Generating an Efficient Compiler for a Data Parallel Language from a Denotational Specification

Johan Ringström, Peter Fritzson, Mikael Pettersson

Email: {johri,petfr,mikpe}@ida.liu.se
Phone: +46 13 281000, Fax: +46 13 282666

Abstract. There are very few examples of the generation of efficient compilers from denotational specifications. Usually such compilers generate code which is orders of magnitude slower than from hand-written ones. However, as has been demonstrated by our DML (Denotational Meta Language) compiler generation system, through appropriate single-threading analysis it is possible to obtain code of comparable quality to hand-written compilers. Another problem with denotational specifications is, because of their denotational nature, the need to introduce complicated power domains to model non-determinism and parallelism. In this work we have used a more practical two-level approach: use denotational specifications to model the meaning of the source language in terms of an abstract machine of low-level operations, including data-parallel operations. Then use operational semantics for the specification of this abstract machine.

This paper reports experience from building a prototype compiler for a small Algol-like parallel language using a version of the DML system called DML-P, which has been extended to support data-parallel operations. The final code contains calls to a portable data-parallel vector code library (VCODE CVL). The speed of generated compilers are within a factor of three from handwritten ones. Extensive benchmarks of a DML-P generated compiler are presented.

1 Introduction

The high abstraction level of denotational semantics makes it attractive as a language specification formalism. However, many efforts to generate compilers from denotational semantics specifications, starting with the SIS system by Peter Mosses 1979 [12], have resulted in compilers and code that run very slowly - often 100 to 1000 times slower compared to commercial compilers, and that also do not interface to commercial product-quality parsers or code generators. The situation has gradually improved through the work of several researchers, e.g. Sethi [25], Paulson [15], Raskovsky [20], Wand [30], Appel[1], Jouvelot[9], and later work by Mosses [13], until Lee [11] and Petterson [16] demonstrated the first practical compiler generation systems accepting denotational specifications.

In comparison with the MESS [11] system, the DML system goes several steps further. It interfaces well with standard tools and it automatically generates a code generator that emits intermediate quadruple code. DML can handle denotational specifications in continuation-passing style, which is well suited for specifying arbitrary control structures. The high code quality obtained using DML-generated compilers for Algol-like languages is possible due to escape-analysis of the intermediate representation, which removes all closures that would otherwise be present at run-time.

However, languages with non-determinism or parallelism are awkward and complicated to specify using denotational semantics since power domains have to be introduced. In order to cope with such languages in a practical way, we have instead turned to a two-level approach: use a denotational specification to model the meaning of the source language in terms of an abstract machine of low-level operations including data-parallel operations, then use operational semantics to specify the abstract machine. In this paper we report some experience of using this approach, including benchmarks obtained from generated compilers. The DML-P (P for parallel) compiler generator system provides a prototyping environment for the design of efficient compilers for data parallel computers. This system has been developed from an early version of the original DML-S (S for Sequential) system. This system implements the SML core language plus structures and signatures. It also includes constructs that make the implementation of a denotational specification easier. The development is done by adding support for data parallel operations as well as a more complete set of primitive operators and types in the target intermediate language.

First we briefly describe data parallel languages using a Predula Nouveau example, which is the subject language in this report. Then follows a brief introduction to denotational semantics including some excerpts of the Predula Nouveau denotational specification, after which we discuss two target data-parallel abstract machines. We continue by giving a description of the implementation of the DML-P compiler generator and the CPS-graph module. This graph is the internal representation for compiled programs. Finally we present benchmarks for generated compilers followed by conclusions and future work.

2 Data Parallel Languages

The data-parallel programming paradigm [6] has become popular in recent years since it exploits massive parallelism while preserving a simple deterministic programming style. Many algorithms can be expressed as operations on collections of similar data objects. Therefore languages containing data parallel programming constructs are called *collection-oriented languages* [2]. In this paper we use a small data-parallel Algol-like language called Predula Nouveau (a successor to Predula, PaRallel EDUcational LAnguage [21]) as a test case in our work on compiler generation for data parallel languages. This language includes a number of constructs operating on arrays, allowing the utilization of parallelism when operating in parallel on elements of such arrays.

Figure 1 shows a small Predula Nouveau program that calculates the inner product. The

```
main() is
    a, b : array 0..5 of integer;
    c : integer;
 begin
    for i in 0..5 loop
        a(i) := i;      (* a := (0, 1, 2, 3, 4, 5) *)
        b(i) := 5-i;    (* b := (5, 4, 3, 2, 1, 0) *)
    end;
    (* First perform elementwise multiplication. Then add the
       products together into the c variable. *)
    c := reduce(op +, each(op *, a, b));
    write(c,"\n");
end;
```

Fig. 1. Inner product - a small Predula Nouveau example of data-parallelism.

arrays a and b contain source vectors which are used when calculating the inner product. This is done by applying the `reduce` function on the elementwise products of the elements in the two arrays. This intermediate array is calculated by the `each` function using the multiplication operator. The `each` and `reduce` operators are data parallel in that they apply their operator argument to all array elements at the same time.

2.1 Data-Parallel Operators

One of the earliest programming languages which included collection oriented programming capabilities is APL [8], which is centered around a vector-like data type. Numerous modern data parallel languages have followed this style of a collection data type together with a set of associated operations. Predula Nouveau includes a parallel version of the array data type found in the Algol family of languages. The *reduce*, *scan*, *each*, and *enumerate* operators are used in Predula Nouveau.

For example, the *reduce* operator takes an operator and an array as arguments. The intuitive meaning is to put the argument operator between the array elements and then to evaluate the resulting expression. This operator must be associative in order not to give a result that depends on the topology of the evaluation tree. It also must have an identity element. This value is returned if an empty array is supplied to *reduce*. Denotational definitions of *reduce* and some other data-parallel operators like *scan*, *each*, and *enumerate* can be found in [22].

3 Denotational Semantics

In a denotational specification, the meaning of a program is a mapping from an abstract syntax to denotations. Domains that form the source of this mapping are called *syntactic domains*. Domains that form the denotations are called *semantic domains*. For a detailed discussion of the foundations of this area, see [26,28]. A more practically oriented introduction is found in [4].

We want to describe a computer language including the operational behaviour of parallel programs while avoiding unnecessary detail, but we also want to be able to automatically generate a compiler for the language. Our approach is to describe the static semantics and part of the dynamic semantics of a programming language using a denotational semantics and to describe the remaining dynamic semantics using an operational semantics. These two parts are from now on named *high-level* and *low-level* semantics. We name the operational part low-level, since it is closer to the target machine. The most well-known example of a language specified by a denotational specification is Scheme [19]. Another example is OCCAM, which uses a combination of a denotational specification for the sequential part of the language and a CSP-based method for the parallel part [7].

3.1 Compiler Generation from Denotational Semantics

Figure 2 illustrates the compiler generation process. The specification of the low level intermediate language could be formally defined using some operational semantics formalism, see e.g. [18]. However, here we adopt a more pragmatic approach by mapping the intermediate representation directly to quadruples which are emitted in the form of calls to the VCODE [3] library together with simple C statements.

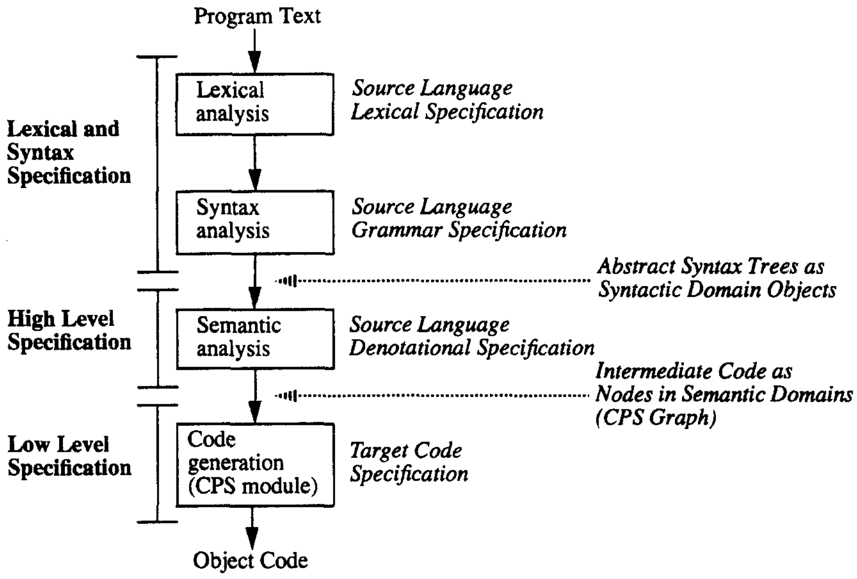

Program Text
Lexical and Syntax Specification
Lexical analysis
Source Language Lexical Specification
Syntax analysis
Source Language Grammar Specification
Abstract Syntax Trees as Syntactic Domain Objects
High Level Specification
Semantic analysis
Source Language Denotational Specification
Intermediate Code as Nodes in Semantic Domains (CPS Graph)
Low Level Specification
Code generation (CPS module)
Target Code Specification
Object Code

Fig. 2. The translation of a source program to object code by a compiler generated by the DML system. The kinds of specification formalism are shown for the modules.

The DML system encapsulates the mapping to target code in a *Continuation Passing Style (CPS) graph*. The nodes in this graph are lambda calculus expressions. By the way the continuation is constructed, lambda abstractions can be stack allocated [18]. This makes it easier for the backend to generate efficient code.

When generating a compiler, e.g. for a tiny subset of C, the original CPS-graph [16] module from the DML-S system can be used, but for Predula Nouveau it is necessary to add extensions for the parallel part of the language, as will be shown later.

3.2 A Comment on the Denotational Specification Notation

Most work on denotational semantics use the Oxford School notation. A very concise notation using one letter identifiers in the greek alphabet is usually used. This is nice for articles and short examples, but not practical for large machine-processable language specifications. Therefore we have adopted Standard ML [5] syntax. Examples of transformations between Oxford School notation and SML are given below.

Oxford School notation	**SML notation**	
ρ_1	`env1`	*(variable ref.)*
$\varepsilon_3 \rightarrow \varepsilon_1 \mid \varepsilon_2$	`if pred then exp1 else exp2`	*(conditional)*
$\chi\,\rho$	`dcont env`	*(application)*
$\lambda\alpha.\alpha$	`fn x => x`	*(lambda)*
(α,β)	(a,b)	*(tuple formation)*
$\langle\alpha,\beta,\gamma\rangle$	`[a,b,c]`	*(sequences)*
let $x \equiv y$ *in* z	`let val x = y in z end`	*(local binding)*
$1 \in Int$	*(expressed by SML pattern matching)*	*(membership check)*
1 *in* C	*(expressed by SML constructors)*	*(injection)*
$\alpha \mid D$	*(expressed by SML pattern matching)*	*(projection)*

4 The Predula Nouveau Specification

Syntax and semantics are specified using different methods. Abstract syntax is described using syntactic domains. Semantics is expressed using a continuation passing style denotational semantics. This part of the specification is built using CPS primitives which are expressed in a CPS back end module.

4.1 Syntax

The concrete syntax is implemented as a scanner and a parser generated by ML-Lex and ML-YACC. The parser builds elements in syntactic domains from an input program. These domains correspond to classes of abstract-syntax objects. There are five flat syntactic domains for literals and identifiers.

$L_n \in$ Nml	(integer numerals)	$L_r \in$ Real	(reals)
$L_c \in$ Char	(characters)	$L_s \in$ Str	(strings)
$I \in$ Ide	(identifiers)		

Using these definitions it is now possible to build non-flat domains:

$D \in$ Decl	(declarations)
$T \in$ TExpr	(types)
$F \in$ Param	(item in function prototype parameter list)
$E \in$ Expr	(expressions)
$C \in$ Cmd	(statements)

At the top level a Predula Nouveau program is a *D* value. Abstract syntax are equations on syntactic domains. Let us define some equations:

$$
\begin{aligned}
D &::= \text{null} \mid \text{type } I\,T \mid \text{var } I\,T \mid \text{vari } I\,T\,E \mid \text{varic } I\,T\,E \mid \text{fun } I\,P\,T\,C \mid D_1\,D_2 \\
P &::= \text{var } I\,T\,M \mid \text{varc } I\,T\,M \mid P^* \\
M &::= \text{no} \mid \text{var} \\
T &::= \text{tvar } I \mid \text{arr } T_1\,T_2 \mid T^* \\
E &::= L_c \mid L_n \mid L_r \mid L_s \mid I \mid \text{un } E_1\,E_2 \mid \text{bin } E_1\,E_2\,E_3 \mid \text{app } E \mid E^* \\
C &::= \text{skip} \mid \text{loopi } C \mid \text{loopd } D\,E\,C \mid \text{break} \mid \text{cont} \mid \text{return } E \\
&\quad\mid \text{decl } E \mid \text{expr } E \mid \text{cond } E\,G \mid C_1\,C_2 \\
G &::= E\,C \mid G^*
\end{aligned}
$$

Variable declarations have three summands: uninitialized, initialized, and constant initialized declarations. Arrays are composed of two types. The first type is the index type and the second is the element type. Definite loops are composed of an induction-variable declaration (including the loop range), a count step expression, and the loop body.

4.2 Semantics

In the semantics specification function domains are used in addition to sum, product, and sequence domains that build the syntactic domains. We concentrate on the part of the predefined environment that specifies data-parallel operators.

The *DVal* domain contains values that can be denoted in a Predula Nouveau program. Predefined value and type environments are defined by the following domain equations:

$$VEnv \equiv Ide \rightarrow DVal \times Type \qquad TEnv \equiv Ide \rightarrow Type \qquad Env \equiv VEnv \times TEnv$$

Denotable values are bound to names in *VEnv* and denotable types are bound to names in *TEnv*. This defines two separate name spaces for Predula value and type identifiers in a natural way. For the pairs in *VEnv* the *Type* domain component is the type of the denotable value. The following function builds *TEnv* values when partially applied:

mktyp (ide,t) tenv ide2 ≡ if ide=ide2 then t else tenv ide2

For example, *mktyp* ([["integer"]], *IntType*) produces a value in the *TEnv* → *TEnv* domain. Larger environment segments are built by function composition on such values. Compare this to symbol tables in traditional compilers.

4.2.1 Definition of the Data-Parallel Functions in Predula Nouveau

We are now ready to define the Predula Nouveau data-parallel operators in the predefined environment. In order to do this we use the *mkopr* function in analogy to the *mktyp* function above to build the value environment. One important observation is that the parallel operational semantics does not have to be specified anywhere in the denotational semantics. The second argument position of *mkopr* in the definition below contains a type checking function for the operator. The third and fourth positions define parameter and result modes. The last position maps to operators in the CPS graph. The Predula Nouveau `scan`, `reduce`, and `each` functions are overloaded: they come in integer, real, and boolean versions. The integer versions, including `enumerate`, are shown below:

```
iparoprs ≡ mkopr ([["enumerate"]], enumtc    intT, I_ENUM) o
           mkopr ([["each"]],      eachtc 1  intT, I_EACH1) o
           mkopr ([["each"]],      eachtc 2  intT, I_EACH2) o
           mkopr ([["scan"]],      scandtc   intT, I_SCAND) o
           mkopr ([["scan"]],      scansdtc  intT, I_SCANSD) o
           mkopr ([["scan"]],      scanstc   intT, I_SCANS) o
           mkopr ([["reduce"]],    reduceutc intT, I_REDUCE) o
           mkopr ([["reduce"]],    reducestc intT, I_REDUCES)
```

For a definition of the binding of real and boolean data parallel functions (together a total of 22) and a more detailed discussion of the type checking functions that are used in the composition, see [22].

4.3 Data-Parallel Extensions in Predula Nouveau

Most of the extensions are introduced to support data parallelism. These extensions consist of the addition of a number of data-parallel functions, like *enum, scan, reduce,* and *each,* that work on the array type.

Currently only one-dimensional arrays of scalar or real values are possible. Still, we feel that this set, which however probably is too small to be used in real-life data-parallel programming, is large enough to demonstrate the feasibility of compiler generation for data-parallel programming languages from denotational specifications.

5 Data Parallel Target Codes

5.1 VCODE and Paris

The VCODE system [3] implements an abstract machine of data-parallel vector operations. It is portable between several parallel architectures including the CRAY and CM-2, but also sequential architectures, such as the Sun workstation. The VCODE operations are implemented on top of a library of C routines, the C Vector Library (CVL).VCODE is a single assignment stack-based language where all instructions operate on arrays. There are no free scalars, not even on the stack. The data-parallel functions are limited to arrays of scalars and floating-point numbers: it is not possible to directly build nested arrays or multidimensional arrays. Instead the VCODE data-parallel functions can handle segmented arrays.

Both VCODE and the CM-2 Paris library [29] are cumbersome to use due to the many implementation details that need to be incorporated in such programs. This applies especially to Paris. VCODE, on the other hand, is limited due to its single-assignment vector-stack, i.e. vectors on the stack cannot be destructively updated. There also is no way to store scalars except as unit vectors, and there are no global or heap stores to complement the stack. Also, to manage multidimensional or nested arrays it is necessary to keep the array topology in a separate segment array. Array decomposition for massively-parallel computers is an active research area: see for example [10].

6 Implementation of the Compiler Generator

The SML/NJ system is used to generate implementations from denotational semantics specifications. Specifications can almost be transformed verbatim to SML/NJ: very few transformations are necessary.

6.1 The module design

The specification is partitioned into three groups of modules, as is shown in Figure 3. The figure also shows the generation of a compiler from specification modules. The language specification part contains language specific modules. The general functions part consists of a module that contains definitions which are independent of both the specified language and the target platform. This module includes the definition of some auxiliary functions in the denotational definition. The lower right part contains implementation dependent specification modules.

All these modules together generate the lower left part in Figure 3 which shows the modules in the generated compiler. The generated `Front` module transforms the input program to an abstract syntax tree. `Mid` transforms the abstract syntax tree to a CPS graph through the *M* semantic function. `Back` transforms the graph to target code (in our case ANSI C quadruples and calls to the CVL library). Currently the SML/NJ module system is used to implement the module composition.

We claim that this way of building the system offers two important benefits. First, the division between the denotational part and the operational part of the language specification is expressed in a natural and flexible way that makes automatic generation of a compiler

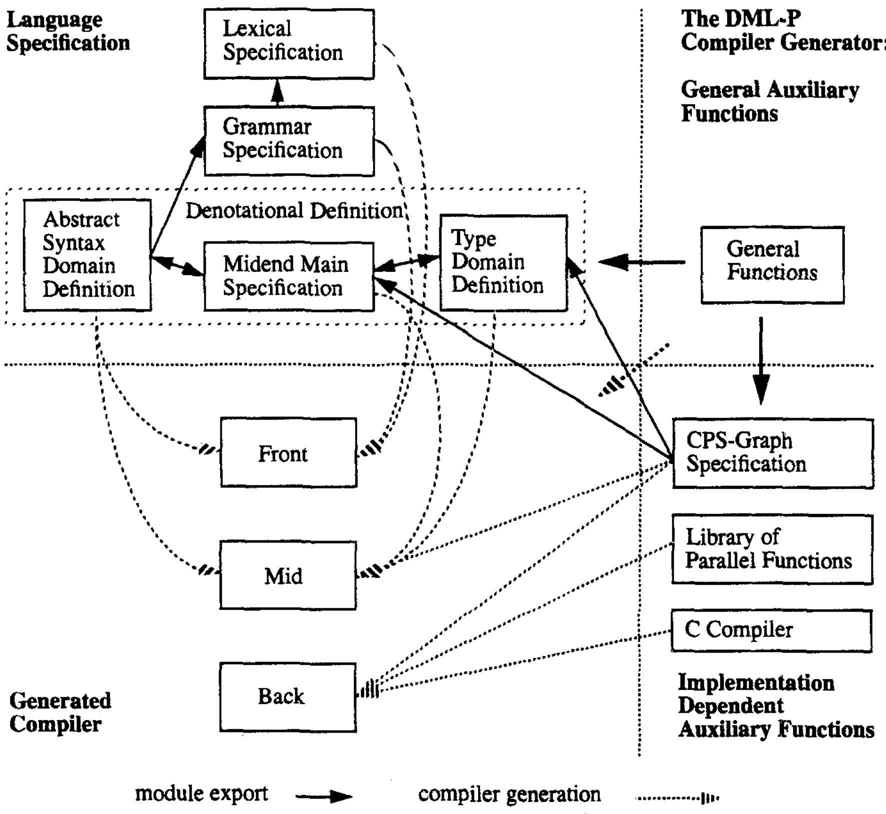


Fig. 3. The module structure of typical language specifications, the DML-P compiler generator, and the generated compiler.

practical. Second, the strong modularisation makes it possible to reuse parts of the specification between different languages. Currently, only the CPS-graph module has been used for more than one language since it also has been used to specify the Tiny-C language in the DML-S system.

6.2 The Predula Nouveau Language Specification

The Predula Nouveau abstract syntax domain definition, midend main specification, and type domain definition (the modules in the dashed box in Figure 3) are written in denotational semantics. The abstract syntax definition is used by both the grammar specification and by the midend. The type domain definitions module contains denotational definitions of the type system. The midend main specification module contains most of the Predula Nouveau semantics including the *M* (as in meaning) semantic function.

6.3 The CPS-Graph Specification

The CPS-graph specification module generates part of the `Back` part of the generated compiler in Figure 3. The module interface includes a number of constructors which builds

```
datatype Opr = ...
          | I_ENUM
          | R_EACH1  | R_EACH2   | R_SCAN  | R_SCANS  | R_SCANSD
          | R_REDUCE | R_REDUCES

val Lambda : Typ * (Val -> Cont) -> Kont

val Halt : unit -> Cont                                 (* Terminate *)
val Return : Val -> Cont                      (* Return from function *)
val Call : Func * Val list * Kont -> Cont
val Cond : Val * Cont * Cont -> Cont
val New : bool * Typ * Kont -> Cont             (* Build a new location *)
val Send : Val * Kont -> Cont             (* Send value to command cont *)
val Update : Val * Val * Cont -> Cont               (* Update location *)
val Fetch : Val * Kont -> Cont            (* Fetch value in a location *)
val VIndex : Val * Val list * Val list * Kont -> Cont (* Get arr loc *)
val PrimFunc : Opr -> Func          (* Func from primitive operator *)

val fix_cont : (Cont -> Cont) -> Cont
val fix_func : Typ * bool list * (Val list -> Func -> Cont) -> Func

val codgen : Cont -> unit
```

Fig. 4. Declaration of Operators and Basic Term Building Blocks of the CPS module of DML-P

a CPS lambda expression. The most important constructors are shown in Figure 4. Inside the module these constructors build a graph. The CPS graph consists of three kinds of nodes. `Cont` nodes are lambda expressions that do not bind a value to a new name. `Kont` nodes are used to bind values to new names. The third kind of node, named `Func` in the figure, is functions. These functions can be either primitive or user-defined. Primitive functions are created by the `PrimFunc` constructor from primitive operators. User-defined functions can only be defined via the `fix_func` constructor. This constructor also binds function parameters.

The module also exports the `codgen` function which generates object code from the CPS lambda expression. Presently, this code is an ANSI-C program that together with the VCODE CVL is sent to the C compiler for compilation to object code. Section 7 gives a more detailed discussion of the code generation module concentrating on the data-parallel extensions compared to the CPS-graph module in the DML-S system. A discussion of the technical detail of the interface and implementation of the module is also found in Appendix A of [18].

6.4 Using the CPS Module Constructors

Let us illustrate using the `I_ENUM` integer data-parallel enumeration operator. Assume we want to apply this operator to its arguments. This is done using the `Call` constructor:

```
Call(PrimFunc I_ENUM,args,kont)
```

The first argument to `Call` is the function to be applied. This is the `I_ENUM` operator injected into the `Func` domain. The second argument `arg` is a sequence of evaluated left or right value arguments to `Call`. The last argument is the expression continuation that receives the result of the application. For a user-defined function, the call constructor instead uses a value in the `Func` domain defined by the `fix_func` constructor.

The `fix_cont` constructor is used to express fixpoints of loops. The `fix_cont` and `fix_func` constructors are the only ones that can be used to express recursion. The `Fetch` constructor fetches an atomic value from a location. The `VIndex` operation sends a location in an array to an expression continuation. Combining these two it is possible to fetch an element from an array location:

```
    VIndex(arr,[offset],[index],Lambda(x,Fetch(x,kont)))
```

`VIndex` sends its location result to `Fetch` using a `Lambda` binding. Work is ongoing to generalise `VIndex` to multidimensional and nested arrays.

7 The New Code Generation Module for Data-Parallel Operations

Here we shortly talk about the internals of the code generation module and the modifications that were necessary to make to the original DML-S system CPS module able to generate data-parallel code [18].

First, new primitive operators have been added. The original module contained 11 operators. These operators were created with the goal of compiling a small subset of the C language. The extended module in DML-P contains a total of 83 operators. This larger amount is partly due to the increased number of types that is supported by the module, but also because the operator domain includes 22 new data-parallel operators, counting different types. The truncated definition of the operator semantic domain in Figure 4 includes some of these data-parallel operators.

Second, the new module can handle a greater range of types than the original module. This includes characters, strings, and floating point numbers. This also made it necessary to add new constructors for literals.

Third, the data-parallel extensions make it necessary to implement arrays more carefully. Currently only fixed size arrays are supported, but work is ongoing to support arrays with dynamically determined size.

Fourth, the most important extension is the addition of the VCODE CVL. This is done as external C calls to the library operations. This library is linked with the generated object code.

From a code generation point of view, the CPS-graph module linearizes the graph into a sequence of instructions. In the original module this is done in two phases. First the graph nodes are marked. This includes counting of references. If a `Cont` node is referenced by

```
#include <stdio.h>
#include <math.h>
#include "datapar.h"

typedef struct{long body[6];
} ttl;

void t1(){
long t23;
long t29;
long t32;
long  *t31;
long  *t34;
char t27;
long  *t18;
long t38;
long  *t17;
long t39;
long t41;
long  *t40;
long t42;
ttl  *t16;
ttl t43;
long t45;
long  *t44;
long t46;
ttl  *t11;
ttl t47;
long t49;
long  *t48;
long t50;
ttl  *t6;
ttl t51;
t6=&t51;
t48=&t50;
 *(t48)=0;
t49=6;
{long t52;
for(t52=0;t52<t49;t52++){
t6->body[t52]= *t48;
}}
t11=&t47;
t44=&t46;
 *(t44)=0;
t45=6;
{long t53;
for(t53=0;t53<t45;t53++){
t11->body[t53]= *t44;
}}
t16=&t43;
t40=&t42;
 *(t40)=0;
t41=6;
{long t54;
for(t54=0;t54<t41;t54++){
t16->body[t54]= *t40;
}}
t17=&t39;
 *(t17)=0;
t18=&t38;
 *(t18)=0;
L36:t27= *t18<=5;
if(!t27) goto L35;
t34=t6->body+( *t18-0)*1;
 *(t34)= *(t18);
t31=t11->body+( *t18-0)*1;
t32=5- *t18;
 *(t31)=t32;
t29= *t18+1;
 *(t18)=t29;
goto L36;
L35:(each_i_mul(t16,6,t6,t11),t16);
t23=reduce_i_add(t16,6);
 *(t17)=t23;
printf("%ld", *t17);
printf("%s","\n");
return ;
}

int main(){
(void)t1();
exit(0);
}
```

8 Benchmarks of a DML-S Generated Compiler for a Sequential C Subset

In order to make a rough comparison between the quality of stand-alone C-based compilers generated by the original DML-S system and the quality of commercial compilers, we prepared a 1000-line program example in a small C subset called Tiny-C, for which a compiler was generated. This program contains function calls, integer and integer array arithmetic in addition to control structures such as if-statements, loops, etc. This example is measured on a Sparcstation ELC workstation (rated at approximately 20 MIPS). Measured time is elapsed time. Code size is measured on the linked executable file. The generated compiler frontend is in C, and produces low-level quadruples expressed in C syntax, which are fed through the standard SunOS 4.1 CC backend.

The generated compiler was approximately three times slower in compilation speed than the Sun C++ compiler (version 2.1) (the generated compiler processing 1954 lines/minute versus 7100 lines/minute for Sun C++ when generating un-optimized code, or 1036 lines/ minute versus 2600 lines/minute when generating optimized code). However, the execution speed of the generated code was approximately the same, 7.8 seconds for the optimized code from the generated compiler versus 7.4 seconds when executing the optimized code from Sun's C++ compiler. However, when using the Gnu C compiler as the backend instead of the Sun C compiler, the Tiny-C compiler generated code which was about 2.5 time slower than code from the Gnu C compiler (3.4 seconds versus 1.4 seconds). Apparently the structure of the intermediate code from the Tiny-C compiler precludes some optimizations.

	Without back-end optimization			**With optimization (-O2)**		
	Code size (kb)	Compila-tion time (seconds)	Execution time (seconds)	Code size (kb)	Compila-tion time (seconds)	Execution time (seconds)
Gen. Tiny-C compiler with Sun CC backend	40 kb	30.7	30.9	24 kb	57.9	7.8
Sun C++ only	24 kb	8.4	26.5	16 kb	23.1	7.4
Gen Tiny-C compiler + Gnu C	40 kb	33.6	8.7	16 kb	33.5	3.4
Gnu C compiler only	24 kb	8.9	4.6	16 kb	13.4	1.4

9 Benchmarks of a DML-P Generated Compiler for Predula Nouveau

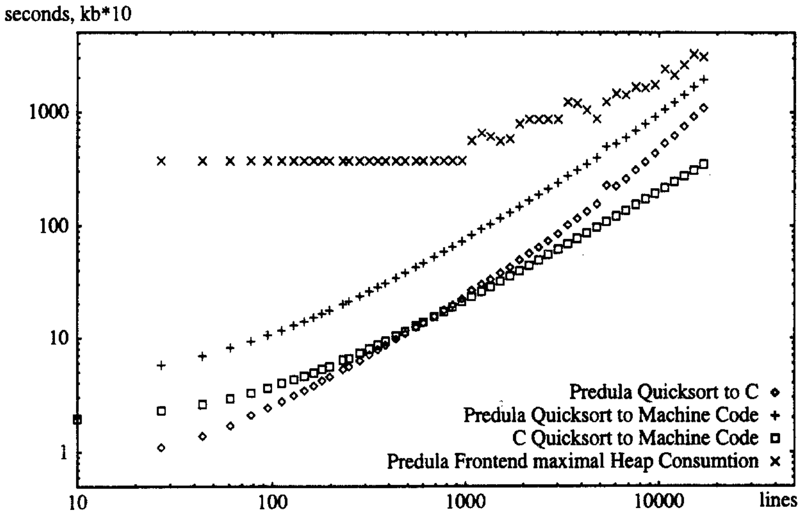
The original DML compiler generation system is implemented in Scheme. It produces compilers in Scheme which are translated to C using Bartlett's Scheme-to-C compiler. Thus stand alone compilers in C, accepting source programs and producing quadruples, are obtained.

The DML-P system has been developed from the original DML system by porting the DML backend to SML/NJ, and by extending the set of low-level operations in the target language to include data-parallel vector operations and scalar operations for a larger range of primitive types (integers, floating point numbers, strings, characters), and arrays built from the mentioned scalar or real types. DML-S only included support for integers and arrays of integers.

The generated compiler frontend which is implemented in SML/NJ, accepts Predula Nouveau source text, and produces low level quadruple code in C, which is then further compiled and linked using the GCC (version 2.4.5) compiler and linker.

9.1 Compiling performance

In the first benchmark, the generated compiler is compared to GCC for speed and maximal heap size on a Quicksort program. The recursive sorting function in Quicksort is unrolled between 0 and 1000 times. 52 logarithmically distributed samples are taken from this range resulting in program sizes varying between 10 and 17000 lines of Predula code. Data for the benchmark was produced on a Sparc1 IPC workstation with 36Mb of primary memory. A similar Quicksort program implemented in C is also measured in this way. Measured time is user+system time.



The maximal heap size for the Predula frontend during compilation is about 3.7 Mb up to about 1000 lines of source code, where the size starts to grow stepwise up to about 30 Mb. The Predula frontend alone is about as fast on smaller Predula Quicksort programs as the Gnu compiler is on smaller C Quicksort programs, but the frontend is slower on larger programs. Considering the entire Predula compiler, most of the time is spent in GCC. This is due to the voluminous code that is currently generated by the Predula frontend.

	Code size (kb)	Compilation time (seconds)	Execution time (seconds)
Predula Nouveau frontend only	---	18.3	---
Predula Nouveau frontend with Gnu C -O2 backend	24	94	22.4
Hand-written C program compiled with Gnu C -O2	24	39	11.8

9.2 Run time performance

The second benchmark compares a 1000 line unrolled Predula Quicksort program to a 1000 line unrolled C Quicksort program. Both programs initialize and sort a 1000 element array 1000 times. The table is measured on a Sparcstation ELC. Measured time is elapsed time and code size is measured on the linked executable file. Compilation speed is about 640 lines/minute for the Predula compiler and about 1540 lines/minute for the Gnu C compiler on 1000 line programs.

10 Conclusions and Future Work

The presented system, DML-P, is to our knowledge one of the first denotational semantics based compiler generators that automatically generates practical compilers for data parallel languages. The system uses a continuation passing style internal representation. This is a graph that includes data-parallel operators. One important goal is to arrive at a sufficiently general set of operators in the intermediate representation to be useful as a target for compilers for a wide range of data-parallel languages. However, despite of being prototypes, compilers generated by DML-S or DML-P has demonstrated surprisingly good performance. Some directions for future work are shown below:

- Include multi-dimensional and nested/segmented arrays in the Predula Nouveau language to evaluate whether DML-P needs any extensions for this.
- Include a Forall construct.
- Code generation for a wider range of parallel architectures.
- Further development of the low-level operational semantics and the CPS graph so it can handle data-parallel operations with multi-dimensional and nested/segmented arrays.

Substantial work still needs to be done on the use of optimization techniques in order to generate better data-parallel object code. This includes further work on using and developing the CPS graph as a formally defined intermediate representation in which data-parallel operations can be expressed.

11 References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1991.

[2] Guy Blelloch. *Vector Models for Data-Parallel Computing.* MIT press, 1990.

[3] Guy Blelloch, Siddhartha Chatterjee, Fritzs Knabe, Jay Sipelstein, Marco Zagha. *VCODE Reference Manual (Version 1.3)* Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, Jul 1992.

[4] Michael J. C Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

[5] Robert Harper, Robin Milner, Mads Tofte. *The Definition of Standard ML, Version 4.* The MIT Press, 1990.

[6] W. Daniel Hillis, Guy L. Steele, Jr. *Data Parallel Algorithms,* Communications of the ACM, Dec 1986, Vol.29, No.12, pp. 1170-1183.

[7] C. A. R. Hoare. *Communicating Sequential Processes,* Prentice-Hall, 1985, ISBN-0-13-0153289-8.

[8] Kenneth E. Iverson *A Programming Language,* John Wiley & Sons, Inc., 1962.

[9] Pierre Jouvelot. *Designing new languages or new language manipulation systems using ML.* SIGPLAN Notices, 21:40–52, Aug 1986.

[10] Kathleen Knobe, J. D. Lucas, Guy L. Steele. *Data Optimization: Allocation of arrays to reduce communication on SIMD machines.* Journal of Parallel and Distributed Computing, 8:102–118, Feb 1990.

[11] Peter Lee. *Realistic Compiler Generation.* PhD thesis, University of Michigan, 1987. Ph.D. thesis published by MIT press 1989.

[12] Peter D. Mosses. *SIS - Semantic Implementation System.* PhD thesis, Aarhus University, 1979. TR DAIMI MD-30.

[13] Peter Mosses. *Unified Algebras and Action Semantics.* In Proc. Symp. of Theor. Sci. (STACS'89), LNCS-349, 1989.

[14] Edwin, M. Paalvast, A. J. Gemund, Henk J. Sips. *A method for parallel program generation with an application to the Booster language.* In Proceedings of the 1990 ACM Fourth International Conference on Supercomputing in Amsterdam, 11-15 Jun 1990.

[15] L. Paulson. *A semantics-directed compiler generator.* In Proceedings of the 9th ACM Conference on Principles of Programming Languages, pages 224–233, 1982.

[16] Mikael Pettersson. *Generating efficient code from continuation semantics.* In Proceedings of the 1990 Workshop on Compiler-Compilers, LNCS-477, Schwerin, Germany, 1990. Springer-Verlag.

[17] Mikael Pettersson, Peter Fritzson. DML - *A Meta-Language and System for the Generation of Practical and Efficient Compilers from Denotational Specifications.* In Proc. of the IEEE International Conference on Computer Languages, San Francisco, Apr 27-30, 1992.

[18] Mikael Pettersson. *DML - A Language and System for the Generation of Efficient Compilers from Denotational Specifications.* Licentiate thesis No 319. Department of Computer and Information Science, Linköping University, May 1992.

[19] H. Abelson, N. I. Adams IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele jr., G. J. Sussman, Mitchell Wand. *Revised^4 Report on the Algorithmic Language Scheme.* Nov 1991.

[20] Martin R. Raskovsky. *Denotational semantics as a specification of code generators.* In Proceedings of the ACM SIGPLAN 82 Conference on Compiler Construction, pages 230–244, 1982.

[21] Johan Ringström, Peter Fritzson, Johan Fagerström. *Predula, A Multiparadigm Parallel Programming and Debugging Environment,* In Proceedings of EUROMICRO'91 conference in Vienna, short note session, IEEE Sep 1991.

[22] Johan Ringström *Compiler Generation for Parallel Languages from Denotational Definitions.* Licentiate thesis, spring 1993. Thesis No. 380.

[23] J.R. Rose and Guy L. Steele, Jr. C*: *An extended C language for data parallel programming,* Technical Report PL 87-5, Thinking Machines Corporation, 1986.

[24] Gary W. Sabot: *Paralation Lisp - Architecture Independent Parallel Programming,* MIT Press, 1988.

[25] Ravi Sethi. *Control flow aspects of semantics directed compiling.* Technical Report CSTR-98, Bell Labs, 1981.

[26] David A. Schmidt. *Denotational Semantics, A Methodology For Language Development.* Allyn and Bacon Inc., 1986.

[27] Richard M. Stallman. *Using and Porting GNU CC.* 1989.

[28] Joseph E. Stoy. *Denotational Semantics.* MIT Press, 1977.

[29] Thinking Machines Corporation. *Introduction to Programming in C/Paris (Version 5).* Thinking Machines Corporation, Cambridge, Massachusetts, Jun 1989.

[30] Mitchell Wand. *A semantic prototyping system.* In Proc of the ACM SIGPLAN'84 Compiler Construction Conference, pages 213–222, 1984.