Polyhedral Compilation Package Interface Specification Draft

Jan Sjödin
Advanced Micro Devices
jan.sjodin@amd.com

Sebastian Pop
Advanced Micro Devices
sebastian.pop@amd.com

January 22, 2009

Abstract

The CLooG library provides code generation capabilities from a polyhedral representation. CLooG was designed to be a component of a compiler infrastructure, there is no existing framework to support transformations, dependence information or tracking additional information such as location information. Currently the internal representation is used as an interface to CLooG. This document defines the interface to the Polyhedral Compilation Package (PCP) which is a loop-optimization framework, including a language to interface between PCP and GCC. The goal is to create a package that will provide loop optimization and analysis capabilities to GCC.

1 Introduction

The CLooG library only provides code generation capabilities from a polyhedral representation. The Polyhedral Compilation Package (PCP) builds on top of CLooG to define a more complete loop optimization framework. The PCP framework contains an optimizer and APIs.

The polyhedral model can represent structured code containing sequences, linear conditions, well behaved loops and affine memory accesses. The compilation unit is a static control part SCoP, which does not have any side effects and all data accesses are statically determined to be linear.

Array subscripts are limited to affine expressions of induction variables and constants. Scalar identifiers defined outside a SCoP are called *parameters*. Parameters cannot be modified in a SCoP. Parameters and arrays that are read inside a SCoP are inputs. The output of a SCoP are the arrays that have been modified. The SCoP declares its inputs and outputs.

The intent of the external language of PCP is to hide the internal representation of PCP (PCP-IR) such that it can evolve without breaking backward compatibility. We define an array language as the interface. This external language only expresses the array accesses and does not specify computations.

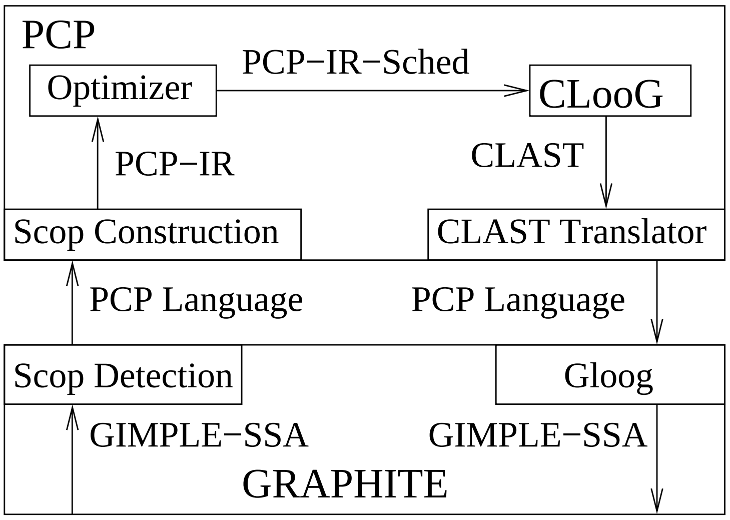

Figure 1: Polyhedral Compilation Package

Computations are encoded as “black boxes” that are parameterized with the reads and writes to arrays. This makes it easier for compilers to translate their internal representation to PCP. PCP is in essence a source to source compiler of the interface language.

Figure 1 shows an overview of PCP. The optimizer in PCP transforms the PCP internal representation (PCP-IR) to optimize the code. To guide the optimizer there are a number of heuristics needed. These heuristics are based on information about the architecture, which must be provided by GCC in the form of a machine description.

There are two more important aspects of the framework, the first is tracking of auxiliary information, the second is testing.

Auxiliary information is not needed to capture the semantics of the program, but may be either hints to the compiler or important information that must be maintained. This information is as annotations on the ASTs. In addition to being used as input to PCP the annotations are used as output from PCP. For example, a compiler may be interested in analysis information or statistics.

Testing and debugging must be an integral part of the design of PCP. We propose a syntax definition in addition to the ASTs so that it will be easy for humans to read and write code. This syntax is purely for testing purposes. It is crucial to be able to test the full expressiveness of the language and there is no guarantee that the front end is capable of producing all these cases. Annotations must also be part of the syntax, since these will be used both to test annotations and the annotations can be used to encode tests.

In the following sections we will describe the language that PCP compiles and definitions for both the syntax and data-structures for representing this language.

2 PCP External Language

In this section we will discuss the language component of the external interface. The complete grammar and API will be defined in the Appendix 6 and Appendix 5.

2.1 Types

PCP is a simple array language and uses a smaller set of types than a general programming language. There is only one scalar type: arbitrary precision integer that is used for array indexing, loop bonds and linear conditions. The only types that must be specified are array types. Arrays are defined by a list of constants that define the size of each dimension. If the list is empty the type stands for a scalar type.

2.2 Constants

Constants are integer type scalars.

2.3 Expressions

An expression is a linear combination of constants, parameters and induction variables. Parameters are declared inputs of the SCoP and never defined.

2.4 Array Definitions and Uses

Def and use define memory writes and reads. Each def/use takes a base array and a list of linear expression subscripts. A maydef encodes a possible write of a memory location, which may be used if there is control flow inside a user statement.

2.5 Statements

Statements are the constructs that modify the machine state, either control flow, or memory.

2.5.1 Copy Statement

The only statement that PCP knows about is the copy statement with two inputs, destination and source. For example:

```
loop (i, 0, ge(10, i), 1)
{
    copy(def(A, i), use(B, i))
}
```

2.6 User Statements

User statements define computations that read and write arrays, but have no other side effects. The user statement consists of a functor, that is a unique name for the statement. The arguments to the statement completely define the memory operations. The order of the arguments is maintained throughout the compilation. The access functions of uses and defs may be rewritten during the compilation process.

2.6.1 Statement Sequence (Block)

A block is a sequence of statements.

2.6.2 Guard

The guard defines a union of polyhedra as a disjunction of conjunctions of linear constraints.

There are two kinds of comparison operators: eq (equality) and ge (greater than or equal) Comparisons take two linear expressions as arguments. For example:

```
guard(or(and(ge(+(i, j), 0),
             eq(-(i, 5), 0)))),
         eq(i,0)))
{
  stmt(use(A, i, j, 10, 3);
}
```

2.6.3 Loop

The loop statement takes four arguments. First a variable declaration for the induction variable. Second, an expression which defines the initial value of the induction variables. Third, a boolean expression which determines when the loop exits. Fourth, the stride (increment) of the induction variable after each iteration. The loop implicitly defines the induction variable. The induction variable can only be accessed inside the loop body.

2.7 SCoP

A SCoP is the compilation unit. It has a set of inputs and outputs. The inputs are scalar values (parameters) which are invariant in the SCoP and arrays which can be modified. Outputs are arrays that have been modified and will be used after the scop.

3 Syntax

The reason for having a textual language interface is to simplify testing and debugging. If there is no simple way to read and understand a piece of code the debugging becomes a lot harder.

The syntax for the external language should be easy to read and write by humans and should not contain ambiguities. The expressiveness of the external language must not only be able to express all legal constructs, but also illegal constructs to allow for negative tests.

Annotations and tests can be encoded in the language through optional arguments. Optional arguments encode extra information that is not needed to express the meaning of the program, but that is needed for other reasons. By specifying a standard syntax to allow parsing optional arguments the parser would provide a listener interface, such that when an object has been created the listeners will be called with the optional arguments, the object and the location information.

To eliminate ambiguities, such as operator precedence and allow for a simple syntax for annotations we use functional form for all constructs. For the complete description of the syntax see Appendix 6.

3.1 Example

Below is a small fragment of C code. Assume that the arrays A, B and C have type double[1000][1000], and that N is a parameter, but it does not vary within the loops.

```
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < 100; j++)
    {
        A[i][j] = B[j][i] + C[i+1][i+j];
    }
}
```

This nest can be translated into a SCoP:

```
N <- parameter()
C <- array(1000, 1000)
B <- array(1000, 1000)
A <- array(1000, 1000)

scop (inputs(B, C), outputs(A), parameters(N))
{
  loop(i, 0, ge(N, i), 1)
  {
    loop(j, 0, ge(100, i), 1)
    {
      // userStmt maps to the add and assignment
      userStmt(def(A, i, j), use(B, j, i), use(C, +(i, 1), +(i, j))
    }
  }
}
```

4 Annotations

Annotations are used to represent auxiliary information that is needed for the compilation process. These can be added to any object in the language. Anno-

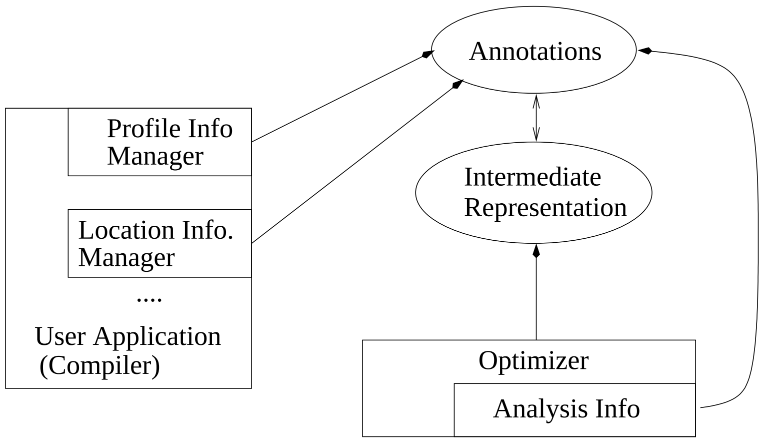

Figure 2: Annotation Framework

tations should be handled by a generic framework, that will allow a compiler to track the information as the code is transformed. One way of implementing this framework is to use a listener model, where a set of managers get informed of the kind of transform that has been performed and can update the annotations they own to reflect the transform. One issue with this approach is that the annotations are added to the ASTs but the internal representation is a set of constraints. That means the interface must be representation independent, and the modified annotations must be added by PCP to the constraints. During code generation CLooG must be able to take these annotations and add them to the generated AST. Figure 2 shows the communication between the different components.

Annotations consist of a tag and a list of annotation arguments. An annotation argument can be a scalar value, an identifier, a string or an annotation.

5 API

The PCP language interface contains the basic primitives to construct and traverse the PCP language data structures. There are very few modifiers exposed because the intent is not to manipulate code, only to transfer information. Some objects are more complicated to create, for example they may have varying number of arguments. Builder interfaces have been provided to aid the construction of these objects. A builder holds temporary information and once all the data has been collected the builder can do consistency checks and then create the object. Any casts or conversions have been encoded in functions. This will hopefully prevent any confusion about which other data types an object can be cast to.

5.1 PCP Object

The `pcp_object` is the base for all PCP datatypes. All objects can be named and have annotations.

```
pcp_object_kind pcp_object_get_kind (pcp_object *object);
void pcp_object_set_name (pcp_object *object, const char *name);
const char *pcp_object_get_name (pcp_object *object);
void pcp_object_set_annots (pcp_object *object, struct pcp_annot_set *annot);
struct pcp_annot_set *pcp_object_get_annots (pcp_object *object);

bool pcp_object_is_array_type (pcp_object *object);
bool pcp_object_is_expr (pcp_object *object);
bool pcp_object_is_bool_expr (pcp_object *object);
bool pcp_object_is_variable (pcp_object *object);
bool pcp_object_is_array_access (pcp_object *object);
bool pcp_object_is_stmt (pcp_object *object);
bool pcp_object_is_scop (pcp_object *object);

pcp_array_type *pcp_object_to_array_type (pcp_object *object);
pcp_expr *pcp_object_to_expr (pcp_object *object);
pcp_bool_expr *pcp_object_to_bool_expr (pcp_object *object);
pcp_variable *pcp_object_to_variable (pcp_object *object);
pcp_array_access *pcp_object_to_array_access (pcp_object *object);
pcp_stmt *pcp_object_to_stmt (pcp_object *object);
pcp_scop *pcp_object_to_scop (pcp_object *object);
```

5.2 PCP Array Type

Type describing the dimensions and size of variables.

```
pcp_object *pcp_array_type_to_object (pcp_array_type *type);

int pcp_array_type_get_num_dimensions (pcp_array_type *type);
int pcp_array_type_get_dimension (pcp_array_type *type, int size);
```

5.2.1 PCP Array Type Builder

Builder to make construction of array types easier.

```
pcp_array_type_builder *pcp_array_type_builder_create (void);
void pcp_array_type_builder_add_dimension (pcp_array_type_builder * builder,
                                           int dimension_size);
pcp_array_type *pcp_array_type_builder_create_type (pcp_array_type_builder *builder);
```

5.3 PCP Expr

Base type for linear expressions.

```
pcp_object *pcp_expr_to_object (pcp_expr *expr);
pcp_expr_kind pcp_expr_get_kind (pcp_expr *expr);

bool pcp_expr_is_parameter (pcp_expr *expr);
bool pcp_expr_is_constant (pcp_expr *expr);
bool pcp_expr_is_iv (pcp_expr *expr);
bool pcp_expr_is_arith (pcp_expr *);

pcp_parameter *pcp_expr_to_parameter (pcp_expr *expr);
pcp_constant *pcp_expr_to_constant (pcp_expr *expr);
pcp_iv *pcp_expr_to_iv (pcp_expr *expr);
pcp_arith *pcp_expr_to_arith (pcp_expr *expr);
```

5.4 PCP Constant

Datatype to represent integer constants.

```
pcp_object *pcp_constant_to_object (pcp_constant *constant);
pcp_expr *pcp_constant_to_expr (pcp_constant *constant);

int pcp_constant_get_value (pcp_constant *constant);

pcp_constant *pcp_constant_create (int value);
```

5.5 PCP IV

Datatype to represent an induction variable

```
pcp_object *pcp_iv_to_object (pcp_iv *iv);
pcp_expr *pcp_iv_to_expr (pcp_iv *iv);

const char *pcp_iv_get_name (pcp_iv *iv);

pcp_iv *pcp_iv_create (const char *name);
```

5.6 PCP Parameter

```
pcp_object *pcp_parameter_to_object (pcp_parameter *parameter);
pcp_expr *pcp_parameter_to_expr (pcp_parameter *parameter);

const char *pcp_parameter_get_name (pcp_parameter *parameter);

pcp_parameter *pcp_parameter_create (const char *parameter);
```

5.7 PCP Arith

Represents an arithmetic operation.

```
pcp_object* pcp_arith_to_object (pcp_arith* arith);
pcp_expr* pcp_arith_to_expr (pcp_arith* arith);

pcp_arith_operator pcp_arith_get_operator (pcp_arith* arith);
int pcp_arith_get_num_operands (pcp_arith* arith);
pcp_expr* pcp_arith_get_operand (pcp_arith* arith, int index);
```

5.7.1 PCP Arith Builder

Builder for arithmetic operations.

```
void pcp_arith_builder_set_operator (pcp_arith_builder* builder,
 pcp_arith_operator operator);
pcp_arith_operator pcp_arith_builder_get_operator (pcp_arith_builder* builder);
void pcp_arith_builder_add_operand (pcp_arith_builder* builder,
pcp_expr* operand);
pcp_arith* pcp_arith_builder_create_arith (pcp_arith_builder* builder);
pcp_arith_builder* pcp_arith_builder_create ();
```

5.8 PCP Bool Expr

Base datatype for boolean expressions.

```
pcp_bool_expr_kind pcp_bool_expr_get_kind (pcp_bool_expr *bool_expr);

bool pcp_bool_expr_is_bool_arith (pcp_bool_expr *bool_expr);
bool pcp_bool_expr_is_compare (pcp_bool_expr *bool_expr);

pcp_compare* pcp_bool_expr_to_compare (pcp_bool_expr *bool_expr);
pcp_bool_arith* pcp_bool_expr_to_compare (pcp_bool_expr *bool_expr);
```

5.9 PCP Compare

```
pcp_bool_expr* pcp_compare_to_bool_expr (pcp_compare* compare);

pcp_compare_operator pcp_compare_get_operator (pcp_compare* compare);
pcp_expr *pcp_compare_get_lhs (pcp_compare* compare);
pcp_expr *pcp_compare_get_rhs (pcp_compare* compare);

pcp_compare* pcp_compare_create (pcp_compare_operator operator,
 pcp_expr* lhs,
 pcp_expr* rhs);
```

5.10 PCP Bool Arith

```
pcp_object* pcp_bool_arith_to_object (pcp_bool_arith* bool_arith);
pcp_bool_expr* pcp_bool_arith_to_bool_expr (pcp_bool_arith* bool_arith);

pcp_bool_arith_operator pcp_bool_arith_get_operator (pcp_bool_arith* bool_arith);
int pcp_bool_arith_get_num_operands (pcp_bool_arith* bool_arith);
pcp_bool_expr* pcp_bool_arith_get_operand (pcp_bool_arith* bool_arith,
                                           int index);
```

5.10.1 PCP Bool Arith Builder

```
void pcp_bool_arith_builder_set_operator (pcp_bool_arith_builder* builder,
  pcp_bool_arith_operator operator);
pcp_bool_arith_operator pcp_bool_arith_builder_get_operator (pcp_bool_arith_builder* builder);
pcp_dynamic_pointer_array* pcp_bool_arith_builder_get_operands (pcp_bool_arith_builder* builder);
pcp_bool_arith_builder* pcp_bool_arith_builder_create ();
pcp_bool_arith* pcp_bool_arith_builder_create_bool_arith (pcp_bool_arith_builder* builder);
void pcp_bool_arith_builder_add_operand (pcp_bool_arith_builder* builder,
 pcp_bool_expr* operand);
```

5.11 PCP Variable

```
pcp_object *pcp_variable_to_object (pcp_variable *var);

bool pcp_variable_get_is_input (pcp_variable *var);
bool pcp_variable_get_is_output (pcp_variable *var);
pcp_array_type *pcp_variable_get_type (pcp_variable *var);
const char *pcp_variable_get_name (pcp_variable *var);

void pcp_variable_set_is_input (pcp_variable *var, bool is_input);
void pcp_variable_set_is_output (pcp_variable *var, bool is_output);

pcp_variable *pcp_variable_create (pcp_array_type *type, const char *name);
```

5.12 PCP Array Access

```
pcp_object *pcp_array_access_to_object (pcp_array_access *access);

pcp_array_operator pcp_array_access_get_operator (pcp_array_access *access);
pcp_variable *pcp_array_access_get_base (pcp_array_access *access);
int pcp_array_access_get_num_subscripts (pcp_array_access *access);
pcp_expr *pcp_array_access_get_subscript (pcp_array_access *access,
  int index);

/* These typdefs are only for implicit documentation. */
typedef pcp_array_access pcp_array_use;
typedef pcp_array_access pcp_array_def;
typedef pcp_array_access pcp_array_maydef;

bool pcp_array_access_is_use (pcp_array_access *access);
bool pcp_array_access_is_def (pcp_array_access *access);
bool pcp_array_access_is_maydef (pcp_array_access *access);

pcp_array_use *pcp_array_access_to_use (pcp_array_access *access);
pcp_array_def *pcp_array_access_to_def (pcp_array_access *access);
pcp_array_maydef *pcp_array_access_to_maydef (pcp_array_access *access);
```

5.12.1 PCP Array Access Builder

Builder used to make construction of array accesses easier.

```
void pcp_array_access_builder_set_operator (pcp_array_access_builder * builder,
                                            pcp_array_operator operator);
pcp_array_access_builder *pcp_array_access_builder_create (pcp_variable * base);
pcp_array_access *pcp_array_access_builder_create_access (pcp_array_access_builder *builder);
void pcp_array_access_builder_add_subscript (pcp_array_access_builder *builder,
                                             pcp_expr *subscript);
```

5.13 PCP Stmt

Base type for statements.

```
pcp_object *pcp_stmt_to_object (pcp_stmt *stmt);
pcp_stmt_kind pcp_stmt_get_kind (pcp_stmt *stmt);
```

```
bool pcp_stmt_is_copy (pcp_stmt *stmt);
bool pcp_stmt_is_user_stmt (pcp_stmt *stmt);
bool pcp_stmt_is_guard (pcp_stmt *stmt);
bool pcp_stmt_is_loop (pcp_stmt *stmt);
bool pcp_stmt_is_sequence (pcp_stmt *stmt);

pcp_copy *pcp_stmt_to_copy (pcp_stmt *stmt);
pcp_user_stmt *pcp_stmt_to_user_stmt (pcp_stmt *stmt);
pcp_guard *pcp_stmt_to_guard (pcp_stmt *stmt);
pcp_loop *pcp_stmt_to_loop (pcp_stmt *stmt);
pcp_sequence *pcp_stmt_to_sequence (pcp_stmt *stmt);
```

5.14 PCP Copy

```
pcp_object *pcp_copy_to_object (pcp_copy *copy);
pcp_stmt *pcp_copy_to_stmt (pcp_copy *copy);

pcp_array_use *pcp_copy_get_src (pcp_copy *copy);
pcp_array_def *pcp_copy_get_dest (pcp_copy *copy);

pcp_copy *pcp_copy_create (pcp_array_def *dest, pcp_array_use *src);
```

5.15 PCP User Stmt

Represents a black box (computation). Array accesses are represented as arguments to the statement.

```
pcp_object *pcp_user_stmt_to_object (pcp_user_stmt *user_stmt);
pcp_stmt *pcp_user_stmt_to_stmt (pcp_user_stmt *user_stmt);

const char *pcp_user_stmt_get_name (pcp_user_stmt *user_stmt);
int pcp_user_stmt_get_num_accesses (pcp_user_stmt *user_stmt);
pcp_array_access *pcp_user_stmt_get_array_access (pcp_user_stmt *user_stmt,
                                                  int index);
```

5.15.1 PCP User Stmt Builder

```
void pcp_user_stmt_builder_set_name (pcp_user_stmt_builder *builder,
                                     const char *name);
void pcp_user_stmt_builder_add_access (pcp_user_stmt_builder *builder,
                                       pcp_array_access *access);
pcp_user_stmt_builder *pcp_user_stmt_builder_create ();
pcp_user_stmt *pcp_user_stmt_builder_create_user_stmt (pcp_user_stmt_builder *builder);
```

5.16 PCP Sequence

```
pcp_object* pcp_sequence_to_object(pcp_sequence* sequence);
pcp_stmt *pcp_sequence_to_stmt (pcp_sequence *sequence);

int pcp_sequence_get_num_stmts (pcp_sequence *sequence);
pcp_stmt *pcp_sequence_get_stmt (pcp_sequence *sequence, int index);
```

5.16.1 PCP Sequence Builder

```
void pcp_sequence_builder_add (pcp_sequence_builder *builder, pcp_stmt *stmt);
pcp_sequence_builder *pcp_sequence_builder_create ();
pcp_sequence *pcp_sequence_builder_create_sequence (pcp_sequence_builder *builder);
```

5.17 PCP Guard

```
pcp_object *pcp_guard_to_object (pcp_guard *guard);
pcp_stmt *pcp_guard_to_stmt (pcp_guard *guard);

pcp_bool_expr *pcp_guard_get_condition (pcp_guard *guard);
pcp_stmt *pcp_guard_get_body (pcp_guard *guard);

pcp_guard *pcp_guard_create (pcp_bool_expr *condition, pcp_stmt *body);
```

5.18 PCP Loop

```
pcp_object *pcp_loop_to_object (pcp_loop *loop);
pcp_stmt *pcp_loop_to_stmt (pcp_loop *loop);

pcp_iv *pcp_loop_get_iv (pcp_loop *loop);
pcp_expr *pcp_loop_get_start (pcp_loop *loop);
pcp_expr *pcp_loop_get_end (pcp_loop *loop);
pcp_constant *pcp_loop_get_stride (pcp_loop *loop);
pcp_stmt *pcp_loop_get_body (pcp_loop *loop);

pcp_loop *pcp_loop_create (pcp_iv *iv, pcp_expr *start, pcp_expr *end,
   pcp_constant *stride, pcp_stmt *body);
```

5.19 PCP Scop

```
pcp_object *pcp_scop_to_object (pcp_scop *scop);

int pcp_scop_get_num_variables (pcp_scop *scop);
int pcp_scop_get_num_parameters (pcp_scop *scop);
pcp_variable *pcp_scop_get_variable (pcp_scop *scop, int index);
pcp_parameter *pcp_scop_get_parameter (pcp_scop *scop, int index);
pcp_stmt *pcp_scop_get_body (pcp_scop *scop);
```

5.19.1 PCP Scop Builder

```
void pcp_scop_builder_add_variable (pcp_scop_builder *builder,
    pcp_variable *variable);
void pcp_scop_builder_add_parameter (pcp_scop_builder *builder,
     pcp_parameter *parameter);
void pcp_scop_builder_set_body (pcp_scop_builder *builder, pcp_stmt *body);
pcp_stmt *pcp_scop_builder_get_body (pcp_scop_builder *builder);

pcp_scop_builder *pcp_scop_builder_create ();
pcp_scop *pcp_scop_builder_create_scop (pcp_scop_builder *builder);
```

5.20 PCP Annot Set

Container for the annotations of an object.

```
pcp_annot_set *pcp_annot_set_create ();
int pcp_annot_set_get_num_annots (pcp_annot_set *annot_set);
pcp_annot_term *pcp_annot_set_get_annot (pcp_annot_set *annot_set, int index);
pcp_annot_term *pcp_annot_set_get_annot_with_tag (pcp_annot_set *annot_set,
                                                  const char *tag);
void pcp_annot_set_add_annot (pcp_annot_set *annot_set, pcp_annot_term *annot);
```

5.21 PCP Annot

Base datatype for annotations.

```
pcp_annot_kind pcp_annot_get_kind (pcp_annot *annot);
void pcp_annot_initialize (pcp_annot *annot, pcp_annot_kind kind);

bool pcp_annot_is_annot_int (pcp_annot *annot);
bool pcp_annot_is_annot_string (pcp_annot *annot);
bool pcp_annot_is_annot_object (pcp_annot *annot);
bool pcp_annot_is_annot_term (pcp_annot *annot);

pcp_annot_int *pcp_annot_to_annot_int (pcp_annot *annot);
pcp_annot_string *pcp_annot_to_annot_string (pcp_annot *annot);
pcp_annot_object *pcp_annot_to_annot_object (pcp_annot *annot);
pcp_annot_term *pcp_annot_to_annot_term (pcp_annot *annot);
```

5.22 PCP Annot Int

Annotation with an integer value.

```
pcp_annot *pcp_annot_int_to_annot (pcp_annot_int *annot_int);
int pcp_annot_int_get_value (pcp_annot_int *annot_int);
pcp_annot_int *pcp_annot_int_create (int value);
```

5.23 PCP Annot String

Annotation with a string value.

```
pcp_annot *pcp_annot_string_to_annot (pcp_annot_string *annot_string);
const char *pcp_annot_string_get_string (pcp_annot_string *annot_string);
pcp_annot_string *pcp_annot_string_create (const char *string);
```

5.24 PCP Annot Object

Annotation with an object value.

```
pcp_annot *pcp_annot_object_to_annot (pcp_annot_object *annot_object);
pcp_object *pcp_annot_object_get_object (pcp_annot_object *annot_object);
pcp_annot_object *pcp_annot_object_create (pcp_object *object);
```

5.25 PCP Annot Term

Compound annotation. The tag is used to identify what kind of annotation it is and to allow annotation managers to exclusively handle the annotations they own.

```
pcp_annot *pcp_annot_term_to_annot (pcp_annot_term *annot_term);
const char *pcp_annot_term_get_tag (pcp_annot_term *annot_term);
int pcp_annot_term_get_num_arguments (pcp_annot_term *annot_term);
pcp_annot *pcp_annot_term_get_argument (pcp_annot_term *annot_term,
int index);
pcp_annot_term *pcp_annot_term_create (const char *tag,
       int num_arguments,
       pcp_annot ** arguments);
```

5.25.1 PCP Annot Term Builder

Builder to easily construct an `pcp_annot_term`.

```
void pcp_annot_term_builder_set_tag (pcp_annot_term_builder *builder,
                                     const char *tag);
const char *pcp_annot_term_builder_get_tag (pcp_annot_term_builder *builder);
void pcp_annot_term_builder_add_argument (pcp_annot_term_builder *builder,
                                          pcp_annot *argument);
pcp_annot_term_builder *pcp_annot_term_builder_create ();
pcp_annot_term *pcp_annot_term_builder_create_annot (pcp_annot_term_builder *builder);
```

6 Test Language Grammar

6.1 Numerals

6.1.1 Grammar

```
  Digit ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'
Numeral ::= ['-'] Digit+
```

6.1.2 Examples

```
Numeral examples: -13
                  555
                  -0653
```

6.2 Identifiers and Strings

6.2.1 Grammar

```
Letter ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
         | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r'
         | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' |
         | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I'
         | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R'
         | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'

Symbol ::= '!' | '@' | '#' | '$' | '%' | '^' | '&' | '*' | '_'
         | '-' | '=' | '+' | '/' | ';' | ':' | '"' | '<' | ','
         | '>' | '.' | '?'

      SymbolLetter ::= Symbol | Letter
SymbolLetterDigit ::= Symbol | Letter | Digit
```

```
          String ::= '"' SymbolLetterDigit* '"'
      Identifier ::= Letter [SymbolLetterDigit*]
        Operator ::= SymbolLetter [SymbolLetterDigit*]
```

6.2.2 Examples

```
String examples: "Hello World!"
                 "#j93hjci3"
                 "?;++"
Identifier examples: foo
                     g65
                     h_#?95
Operator examples: +
                   x53#
                   <<a
```

6.3 Annotations

6.3.1 Grammar

```
           OptArgTag ::= Identifier
          OptArgAtom ::= Numeral
                       | Identifier
                       | String
          OptArgTerm ::= OptArgTag '(' OptArgListContent' )

NEOptArgtListContent ::= OptArgTerm [',' NEOptArgtListContent]
   OptArgListContent ::= epsilon
                       | NEOptArgtListContent
          OptArgList ::= epsilon
                       | '|' OptArgListContent
```

6.3.2 Examples

```
OptArgTerm examples: | foo(bar, baz)
                     | transform(block)
                     | loc(file("myfile.c"), line(372)), freq(5543)
```

6.4 Types

6.4.1 Grammar

```
        NEConstList ::= (Identifier | Numeral) [',' NEConstList]
        ConstList ::= epsilon
                    | NEConstList
             Type ::= 'array' '(' ConstList OptArgList ')'
```

6.4.2 Examples

```
TypeDef Examples: myArray <- array ( 3, 2 )
```

6.5 Variables

6.5.1 Grammar

```
VarDecl ::= Identifier '<-' 'variable' '(' (Type | Identifier) OptArgList ')'
```

6.5.2 Examples

```
VarDecl examples: myArrayVar <- variable(array(32, 32, 10))
```

6.6 Parameters

```
ParameterDecl ::= Identifier '<-' 'parameter' '(' OptArgList ')'
```

6.6.1 Examples

```
Parameter examples: myParam <- parameter()
```

6.7 Expressions

6.7.1 Grammar

```
MultExpr ::= '*' '(' Numeral ',' Identifier ')'
ArithOperator ::= '+' | '-' | 'min' | 'max' | 'floor' | 'ceil'
NELinearExprList ::= LinearExpr [',' NELinearExprList]
LinearExpr ::= Identifier
             | Numeral
             | MultExpr
             | ArithOperator '(' LinearExpr ',' NELinearExprList ')'
EqExpr ::= ( 'eq' | 'ge' ) '(' LinearExpr ',' LinearExpr ')'
BoolOperator ::= 'and' | 'or '
NEBoolExprList ::= BoolExpr [',' NEBoolExprList]
BoolExpr ::= EqExpr
           | BoolOperator '(' BoolExpr ',' NEBoolExprList ')'


NEExprArgList ::= LinearExpr [',' NEExprArgList]
  ExprArgList ::= epsilon | NEExprArgList
         Def  ::= 'def' '(' Identifier (epsilon | ',' NEExprArgList) ')'
         Use  ::= 'use' '(' Identifier (epsilon | ',' NEExprArgList) ')'
   UseDefList ::= (Use | Def) [',' UseDefList]
```

6.8 Statements

6.8.1 Grammar

```
UserStmt  ::= Identifier '(' UseDefList [',' Numeral ',' Numeral] OptArgList ')'
CopyStmt  ::= 'copy' '(' Def ',' Use [',' Numeral ',' Numeral] OptArgList ')'
GuardStmt ::= 'guard' '(' DisjExpr OptArgList ')'  '{' StmtList '}'
LoopStmt  ::= 'loop' '(' Identifier ',' LinearExpr ',' BoolExpr ','
              Numeral OptArgList ')' '{' StmtList '}'
     Stmt ::= UserStmt
            | CopyStmt
            | GuardStmt
            | LoopStmt
 StmtList ::= Stmt+
```

6.8.2 Examples

```
GuardStmt example: guard(ge(-(i, 10)))
                    {
                      mystmt(use(A, -(x,y))
                    }

LoopStmt example: loop(i, 0, ge(100, i), 1)
                   {
                      mystmt2(def(A, i), use(A, +(i, 1))
                   }
```

6.9 Scops

6.9.1 Grammar

```
NEIdentifierList ::= Identifier | Identifier ',' NEIdentifierList
IdentifierList ::= epsilon | NEIdentifierList
Inputs ::= 'inputs' '(' IdentifierList ')'
Outputs ::= 'outputs' '(' IdentifierList ')'
Parameters ::= 'parameters' '(' IdentifierList ')'

Decls ::= [VarDecl | ParameterDecl]*

Scop ::= Decls 'scop' '(' Inputs ',' Outputs ',' Parameters  OptArgList ')'
         '{' StmtList '}

Program ::= Scop
```

6.9.2 Examples

```
Program example:

B <- array(10)
N <- parameter()
A <- array(10)
scop(inputs(A), outputs(B), parameters(N))
{
    loop(i, 0, N, 1)
    {
        mystmt(def(B, i), use(A, +(i, 1)))
    }
}
```