. Thirdly each of these methods could themselves be carrying out several operations in parallel. Thus, an object may have several threads of control, each corresponding to different types of concurrent execution.

The primary components of the parallelization of OO programs are flow analysis, dependence analysis, and parallel code generation. A general overview is presented in Figure 7.

Flow analysis creates a graphical representation of the flow of data among components of a program. Unlike the flow analysis performed on non-OO programs where nodes represent simple statements, the flow graph we create has three types of nodes: (1) individual methods, (2) a collection of methods within a class or object, and (3) all the methods of an OO program. The primary complexity introduced by the OO paradigm is determining the precise instance

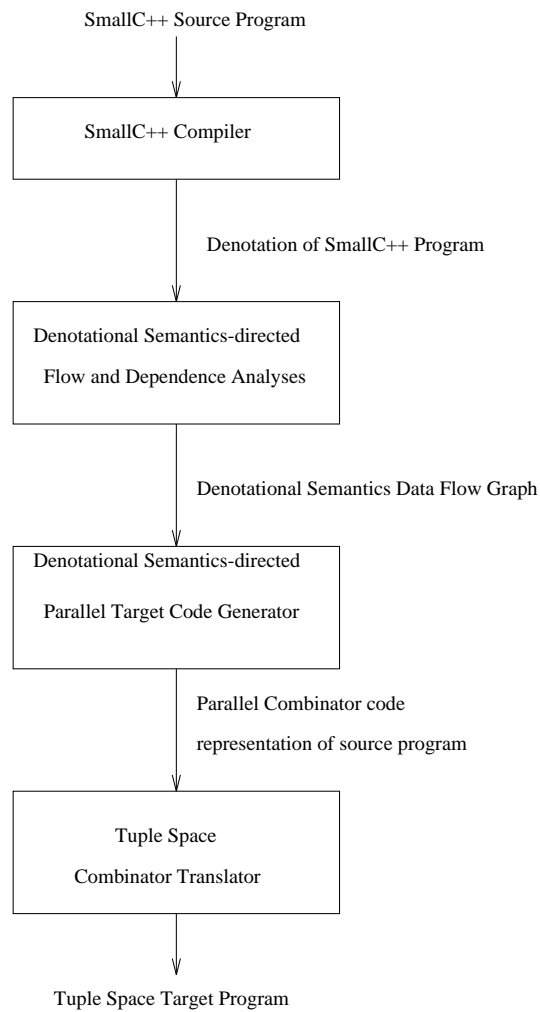


Fig. 7. Overview of Denotational Semantics-Directed Parallelization

variables and methods being referenced in the presence of polymorphism and inheritance.

Dependence analysis uses the flow graph to determine the data dependencies and the interconnectedness among the objects in the program. Because we are working with an OO programming paradigm, we must consider both the dependencies within methods and the dependencies that result from message passing. The first type is handled by constructing a standard dependence graph for the code within each method. We determine the interconnections and flow of data among objects using a second type of dependence graph we call a *message flow graph*.

Parallel code generation uses the flow and dependence information to partition the objects into processes in a manner that reduces communication costs. This includes determining whether an object process should be executed synchronously or asynchronously. Furthermore, we may generate code that automatically controls communication between processes by increasing the amount of data transferred in a single message so as to reduce the number of messages sent. This includes, for example, the restructuring of for-loops containing a message updating a database to a single message containing all of the updates. The result of the parallel code generation is a parallel combinator program which may be executed using Tuple Space.

The sequential combinator code produced for the example program in Section 5.1 is now transformed to the program shown in Figure 8 after performing flow and data dependence analysis. This program contains the parallel combinators identifying parts of the code that can be executed in parallel. The creation of the two point objects (and the distance variable) are done in parallel. Various portions of the calculation of the distance between the two points are also done in parallel.

The tuple space code corresponding to the parallel combinator code is shown in Figure 9. The Tuple Space representation `outs` two tuples, one for each `Point` object. The names `"Pointobj1"` and `"Pointobj2"` are keys that are used to identify the tuples, created by concatenating the Class Name and the Object Name (we would wish to use a more unique key creation algorithm in practice). The first `eval` forks four processes each executing one method and returning a value in the temporary variables. The second `eval` forks two more processes, each using the previously computed temporary values. The final `eval` forks a process which computes the distance and returns it in `d`. In essence, the computation in the above program is done in just three steps - one step for each `eval`, ignoring the `outs` in the first two statements.

6 Summary

We have implemented the sequential semantics generating sequential combinator code from SmallC++ programs. We are currently implementing the program dependence graphs from the sequential combinator code. We intend to investigate the graphs to discover more parallel combinators that could be used in generat-

```

(compose
  (parallel (create-object Point obj1 3 4)
            (create-object Point obj2 5 6) (create-object int d))
  (assign d
    (call sqrt
      (plus
        (parallel
          (call square (minus
            (parallel
              (message-send obj1 GetX)
              (message-send obj2 GetX))))
          (call square (minus
            (parallel
              (message-send obj1 GetY)
              (message-send obj2 GetY))))))))))
  (write d))

```

Fig. 8. Denotation of the example program in parallel combinators

```

out ("Pointobj1", 3, 4); out ("Pointobj2", 5, 6);
eval ("dtmp1", tmp1 = GetX (Pointobj1), tmp2 = GetX (Pointobj2),
      tmp3 = GetY (Pointobj1), tmp4 = GetY (Pointobj2));
eval("dtmp2", tmp5 = square(tmp1 - tmp2), tmp6 = square(tmp3 - tmp4));
eval("d", d = sqrt (tmp5 + tmp6));

```

Fig. 9. Tuple Space code for example program

ing the parallel combinator code. This would be followed by an implementation of the mapping from the parallel combinator code to the tuple space code which would be run on a network of sun workstations. Our future research goal is to extend SmallC++ and provide a complete semantics specification for a larger subset of C++. Also once the tool is developed, we are planning to apply this technique to SmallTalk.

References

1. Aho, A.V., Sethi, R., Ullman, J.D., *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.
2. Bik, A. J. C., Gannon, D. B., "Automatically Exploiting Implicit Parallelism in Java," in *Concurrency: Practice and Experience*, vol. 9, no. 6, 1997, pp. 579-619.
3. Gelernter, D., "Generative Communication in Linda," in *ACM Transactions On Programming Languages and Systems*, vol. 7, no. 1, Jan. 1985, pp. 81-112.

4. Genjiang, Z., Li, X., Zhongxiu, S., "A Path-Based Method of Parallelizing C++ Programs," in *SIGPLAN Notices*, vol. 29. no. 2, Feb 1994, pp. 19-24.
5. Goldberg, A.J., Robson, A.D., *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.
6. Hindley, J., R., Seldind, J., P., *Introduction to Combinators and λ -calculus*, Cambridge University Press, 1986.
7. Hudak, P., Goldberg, B., "Distributed Execution of Functional Programs using Serial Combinators," in *IEEE Transactions on Computers*, vol. C-34, no. 10, Oct. 1985, pp. 881-891.
8. Knox, D.L., Wright, C.T., "Combinators as Control Mechanisms in Multiprocessing Systems," in *Proceedings of the International Conference on Parallel Processing*, 1987, pp. 158-161.
9. Lee, P., *Realistic Compiler Generation*, MIT Press, Cambridge, MA, 1989.
10. Milner, R., Tofte, M., and Harper, R., *The Definition of Standard ML*, MIT Press, Cambridge, MA, 1990.
11. Muthukrishnan, P.K, Bryant, B.R, "The Syntax and Semantics of SmallC++," *Technical Report, Department of Computer and Information Sciences, University of Alabama at Birmingham*, 1995.
12. Patterson, L., *Fault Tolerant Tuple Space*, Ph.D. thesis, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL, 1992.
13. Peyton Jones, S., *The Implementation of Functional Programming Languages*, Prentice Hall, Englewood Cliffs, NJ, 1987.
14. Schmidt, D.A., *Denotational Semantics A Methodology for Language Development*, Allyn and Bacon, Inc., Boston, MA, 1986.
15. Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.
16. Tokoro, M., Nierstrasz, O., Wegner, P., eds., *Object-Based Concurrent Computing, Proceedings of ECOOP '91 Workshop*, Springer-Verlag, 1991.
17. Tomlinson, C., Scheeval, M., "Concurrent Object-Oriented Programming Languages," in *Object-Oriented Concepts, Databases, and Applications*, ACM Press/Addison-Wesley, Reading, MA, 1989, pp. 79-124.
18. Wegner, P., "The Object-Oriented Classification," in *Research Directions in Object-Oriented Programming*, MIT Press, Cambridge, MA, 1987, pp. 479-560.
19. Yin, M., Bic, L., Ungerer, T., "Parallel C++ Programming on the Intel iPSC/2 Hypercube," in *Proceedings of the 4th Annual Parallel Processing Symposium*, 1990, pp. 380-394.
20. Yonezawa, A., Tokoro, M., eds., *Object-Oriented Concurrent Programming*, MIT Press, Cambridge, MA, 1987.
21. Zima, H., Chapman, B., *Supercompilers for Parallel and Vector Computers*, ACM Press/Addison-Wesley, Reading, MA, 1990.