Oil Reference Manual

$Revision: 1.17 $

Compiler Tools Group
Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO, USA
80309-0425

Table of Contents

1 Introduction **3**

- 1.1 Operator identification 3
- 1.2 Coercions 4
- 1.3 Sets of possible types 4
- 1.4 Classes 5
- 1.5 Names for operators, types and classes 5

2 A review of an example OIL Specification **7**

3 OIL's Specification Language **9**

- 3.1 Identifiers 9
- 3.2 Comments 10
- 3.3 Statement types 10
  - 3.3.1 Operator definition 11
    - 3.3.1.1 Function Signatures 12
  - 3.3.2 Coercion operator definition 13
  - 3.3.3 Operator identification 14
  - 3.3.4 Class definition 15
  - 3.3.5 Type set definition 15

4 OIL's Support Library **17**

- 4.1 Library Types 18
- 4.2 Set of Possible Types 19
- 4.3 Validating operator identification 20
- 4.4 Simpler operator identification 20
- 4.5 Looking at an operator's type signature 21
- 4.6 Coercion sequences 21
- 4.7 Instantiate Classes 22
- 4.8 Name Functions 22
- 4.9 Compile Time 22
  - 4.9.1 Types 23
  - 4.9.2 Operators 23
  - 4.9.3 Argument Signatures 24
  - 4.9.4 Coercions 24
  - 4.9.5 Identifications 24
  - 4.9.6 Classes 25

5 Relating an OIL specification to library function calls . . . 27

- 5.1 Using Names . . . 27
- 5.2 A simple example . . . 27
  - 5.2.1 Definitions from the specification . . . 28
  - 5.2.2 Operator Identification . . . 28
  - 5.2.3 Operator Signatures . . . 28
  - 5.2.4 Coercion sequence . . . 29
- 5.3 A more complex example . . . 29
  - 5.3.1 Using type sets . . . 30
- 5.4 Using Classes . . . 31

6 Design Hints . . . 33

- 6.1 Incremental Design and Implementation . . . 33
- 6.2 Identifying Operands . . . 33
- 6.3 Error Handling . . . 33
- 6.4 Modeling OIL's function . . . 34
- 6.5 Schema Restrictions . . . 34
- 6.6 Identification Algorithm . . . 34

Index . . . 35

This a reference manual for OIL (Eli's operator identification specification facility.) OIL is a specification language and a set of library functions used in the specification and implementation of operator identification within a compiler.

The specification language is the means for describing the operator identification for a compiler's source language and a mapping into the compiler's target language. The support library is the means for using the relationships described in the user's specification to implement the desired operator identification in the context of an attribute grammar. The relationship between a user's specification and calls on the library functions are elaborated by means of some examples.

1 Introduction

OIL deals with three primary concepts: operators, types and indications. The operators which OIL refers to are strongly typed functions. The types which are named in an OIL specification are referred to as 'primitive types' but there are also 'constructed types' which are created at compile time. Indications are a name given to a list of operators which may be identified by an appearance of an indication in an expression.

Although operators, types and indications are primary terms in discussing OIL and operator identification, the rest of this chapter is devoted to explanations of:

`Operator identification::`
: A review of operator identification by example.

`Coercions::`
: Coercions allow the compiler to use a value of one type where another is specified.

`Sets of possible types::`
: The set of all the types which an expression might return.

`Classes::`
: A set of operators and coercions which can be instantiated for a newly created constructed type.

`Names::` An enumeration associated with each operator, type and class identifier.

1.1 Operator identification

Anyone familiar with compiler construction is familiar with operator identification but may not necessarily know it by that name. Thus operator identification is introduced by an example.

In Pascal the expression `x+y` can have a number of meanings or it may be invalid. For it to be valid `x` and `y` must refer to type values which allow the identification of a well typed operator associated with `+`. The operator identified might be *integer addition*, *real addition* or *set union*. The operator identified is determined by the types associated with `x` and `y` and be formulated like:

- `x` and `y` are both integer then *integer addition*
- `x` and `y` are *sets* with the same element type then *set union*
- one of `x` and `y` is *real* and the other is *real* or *integer* then *real addition*

- otherwise `x+y` is undefined or an error

The `+` in `x+y` is called the indication and `x` and `y` are the operands. The process of operator identification chooses a strongly typed operator associated with the indication and has a type signature which matches the types of the operands. This is a simplification but it does capture the essentials of operator identification.

The operation of 'matching' the operand type with the type signature is really the relation 'is coercible to', see Section 1.2 [Coercions], page 4.

1.2 Coercions

A coercion is a distinguished operator which the compiler may insert in an expression to make operator identification possible. In the previous section the Pascal expression `x+y` could identify a real expression if `x`'s type was real but `y`'s type was integer. OIL allows this to be specified using a coercion from integer to real. Thus whenever an integer is supplied but a real is required the compiler may apply the coercion to the integer argument and thus have only real arguments to identify real addition with.

If there is a coercion from type A to type B then type A is said to be 'coercible to' type B.

A coercion sequence is any number of coercions applied in sequence. In OIL a coercion sequence is used as if it were a single coercion. Thus if type A is coercible to type B and type B is coercible to type C then there is a coercion sequence from type A to type C. The elements in the sequence are the coercion operators in the order they need to be applied to satisfy the type constraint. Thus using a coercion sequence type A may match and argument of type C.

Coercion sequences lead to a revised definition of 'coercible to'. If there is a coercion sequence from type A to type B then type A is said to be 'coercible to' type B. Note that since a coercion sequence may be empty every type is coercible to itself.

1.3 Sets of possible types

Each type has an associated *set of possible types*. A type B is in the set of possible types for type A if A is coercible to B.

Likewise an expression may have a set of possible types. Take `x+y` as an example. The set of possible types associated with the expression would include the result type of any operator which can be identified by `+` and an element from the set of possible types for `x` and an element from the set of possible types for `y`.

Note that the set of possible types is a transitive closure of the relation 'is coercible to'. When a new coercion is created then the transitive closure of every type may need to be updated. But in practice this is not the case and most coercions have no impact on the set of possible types for most types.

1.4 Classes

Formally OIL's classes are parametric abstract types. But they may be most easily comprehended as a procedure which takes some types as an argument and creates a new type and adds a set of operators and coercions into an existing type schema. The net effect being that the new operators may now be identified and the set of possible types for some types may be increased.

Consider the subrange type constructor in Pascal. It creates a coercion from the subrange to the base type.

1.5 Names for operators, types and classes

You will want associate semantics to the identifiers in your OIL specification. *Names* allow a convenient way of doing that. All types, classes and operators have names associated with them. The name is represented as a definition table key (see Section 'The Definition Table Module' in Definition Table) and can be referenced in your attribute grammar by the identifier used in the OIL specification.

The name associated with an ADT type, class or operator is accessed by using the unary functions: `OilTypeName`, `OilClassName` and `OilOpName`.

2 A review of an example OIL Specification

The following is a complete OIL specification which we will analyze more completely later (see Chapter 5 [Interrelationship], page 27.) But we will briefly review it here to give an intuitive feel for both the syntax and the semantics of the language.

In our example below is a specification of Pascal's overloaded operator (‘+'.) It can be identified with either integer addition (*iAdd*), real addition (*rAdd*) or set union (*sUnion*.) And we define a coercion operator (*Float*) to allow integer values to be used where real values may appear.

```
OPER iAdd( int_t, int_t ): int_t;     /* the usual '+' operators for Pascal
*/
OPER rAdd( real_t, real_t ): real_t;
OPER sUnion( set_t, set_t ): set_t;

INDICATION Plus: iAdd, rAdd, sUnion;  /* will be identified together */

COERCION Float( int_t ): real_t;    /* usual Pascal coercion from int to real
*/
```

An OIL specification defines the identifiers for use in calls to the library functions. The above specification defines these identifiers:

- `int_t`, `real_t`, `set_t` to be type denotations.
- `iAdd`, `rAdd`, `sUnion` to be typed binary operator denotations with the expected functional signatures.
- `Plus` to be an operator indication which may be identified with `iAdd`, `rAdd` and `sUnion`.
- `Float` is defined to be a coercion from `int_t` to `real_t`.

For a more in-depth examination of this specification see Section 5.2.4 [Simple Example], page 29.

3 OIL's Specification Language

An OIL specification is composed of a list of statements. The statements describe relationships among identifiers which the library functions will interpret as describing an operator identification scheme.

With the specification language a compiler writer defines four sets of mutually exclusive identifiers:

- `Operators` represent both indications and denotations. That is an operator can be used as an indication and also have a type signature.
- `Types` represent primitive types. Unlike the types which are instantiations of classes.
- `Classes` represent parameterized sets of operators. A new version of each operator is constructed for each instantiation of the class. Likewise each instantiation of a class creates a new type which is not primitive.
- `Type Sets` represent a set of primitive types which can be used to construct multiple instances of an operator; one for each element in the type set.

The rest of this chapter of the manual describes the syntax of the specification language by covering the three main lexical and syntactic constructs: `Identifiers`, `Statements` and `Comments`.

3.1 Identifiers

An `identifier` in an OIL specification is a sequence of characters which:

- begins with an alphabetic character: A-Z, a-z
- continues with an alphanumeric or underscore character: A-Z, a-z, 0-9, _.

Examples:

```
iAdd, Fmul2
```

Identifiers are defined both explicitly and implicitly. OPER and COERCION statements explicitly define operators. CLASS statements explicitly define classes. SET statements explicitly define sets. Types are all implicitly defined; if an identifier appears only in a function signature or as a type element in a type set expression then it is a type. If the indication in an INDICATION statement has no defining OPER statement it is implicitly defined as an operator.

An identifier can only be in one of the sets: `operator`, `type`, `class` or `type set`.

Constraints: An identifier can denote a `type` or an `operator` but not both.

An `identifier list` is a sequence of identifiers separated by commas.

Since OIL produces names to be used in your attribute grammar, you must not use a reserved word of your attribute grammar as an identifier in an OIL specification.

Examples:

```
iAdd,iMul  or  Fmul, Fadd, Fdiv
```

3.2 Comments

C-style comments, beginning with `/*` and ending with `*/`, are allowed in an OIL specification. A comment may appear anywhere that white space might be appropriate.

3.3 Statement types

There are five types of statements in OIL:

- `Operator definition` which defines the functional type signature of one or more operator denotations.
- `Coercion operator definition` which defines a coercion operator and the ability for the source type of the coercion operator to be acceptable in place of the destination type of the coercion operator when performing operator identification.
- `Operator identification` which defines an operator indication and a set of operator denotations which the indication may identify.
- `Class definition` which defines a set of operators to be constructed when a parameterized type is instantiated.

- `Type set definition` which defines an identifier to represent a set of types and allows explicit multiple operator definitions when used in an operator definition.

3.3.1 Operator definition

The basic form of an operator definition is:

```
'OPER' <Op-name> '(' <arg-list> ')' ':' <result-id> ';'
```

where:

`<Op-name>` is an operator identifier.
`<arg-list>` is a list of identifiers separated with commas which describes the argument signature of the operator.
`<result-id>` is an identifier which determines the result type of the operator.

The `<result-id>` and each identifier in the `<arg-list>` may be either a primitive type name, a SET name or if in a CLASS definition then they may refer to the CLASS name being defined or one of the parameters to the CLASS definition. see Section 3.3.1.1 [Function Signature], page 12

Constraints: Any given `<Op-name>` can appear only once in all definition statements.

An example is:

```
OPER iAdd( int_t, int_t ): int_t;
```

In the example operator definition statement above, 'iAdd' is defined to be an operator and 'int_t' to be an operand type.

The multiple form of an operator definition is:

```
'OPER' <Op-name-list> '(' <arg-list> ')' ':' <result-id> ';'
```

where:

`<Op-name-list>` is a list of operators separated by commas.
`<arg-list>` and `<result-id>` are as before.

All the operators appearing in `<Op-name-list>` are given the same functional signature.

Constraints: Each operator in `<Op-name-list>` must appear only once in any operator definition.

An example is:

```
OPER rAdd, rSub, rMul, rDiv ( real_t, real_t ): real_t;
```

3.3.1.1 Function Signatures

How many and what kind of operator is being defined can vary a great deal depending on the definition of the identifiers which appear in the function signature. There are essentially three different kinds of function signatures:

Simple signature which only references primitive types.

Class signature which references a class or parameter name, though it may also refer to primitive types.

Set signature which references a set name in its signature, though it may also refer to primitive types.

A simple signature only defines a single operator.

A class signature defines a pattern of a single operator to be created when the class is instantiated. The actual signature constructed has the class name replaced by the created type and the parameter names are replaced with the corresponding positional argument which is used in instantiating the class.

A set signature defines one operator for each element in the value of the referenced set. Consider the example:

```
SET s=[a,b];
sop(s):c;
```

There are two operators with the name `sop` defined, with the signatures:

```
sop(a):c;
```

```
sop(b):c;
```

If a set name is referenced more than once in the signature the same value appears in the corresponding position in the signature. For example consider the specification:

```
SET s=[a,b];
SET r=[c,d];
OPER sop(s,r):s;
```

Four operators with the name `sop` are created. With the four signatures:

```
sop(a,c):a;
sop(b,c):b;
sop(a,d):a;
sop(b,d):b;
```

The signature is duplicated once for each value in a unique set name. The set name is replaced with each value in turn reguardless of how many times the set name is referenced in the signature.

Constraints:

If an identifier in a function signature is a CLASS name or a parameter to a CLASS then the operator definition must be in the body of the CLASS definition.

No reference to a SET name may be used in a CLASS operator definition.

3.3.2 Coercion operator definition

The basic form of a coercion operator definition is:

```
'COERCION' <Cop-name> '(' <source-id> ')' ':' <result-id> ';'
```

where:

`<Cop-name>` is a coercion operator identifier.
`<source-id>` is an identifier which determines the source type of the coercion.
`<result-id>` is an identifier which determines the result type of the coercion.

The `<source-id>` and `<result-id>` may be either a primitive type name, a SET name or if in a CLASS definition then they may refer to the CLASS name being defined or one of the parameters to the CLASS definition.

An example is:

```
COERCION cFloat( int_t ): real_t;
```

In the example coercion operator definition statement above, 'cFloat' is defined to be an operator and, 'int_t' and 'real_t' are defined to be a operand types.

Constraints: `<Cop-name>` can appear in only one definition statement.

3.3.3 Operator identification

The basic form of an operator identification is:

```
'INDICATION' <Op-name> ':' <Op-name-list> ';'
```

where:

`<Op-name>` is an operator identifier.
`<Op-name-list>` is list of operator denotations separated by commas.

The order of appearance from left to right of the operators in the `<Op-name-list>` determines a search order for the identification process. When an operator identification operation is performed on `<Op-name>` then each operator in `<Op-name-list>` is tried from left to right.

Constraints:

- All the operator denotations referenced must have signatures of the same length. (i.e. If the first operator denotation has two arguments all the rest of the operators referenced must also have two arguments.)
- All operator denotations must have appeared in an operator definition statement.
- `<Op-name>` must not appear in any other operator identification statements. All potential operators to be identified by `<Op-name>` must appear in one operator identification statement.

An example is:

```
INDICATION Plus: iAdd, rAdd, sUnion;
```

In the example operator identification statement above, 'Plus' is defined to be an operator and, 'iAdd', 'rAdd' and 'sUnion' are defined to be operators.

The example defines that 'iAdd' will be the first operator tested for identification of 'Plus' and 'sUnion' would be the last operator tested for identification.

3.3.4 Class definition

The form of class definition is:

```
'CLASS' <Class-name>'(' <Param-name-list> ')' <simple-stmts> 'END' ';'
```

where:

`<Class-name>` is a class identifier.
`<Param-name-list>` is a list of parameter identifiers separated by commas.
`<simple-stmts>` is a set of operator and coercion statements which will be created by an instantiation of this class.

The declarations in the `<simple-stmts>` do not define operators and coercions but patterns for the creation of such operators. When a class is instantiated then the patterns for that class are used to define the operators and coercions from the patterns.

The type identifiers referenced in the declarations in `<simple-stmts>` may refer to the class name(`<Class-name>`), a specific primitive type or a parameter name(from `<Param-name-list>`.) When a class is instantiated a type corresponding to each identifier is used to create an operator of coercion from the patterns in `<simple-stmts>`.

3.3.5 Type set definition

The form of set definition is:

```
'SET' <Set-name>'=' <set-expression> ';'
```

where:

<Set-name> is a set identifier.
<set-expression> is a expression which defines the types which are members of the set.

A set expression may be composed of any of the following constructs(where s1,s2... are set expressions):

- [<Type-name-list>] where <Type-name-list> is a list of one or more primitive type names.
- s1 + s2 which yields the union of the two type sets.
- s1 * s2 which yields the intersection of the two type sets.
- s1 - s2 which yields the difference of the two type sets.
- <Set-name> which identifies a previously defined type set and yields its value.
- <Type-name> which identifies a primitive type and yields a singleton set with that type as its sole member.

4 OIL's Support Library

The library functions are grouped according to classes of functions. Within each class a C definition of the function is presented, followed by a brief description of the semantics of the function.

The next section describes the five C types used and defined by the library. These types are defined entirely by the functions in OIL's library.

The follow-on section describes operator identification using *set of possible types* which is the most general identification method supported by OIL.

Should a call to an OIL function for operator identification or coercion sequence construction fail, special values are returned. Functions are supplied to test for these special values allowing production of error messages for these error cases.

Besides operator identification using set of possible types, there is also a simpler identification algorithm which is strictly bottom up. It is useful when the full power of OIL is not necessary and the efficiency of a one pass algorithm can be utilized.

Looking at an operator's type signature is fundamental to propagating the constraints of an operator's type signature out from the node it labels. OIL supplies a function to examine any given operator's function signature,

Coercion sequences are fundamental to most uses of OIL and constructing and examining coercion sequences is critical to examining them for code generation purposes.

Creating instances of specified classes. OIL's *CLASS* construct allows for easy support of most type constructors. OIL allows a type constructor to be specified and then instantiated to create a type which conforms to the specification of the *CLASS*.

Names are an enumeration of the identifiers in the specification for easy comparisons. When a class is instantiated all its operators have new unique signatures based on the created type, but the names of corresponding operators are the same as those used in the specification of the class operator.

Construction of types, operators, identifications and classes can be performed by the library functions. Thus should a particular application need to build and define a unique class, coercion or any OIL object it can be done during the compilation using these functions.

4.1 Library Types

The semantics of the functions are described in terms of the basic types understood by OIL.

`tOilType` This type is associated with the identifiers defined as (*type*) in the OIL specification and represents type denotations. It is used to define function signatures for operators and thus the coercion graph.

`tOilOp` This type is associated with the identifiers defined as (*operator*) in the OIL specification. An element of this type can have associated with it either a function signature or a list of identifiable operators. All coercions are operators.

`tOilTypeSet`

Is a private type of the OIL library functions which represents a set of type denotations and represents the *set of possible types* concept.

`tOilClass`

This type is associated with identifiers defined as (*class*) in the OIL specification and identifies the set of operators and coercions defined for that class. It is used as the handle for instantiating a class. The instantiation operation creates a new object of type: `tOilType` and adds new instances of the operators and coercions defined for the class.

`tOilCoercionSeq`

This type represents a sequence of coercion operators which will transform a value of one type into a desired type. The coercions sequence may be empty or of an arbitrary length. Each element in a coercion sequence is a coercion operator.

There is one other type which is important for use of the ADT and that is the *name* of a class, type or operator. A *name* is represented as a definition table key (see Section 'The Definition Table Module' in Definition Table) and each identifier in the OIL specification has a unique *name* associated with it. This allows a class operator to be treated the same regardless of its arguments. But since the argument signature of an instantiated class operator will refer to the types used to instantiate the class, the type specific information can be referenced as needed.

4.2 Set of Possible Types

The functions `OilIdResultTS*` identify the set of possible result types given the operator indication (*oi*) and the set of possible result types (*ats**) for each operand.

The set of types (*tOilTypeSet*) returned describes the the union of the result types of any of the operators which can be identified by (*oi*) with any combination of argument types selected by the argument type sets, (*at*.*) Also in this set are types which can be reached from identified operators by means of a coercion sequence.

```
tOilTypeSet OilIdResultTS1( oi:tOilOp, ats:tOilTypeSet );
tOilTypeSet OilIdResultTS2( oi:tOilOp, ats1,ats2:tOilTypeSet );
tOilTypeSet OilIdResultTS3( oi:tOilOp, ats1,ats2,ats3:tOilTypeSet );
```

The functions `OilIdOpTS*` identify an operator given the operator indication (*oi*), the result type (*rt*) and the sets of possible argument types (*ats*.*)

```
tOilOp OilIdOpTS1( rt:tOilType, oi:tOilOp, ats:tOilTypeSet );
tOilOp OilIdOpTS2( rt:tOilType, oi:tOilOp, ats1,ats2:tOilTypeSet );
tOilOp OilIdOpTS3( rt:tOilType, oi:tOilOp, ats1,ats2,ats3:tOilTypeSet );
```

The function `OilTypeToSet` constructs a set of types from a given type denotation. The set of types returned contains *t* and all the type denotations which can be reached from *t* by any sequence of coercion operators.

```
tOilTypeSet OilTypeToSet( t:tOilType );
```

The function `OilSelectTypeFromTS` selects a type from a given set of types. The type selected is the type which is both in the set *ts* and can be coerced to all the types in the set.

```
tOilType OilSelectTypeFromTS( ts:tOilTypeSet );
```

The following equation is true for any type `t`:

```
OilSelectTypeFromTS( OilTypeToSet( t ) ) = t
```

The function `OilBalance` selects a type which can be coerced to all the elements which sets *ts1* and *ts2* have in common. This operation corresponds with function of type balancing in typed expression analysis from compiler and programming language theory.

```
tOilType OilBalance( ts1,ts2:tOilTypeSet );
```

The following is true for all type sets `ts1` and `ts2`:

```
OilBalance( ts1,ts2 ) = OilSelectTypeFromTS( ts1 AND ts2 )
```

One other important operation on sets is a test for set membership. This operation is performed on a type set by the function `OilSetIncludes`, which returns true if the set `s` includes the type `t`.

```
int OilSetIncludes( s:tOilTypeSet, t:tOilType );
```

`OilSetIncludes` will be the usual mechanism for testing if a expression can be coerced to a particular type. The expression's set of possible types is calculated and `OilSetIncludes` is used to check if the type in question is in the set of possible types.

4.3 Validating operator identification

The function `OilIsValidOp` validates that a given value denotes a valid operator. Since any operator identification operation will return some operator indication, we need to validate that the operator identified was not the catchall *illegal operator*.

```
int OilIsValidOp( op: tOilOp );
```

4.4 Simpler operator identification

The function `OilIdOp*` can identify the operator associated with the indication (*oi*) which has an argument type to which (*at*) can be coerced. In general these are less powerful than the 'set of result type' operators, but for simple languages they are both faster and easier to use. You can probably use these if you can identify the correct operator from the the types of the operands alone without regard to the context.

```
tOilOp OilIdOp1( oi: tOilOp, at: tOilType );
tOilOp OilIdOp2( oi: tOilOp, at1,at2: tOilType );
tOilOp OilIdOp3( oi: tOilOp, at1,at2,at3: tOilType );
```

4.5 Looking at an operator's type signature

The function `OilGetArgType` allows us to get the type of the n'th argument (*arg*) of an operator(*op*.) The 0'th argument returns the result type from the function signature.

```
tOilType OilGetArgType( op:tOilOp, arg:int );
```

4.6 Coercion sequences

The function `OilCoerce` allows us to construct a sequence of coercion operators from type *t1* to *t2*. The first operator in the sequence (see `OilHeadCS` below) will have a result type of *t2*. The last operator in the sequence will have a source type of *t1*. `OilCoerce` will always return a coercion sequence. But if there is no valid coercion sequence between the types then the catchall *error coercion sequence* is produced.

```
tOilCoercionSeq OilCoerce(  t1,t2:tOilType );
```

These operations on coercion sequences (`tOilCoercionSeq`) allow us to step through a coercion sequence and perform an action for each operator in the sequence.

The function `OilEmptyCS` will test a coercion sequence to see if it is empty. The result will be true is the argument is empty and false otherwise.

The function `OilHeadCS` returns the first operator in the sequence. The operator returned by `OilHeadCS` will have been defined by a coercion statement. Or it will be the error operator in the case of an *error coercion sequence*.

The function `OilTailCS` returns the rest of the sequence once the first operator in the sequence is removed.

```
int OilEmptyCS( cs: tOilCoercionSeq );
tOilOp OilHeadCS( cs: tOilCoercionSeq );
tOilCoercionSeq OilTailCS( cs: tOilCoercionSeq );
```

The function `OilIsValidCS` allows us to validate a coercion sequence. It is crucial to detect invalid typing for a subexpression since every call to `OilCoerce` will return a coercion sequence and we need to know if the sequence returned was the catchall *error coercion*.

```
int OilIsValidCS( cs: tOilCoercionSeq );
```

4.7 Instantiate Classes

When a class is instantiated a new type is created and the set of operators and coercions defined for that class are created using the created class and the types indicated by the parameters to build the actual function signatures for the created operators. The 'is coercible to' relation is enhanced by all the coercions defined by the instantiation.

Classes can be instantiated by calling one of the functions:

```
tOilType OilClassInst0( c:tOilClass, n:DefTableKey );
tOilType OilClassInst1( c:tOilClass, n:DefTableKey, at:tOilType );
tOilType OilClassInst2( c:tOilClass, n:DefTableKey, at1,at2:tOilType );
```

Constraints:

The number of parameters defined for the class must match the number of types supplied as arguments.

4.8 Name Functions

Each type, operator and class has a *name* associated with it. There is a function for retrieving the name associated with a specific type during attribution:

```
DefTableKey OilTypeName( t:tOilType );
DefTableKey OilOpName( op:tOilOp );
DefTableKey OilClassName( c:tOilClass );
```

If the identifier `MyType` is a type in a specification then the C symbol `MyType` will have the value of `OilTypeName(MyType)`.

4.9 Compile Time

The OIL library has all of the necessary functions for the construction of OIL entities during the execution of the generated compiler. This capability allows the changing of the OIL schema in more detailed ways than simply instantiating an already specified class. The different capabilities for modifying the schema are:

Type Constructor
: A new type may be constructed without using Classes.

Operator Constructor
: A new operator may be explicitly constructed with a given signature(remember operators are use to represent indications also.)

Signature Constructor
: An argument signature for an operator can be constructed from types.

Coercion Constructor
: A properly constructed operator can be declared to be a coercion.

Identification Constructor
: Any operator can be defined to 'indicate' any other operator.

Class Constructor
: Can be built from class operators and coercions.

4.9.1 Types

In addition to class instantiation, a new type may be constructed with the function `OilNewType`. Its only argument is the name to be associated with the new type.

```
tOilType OilNewType( id:DefTableKey );
```

4.9.2 Operators

Constructing a new operator is a two step process, first a new argument signature must be constructed and then a new operator with that signature can be constructed using the function `OilNewOp`. Besides the argument signature, `OilNewOp` requires the name of the new operator(`id`), and the cost of the new operator(`cost`.)

```
tOilOp OilNewOp(id:DefTableKey,sig:tOilArgSig,cost:int);
```

4.9.3 Argument Signatures

Argument signatures are built in two steps: an empty signature is constructed with `OilNewArgSig` and then a type is pushed onto the front of the signature using `OilAddArgSig`.

```
tOilArgSig OilNewArgSig(dummy:int);
tOilArgSig OilAddArgSig( arg:tOilType, sig:tOilArgSig );
```

Note that by convention, the last type pushed onto the signature is the result type of the created operator.

4.9.4 Coercions

Any operator with an argument signature of length 2 can be a coercion by simply applying `OilAddCoercion` on it.

```
int OilAddCoercion( op:tOilOp );
```

Constraints

A check is not made that the signature is of length 2.

4.9.5 Identifications

A relationship between an operator indication (`ind`) and an operator (`op`) is established by simply supplying them to the `OilAddIdentification` function.

```
int OilAddIdentification( ind, op:tOilOp );
```

Constraints

A check is not made regarding the redundancy of the new identification with respect to the existing schema. You can have ambiguity of which operator is identified by any given indication and operand signature. OIL will choose the most recently declared, least cost identification.

4.9.6 Classes

Classes are very complex entities and are constructed in stages. First an empty class in created using `OilNewClass`. The argument `id` specifies the name of the class and the argument `argNum` specifies how many parameters the class has.

```
tOilClass OilNewClass(id:DefTableKey,argNum:int);
```

To an existing class(`c`) we can add an operator with a given class signature(`sig`) and given cost(`cost`) with the function `OilAddClassOp`.

```
tOilClassOp OilAddClassOp(id:DefTableKey,sig:tOilClassArgSig,cost:int, c:tOilClass);
```

With a class operator we can create an identification of an instantiated class operator(`op`) by an existing operator(`ind`) with the function `OilAddClassOpId`.

```
int OilAddClassOpId(ind:tOilOp,op:tOilClassOp);
```

The function `OilAddClassCoercion` is used to define an existing class operator(`op`) to be a coercion.

```
int OilAddClassCoercion(op:tOilClassOp);
```

Building a class argument signature is similar to constructing a simple argument signature but it is complicated by the fact that a class argument needs to be described in terms of a parameter binding. A class argument's parameter binding determines the value of the parameter when the class is instantiated. Like simple signatures we first build an empty class signature and then push arguments onto it. An empty class signature is created with `OilNewClassSigArg` and an argument description is added with `OilAddClassSigArg`.

```
tOilClassArgSig OilNewClassSigArg(dummy:int);
```

```
tOilClassArgSig OilAddClassSigArg(
    td:tOilClassSigArgDesc,
    st:tOilType,
    pi:int,
    cs:tOilClassArgSig
  );
```

The extra arguments to `OilAddClassSigArg` describe the possible bindings which will instantiate the class signature.

`td` Can be `eClassRef` to indicate that a reference to the created type replaces this argument, `eParamRef` to indicate that one of the parameters to the class instantiation will replace this argument or `eSpecTypeRef` to indicate that a specific type will replace this argument.

`st` Specifies which explicit type will replace this argument.

`pi` Selects which parameter of the class instantiation will replace this argument.

5 Relating an OIL specification to library function calls

To explain the relationship between the specification and the abstract data type we will examine different extractions from some possible specifications and review the behavior of some of the related functions in the abstract data type.

5.1 Using Names

Each entity defined in an OIL specification is represented by a definition table key. The OIL value is accessible as a property of that definition table key. For example, suppose that `iAdd` was defined by an `OPER` statement in OIL. This would result in a known key (see Section 'How to specify the initial state' in Definition Table) `iAdd`. `iAdd` would also have an `OilOp` property whose value was the actual OIL operator (of type `tOilOp`). Thus the actual OIL operator corresponding to `iAdd` could be obtained by the function call `GetOilOp(iAdd,OilErrorOp())` (see Section 'Behavior of the basic query operations' in Definition Table).

In order to avoid the overhead of querying a property for constant information, OIL also defines an identifier as the actual OIL operator. This identifier is constructed by prefixing the OIL identifier with `OilOp`. Thus the identifier `OilOpiAdd` denotes the value that would be obtained from the function call `GetOilOp(iAdd,OilErrorOp())`.

Similar conventions are used for OIL types and OIL classes: The OIL identifier denotes a known key, and the actual OIL entity (of type `tOilType` or `tOilClass` respectively) is denoted by prefixing either `OilType` or `OilClass` to that identifier. A known key denoting an OIL type also has an `OilType` property, and a known key denoting an OIL class has an `OilClass` property.

5.2 A simple example

Let us consider the following OIL specification:

```
iAdd ( int_t, int_t ): int_t;     /* the usual '+' operators for Pascal */
rAdd ( real_t, real_t ): real_t;
sUnion ( set_t, set_t ): set;

Plus: iAdd, rAdd, sUnion;  /* will be identified together */
```

```
COERCION Float( int_t ): real_t;    /* usual Pascal coercion from int to real
*/
```

5.2.1 Definitions from the specification

All of the identifiers in this specification will denote values to the library functions. The functions in the library will be applied to values constructed from these identifiers and will return values represented by these identifiers.

5.2.2 Operator Identification

The most basic operation is that of operator identification so we will start there. When semantically analyzing a binary expression formed with a plus sign (`+`), the compiler would use the function `OilIdOp2` applied to the value denoted by *Plus* (which indicates the syntactic operator) and the types of the operands to the plus sign.

The invocation `OilIdOp2( Plus, int_t, real_t )` would return the operator *rAdd* because *int_t* was coercible to *real_t*. Similarly :

`OilIdOp2( Plus, set_t, set_t )` would return `sUnion`

`OilIdOp2( Plus, int_t, int_t )` would return `iAdd`

`OilIdOp2( Plus, real_t, real_t )` would return `rAdd`

Any combination of operand types like *real_t* and *set_t* would return a value denoting an erroneous operator. Example: `OilIsValidOp( OilIdOp2( Plus, real_t, set_t ) )` would return an integer value of 0.

5.2.3 Operator Signatures

Once we have identified an operator will need to know its type signature so that we may return the type of the subexpression computed by the operator and so we may determine the types required by the operator from its respective operands. The function `OilGetArg` gives us that facility.

The expression `OilGetArg( iAdd, 0 )` would return *int_t* as the type of the result of the operator *iAdd*. Likewise `OilGetArg( sUnion, 1 )` would return *set_t* as the required type of the first operand to the 'sUnion' operator.

5.2.4 Coercion sequence

Once we have the type returned by an operator and know the type required of this sub-expression (from the parent context) we may need to apply a sequence of coercions on the result of the operator to satisfy the requirements of the parent context. The function `OilCoerce` supplies the necessary function.

In the case of our example we might require a *real_t* result from an *iAdd* operator (which returns *int_t*.) The expression `OilCoerce( int_t, real_t )` would return a coercion sequence which represented the coercion of an *int_t* type value to a *real_t* type value. This coercion sequence (call it *cs*) would then be analyzed with the functions: OilEmptyCS, OilHeadCS and OilTailCS. The expression `OilHeadCS( cs )` would evaluate to *Float* and `OilEmptyCS( OilTailCS( cs ) )` would evaluate to true. These expressions describe the fact that only the coercion operator *Float* was necessary to transform *int_t* to *real_t*.

If no coercions were necessary then `OilEmptyCS( cs )` would have yielded the value true. Likewise to detect an impossible coercion, the function `OilIsValidCS` would be used. The expression `OilIsValidCS( OilCoerce( real_t, set_t ) )` would yield the value false to indicate that such a coercion was not possible.

5.3 A more complex example

Not all operator identification schemes can be implemented with the simple bottom-up type evaluation shown in the previous section. Sometimes the desired result type will affect which operator denotation is identified with a given operator indication. OIL supplies this capability with the **set of types** operations.

Below is an example OIL specification which is designed to use **set of types.** The specification shows that there are two multiplication operators(`sMulS` `sMulD`) on type `single`. One multiplication operator returns a double length result (`double`) the other returns a single length result(`single`.) These declarations have a natural correspondence with many machine architectures. The operator indication `Mul` is defined to identify either `sMulS` or `sMulD`.

```
sMulS ( single, single ): single;
sMulD ( single, single ): double;
dMulD ( double, double ): double;

COERCION iCvtStoD ( single ): double;

Mul: dMulD, sMulD, sMulS
```

5.3.1 Using type sets

To use *type set* functions we must begin with constructing the *possible result type set* of a terminal. For this we use the function `OilTypeToSet`. Like so:

`OilTypeToSet( single )` yields `[ single, double ]`

`OilTypeToSet( double )` yields `[ double ]`

For the rest of this example we will use the identifiers `ss` and `ds` to represent the type sets for single and double, respectively.

To analyze an entire expression with *type sets* we must also be able to determine the set of types associated with an operator indication and its set of operands. For this we use the `OilIdResultTS*` functions. Like so:

`OilIdResultTS2( Mul, ss, ss)` yields `[ single, double ]`

`OilIdResultTS2( Mul, ds, ss)` yields `[ double ]`

`OilIdResultTS2( Mul, ds, ds)` yields `[ double ]`

When we get to the root of an expression (like in an assignment) we would then use a desired type determined from the context of the root of the expression (like the destination type of the assignment) to determine which operator we wanted to select. For this we use the `OilIdOpTS*` functions. Like so:

`OilIdOpTS2( single, Mul, ss, ss)` yields `sMulS`

`OilIdOpTS2( double, Mul, ss, ss)` yields `sMulD`

`OilIdOpTS2( double, Mul, ds, ss)` yields `dMulD`

By using *type sets*, the operator indication *Mul* with *single* operands can identify *sMulD*, thus directly producing a double result; whereas with the simple scheme used previously (see Section 5.2.4 [Simple Example], page 29.) an additional coercion would be needed to return a double result.

5.4 Using Classes

There are three steps to using classes: (1)specifying them with OIL, (2)instantiating them using the `OilClassInst*` functions and (3)identifying the enriched indication mappings with enriched coercion graph.

The following OILspecification allows us to define any number of `Set`s during compilation. And we specify the overloading of the '`+`'(`loPlus`) operator to allow set union.

```
CLASS Set( element ) BEGIN
  COERCION coElemToSet(element):Set;
  OPER soUnion(Set,Set):Set;
END;

OPER soIadd(tInt,tInt):tInt;
OPER soRadd(tReal,tReal):tReal;

INDICATION loPlus: soIadd, soRadd, soUnion;
```

We can then construct a simple binary expression compiler which uses a constant set for one of its possible operand types.

```
NONTERM Expr: op:tOilOp, st:tOilType;

RULE Dyadic: Expr::= Term Ind Term
STATIC
  Expr.st:= OilClassInst1( Set, Set_name, tInt );
  Expr.op:= OilIdOp2( Ind.op, Term[1].type, Term[2].type )
END;

NONTERM Term: type:tOilType;

RULE Set: Term::= 's'
STATIC Term.type:= INCLUDING Expr.st END;

RULE Integer: Term::= 'i'
STATIC Term.type:= tInt END;

RULE Real: Term::= 'r'
```

```
STATIC Term.type:= tReal END;

NONTERM Ind: op:tOilOp;

RULE Plus: Ind::= '+'
STATIC Ind.op:= loPlus END;
```

We use the following request to construct the compiler:

```
test3.specs :exe>test3.exe
```

6 Design Hints

Some 'usual' problems in operator identification are presented with some suggested work-arounds using OIL(along with general guidelines for effective use of OIL.)

6.1 Incremental Design and Implementation

It is relatively easy to design and implement with OIL in incremental steps. One can work with the atomic/primitive types of your schema first and get the desired behavior with only identification. You can then add coercion sequence construction. And then work on the class specifications. Last of all one should work on compile time entity definition. These guide lines can be easily ignored but I would suggest that you find replacements for these rules rather than not do incremental design and implementation.

6.2 Identifying Operands

For some problems it is easier to identify the operand types and associated coercion sequences independently of the operator. When using OIL to select addressing modes for assembly language this is often the case as the instruction set is factored into operation and operand address. Operations can be factored into classes which support different operand address mode signatures. The classes can be defined as an indication and the different patterns of address modes are the strongly typed data flow operators for the instruction.

If we didn't use this method of 'instruction classes' then we would have to duplicate the address mode patterns for each instruction in the class. This would be time consuming, redundant, and a strain on our name generation faculties. And such a multiplicity of names would cause its own confusion, reducing the benefit of using OIL.

6.3 Error Handling

There are two ways to handle errors with OIL: use your own error type and use OIL's. Each has its own advantages.

OIL's error type is `OilErrorType` and is by definition coercible to and from any type. Thus once an expression is assigned this value it will make an end-run on OIL's strong typing and match any operator which satisfies the other argument constraints. In the degenerate case where all operands are `OilErrorType` then any identifiable operator will be chosen.

If your own type is used(say `ErrorType`) you must define an operator with it in its signature(the only way to define a type other than class instantiation.) By not having any type coercible to it no operator will be identified and thus no valid operator will be returned and the function `OilIsValidOp` will return false allowing easy error detection.

The error operator(`OilErrorOp`) has a type signature of all `OilErrorType`. Thus if it is identified, which in the error case it is, no type errors will be propagated up or down the tree.

6.4 Modeling OIL's function

If you need to model OIL in your compiler design, one of the most convenient ways is to consider the set of declarations which the OIL library manages as a database. This data base is initialized by the OIL specification and modified and accessed via the library functions.

6.5 Schema Restrictions

OIL has a very simple schema model. All of the library routines only add declarations to an OIL schema. There is no way to remove a declaration from an OIL schema once it has been added. This can have two impacts on your use of OIL. Every added declaration increases the cost of an identification which it *may* impact. Once an identification is added to the schema it may satisfy an identification request, the *only* way to prevent it is to control what kinds of requests are made.

6.6 Identification Algorithm

OIL has a two level search strategy: minimum cost identification and if costs are equal the most recent identification added to the schema.

Index

A

abstract data type ... 27
argument type ... 19, 21

B

balancing ... 19

C

class definition ... 10, 15
classes ... 9
coercion ... 7, 13, 19, 29
coercion operator ... 19
coercion operator definition ... 10, 13
coercion sequence ... 18, 21, 29
coercion sequence, empty ... 21
coercion sequence, error ... 21
coercion sequence, head of ... 21
coercion sequence, tail of ... 21
creating new names ... 22

D

double ... 29

E

empty coercion sequence ... 21
error coercion ... 21
error coercion sequence ... 21
expected argument type ... 21

H

head of coercion sequence ... 21

I

identifier ... 7
illegal operator ... 20
impossible coercion ... 29
incremental design ... 33

L

library functions ... 17

M

multiple operator definition ... 11

N

names ... 5, 22, 27

O

OIL comments ... 10
OIL identifiers ... 9
OIL library ... 17
OIL specification ... 9
OIL statements ... 9, 10
OilAddArgSig ... 24
OilAddClassCoercion ... 25
OilAddClassOp ... 25
OilAddClassOpId ... 25
OilAddClassSigArg ... 25
OilAddCoercion ... 24
OilAddIdentification ... 24
OilBalance ... 20
OilClassInst0 ... 22
OilClassInst1 ... 22
OilClassInst2 ... 22
OilClassName ... 22
OilCoerce ... 21
OilEmptyCS ... 21
OilGetArg ... 29
OilGetArgType ... 21
OilHeadCS ... 21
OilIdOp1 ... 20
OilIdOp2 ... 20
OilIdOp3 ... 20
OilIdOpTS1 ... 19
OilIdOpTS2 ... 19
OilIdOpTS3 ... 19
OilIdResultTS1 ... 19

OilIdResultTS2 ... 19
OilIdResultTS3 ... 19
OilIsValidCS ... 22
OilIsValidOp ... 20
OilNewArgSig ... 24
OilNewClass ... 25
OilNewClassSigArg ... 25
OilNewOp ... 23
OilNewType ... 23
OilOpName ... 22
OilSelectTypeFromTS ... 19
OilSetIncludes ... 20
OilTailCS ... 21
OilTypeName ... 22
OilTypeToSet ... 19
operator definition ... 10, 11
operator denotation ... 7, 29
operator identification ... 10, 14, 28
operator indication ... 7, 19, 29
operator signature ... 28
operators ... 9
overloaded operators ... 7

P

PASCAL ... 7, 28
possible result types ... 19

R

result type ... 19, 21

S

selected type ... 19
sequence of coercions ... 21, 29
set of types ... 18, 19, 29
signature ... 12, 14, 21, 28
single ... 29
source type ... 21
support library ... 17

T

tail of coercion sequence ... 21
tOilArgSig ... 24
type balancing ... 19
type denotation ... 7, 19
type set ... 9, 11, 18, 30
type set definition ... 15
type signature ... 21
types ... 9

V

valid operator ... 20
validating a coercion sequence ... 21
validating operator identification ... 20