Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 17-009

Reliable and automatic composition of language extensions to C - Supplemental Material

Ted Kaminski, Lucas Kramer, Travis Carlson, Eric Van Wyk

September 5, 2017

Reliable and automatic composition of language extensions to C

— Supplemental Material —

Ted Kaminski, Lucas Kramer, Travis Carlson, Eric Van Wyk
University of Minnesota
tedinski@cs.umn.edu, krame505@umn.edu, carl4980@umn.edu, evw@umn.edu

This technical report provides a more complete description of many of the ableC language extensions described in our OOPSLA 2017 paper "Reliable and automatic composition of language extensions to C" [6]. It describes several additional ableC language extensions that pass the modular analyses and thus can be reliably and automatically composed together by a programmer to form a working translator for a custom extended language. The OOPSLA paper describes extensions that illustrate many of the types of language extensions that can be specified as ableC extensions and is thus self contained. Yet, the extensions discussed here expand on these capabilities and describe extensions that are substantially larger in scope or illustrate additional capabilities of ableC and the underlying tools Silver and Copper.

(1) SQL: this extension provides constructs for working with SQLite databases. In addition to the extended code appearing more natural than direct use of library functions, the extension is able to statically detect syntax and type errors in SQL queries.

(2) Halide: this extension is inspired by the Halide system [8]. Like Halide, it allows programmers to specify a sequence of optimizing transformations that the extension performs on, typically, array processing code written as nested for-loops.

(3) Boolean-valued condition tables: these demonstrate how embedded DSLs can specify their own white-space and comment conventions. In this case, line breaks are part of the context-free syntax and identify the separation between table rows.

(4) Templates: this extension makes use of the lifting mechanism described in section 7.2 to add declarations of instantiated functions and structures at the top level of the program. Templated functions and types in the style of C++ are introduced which enable type-generic code with better guarantees of type safety. Beyond its usefulness to library writers and end users, the extension provides a mechanism for extension writers to easily implement parametric types and functions to be used for forwarding.

(5) Vectors: this extension makes use of the template extension and also shows how extensions can be layered on top of one another. Standard vector operations and Python-inspired mutation operations are implemented via the operator overloading mechanism described in section 7.1.

(6) C++ type-safe enumerated type: this extension uses type qualifiers described in section 7.3 to introduce a type-safe enumeration construct, which prevents arbitrary integers from being assigned to a variable of enumerated type. C++ performs such checks by changing the semantics of enumerated types. Here we show that similar behavior can be obtained purely as an extension to C, with no changes required to the semantics of the host language other than those required to support extensible type qualifiers.

(7) Lambda-expressions: this is another example of an extension that lifts declarations to the top level as described in section 7.2. This extension introduces closures, anonymous functions that capture the scope in which they were created.

Other extensions include MATLAB style matrix operations such as array slicing and the generation of MEX functions that can be called directly from MATLAB, Go-style concurrency primitives, and constructs for array computations from Single Assignment C (SAC) [4].

Additionally, we have created a language extension that expands significantly upon the algebraic data type extension discussed in the paper. This provides a flexible mechanism for specifying such types and term-writing rules that can be applied to values of these types. In this regard it implements many of the features in Tom [7, 1] but as a composable language extension. Additionally it implements many primitive term-rewriting strategies and constructs for combining them to form complex rewriting strategies in the style of those in Stratego [10]. A technical report discussing this extension can be found in the *non-anonymous* supplemental material.

1 Database access based on the SQLite library

SQL is a domain-specific language that can be naturally implemented as an extension. Although libraries that provide functions for working with databases exist, the direct use of such functions is error-prone and unnatural as SQL queries are often passed as simple strings to library functions. Here we present an extension that introduces new syntax for working with SQLite databases.

Figure 1 illustrates the use of the SQLite extension through a sample program that connects to a database, inserts rows into the database, performs a non-trivial query, and loops over the selected rows of the query. Code written by the extension user is shown on the left-hand side of the figure, and the corresponding generated code is shown on the right-hand side. It is important to note that, although the run-time behavior of the two programs is identical, the extended code is able to perform additional compile-time error checking.

A variable that connects to a database of a given schema is declared with a new marking terminal `use` (line 5). A schema consists of one or more named tables, each in turn containing one or more named columns of type `INTEGER` or `VARCHAR`. This schema information is completely dropped in the generated code and is used only for static error checking — we will see later that a query that refers to a table or column that does not exist in the schema will raise a compile-time error. The generated code consists simply of a declaration of a variable of type `struct _sqlite3_db_s *` initialized using `new_sqlite_db()`, both of which are declared in `sqlite.xh`.

The included `sqlite.xh` declares various wrappers around structures and functions of the SQLite library intended to simplify the generated code. For example, `struct _sqlite3_db_s` wraps `sqlite3_stmt`, is initialized with `_new_sqlite_db()` which calls `malloc()` and `sqlite3_open()`, and is closed with `db_exit()` which calls `sqlite3_close()` and `free()`. Similar wrappers exist for queries, *e.g.* `struct _sqlite_query_s` wraps `sqlite3_stmt`.

Queries begin with `on db query` (line 20) and forward to a declaration of a `struct _sqlite_query_q` followed by a call to `sqlite3_prepare_v2()`. The results of the query can then be iterated over using `sqlite3_step()` and committed using `finalize()` which checks the status of `sqlite3_finalize()` and frees the memory allocated for the query struct. For convenience, this finalization can be performed immediately using `on db commit` (line 14).

The concrete syntax of queries are based on standard SQL syntax. This allows ill-formed queries to be detected at compile time, which would not otherwise be possible until run time since the generated queries are represented as regular C strings (line 12 of the generated code). Such checks are not limited only to syntax errors; using information from the database schema it is possible to detect attempts to access tables that do not exist in the database or rows that do not exist in a certain table. To allow the use of arbitrary embedded C expressions (line 24), the query language has been extended slightly; where an SQL expression is allowed, we additionally allow `$( C_EXPR )`. Rather than insert a string representation of the embedded C expression directly into the generated query string, a '`?`' is inserted which specifies a parameter to be bound with a call to `sqlite_bind_*()` (line 14 of the generated code) and provides protection from injection attacks. Although it is possible to write similarly safe code through direct use of the SQLite library functions, note that it is impossible to write unsafe code using the extension; rather than educating and trusting users to follow best practices, we can now simply rely on guarantees provided by the language.

Another new marking terminal `foreach` (line 28) is used for iterating over the results of a query. Each iteration of the loop handles one row of the results through the generation of a `while (sqlite3_step() == SQLITE_ROW)` loop. An anonymous structure (lines 20–28 of the generated code) is generated containing a field for each result column. The fields' names match the selected columns of the query and, in each iteration of the loop, are automatically initialized to the correct column number. Direct use of the SQLite library functions require columns to be specified by index and type, which is an error-prone process to maintain manually.

This extension could possibly be improved in several ways. Consider that a schema declaration (line 6) is reminiscent of a struct; it consists of a list of columns of a given type. Perhaps SQL schemas could be more closely integrated with C data structures by parameterizing tables by structs rather than explicitly specifying each column, *e.g.* line 6 could be replaced with `table person (struct person_t)` where `struct person_t` contains `person_id`, `first_name`, and `last_name` fields of appropriate type. This might make additional useful constructs possible. Perhaps a table could be populated directly from an array of structs, *e.g.* `struct person_t c_people[] = {... }; on db populate {TABLE person FROM c_people};`

```
1.  #include <stdio.h>
2.  #include <sqlite.xh>
3.  int main(void)
4.  {
5.    use "test.db" with {
6.      table person  ( person_id  INTEGER,
7.                      first_name VARCHAR,
8.                      last_name  VARCHAR ),
9.      table details ( person_id  INTEGER,
10.                     age        INTEGER,
11.                     gender     VARCHAR )
12.   } as db;
13.
14.   on db commit { INSERT INTO person
15.                  VALUES (0, 'Aaron', 'Allen') };
16.   on db commit { INSERT INTO details
17.                  VALUES (0,  5, 'M') };
18.
19.   const char except_surname[] = "Adams";
20.   on db query {
21.     SELECT   age, gender, last_name AS surname
22.     FROM     person JOIN details
23.     ON       person.person_id = details.person_id
24.     WHERE    surname <> $(except_surname)
25.     ORDER BY surname DESC
26.   } as people;
27.
28.   foreach (person : people) {
29.     printf("%10s %2d %s\n",
30.             person.surname, person.age,
31.             person.gender);
32.   }
33.   finalize(people);
34.   db_exit(db);
35.   return 0;
36. }
```

```
1.  // expand <stdio.h>
2.  // expand <sqlite.xh>
3.  int main(void)
4.  {
5.     struct _sqlite3_db_s *db
6.       = new_sqlite_db("test.db");
7.     ... // commits elided
8.     const char except_surname[] = "Adams";
9.     struct _sqlite_query_s *people
10.      = new_sqlite_query();
11.    sqlite3_prepare_v2(db->db,
12.      "SELECT age, gender, ... "
13.      160, people, 0);
14.    sqlite3_bind_text(people->query, 1,
15.      except_surname, -1, 0);
16.
17.    sqlite3_reset(people->query);
18.    while (sqlite3_step(people->query)
19.             == SQLITE_ROW) {
20.      const struct {
21.        int age;
22.        const char *gender;
23.        const char *surname;
24.      } person = {
25.   sqlite3_column_int(people->query, 0),
26.   sqlite3_column_text(people->query, 1),
27.   sqlite3_column_text(people->query, 2),
28.      };
29.      printf("%10s %2d %s\n",
30.              person.surname, person.age,
31.              person.gender);
32.    }
33.   finalize(people);
34.   db_exit(db);
35.   return 0;
36. }
```

Figure 1: Sample program utilizing SQLite extension, with generated code.

2 Parallel and vector processing inspired by Halide

When developing performance-critical code, there are a number of optimizing transformations that may be applied to iterative constructs. These may come at the cost of code clarity, difficulty in trying different combinations of transformations, and the potential to introduce errors in this process. Halide [8] is a tool that allows for the semantics of an essentially parallel algorithm to be written simply, and then a schedule for how to translate the algorithm into some set of parallel and sequential constructs. This extension is inspired by Halide, in that a simple iterative structure can be written, followed by a list of transformations. These are essentially the same as those provided by Halide, but can be expressed more clearly with a custom DSL than when constrained to C++ syntax.

This extension introduces a new form of statement,

```
transform { <iteration_statements> } by { <transformations> }
```

which can be broken down into two main components: a DSL for specifying a set of nested loops, and a DSL for specifying transformations to perform on these loops.

The concrete syntax for iteration statements is similar to that of statements in ableC. An iteration statement may consist of a block of other iteration statements, a for loop construct (which may specify multiple nested loops), or an expression followed by a semicolon. We refer to the variables defined in a loop as *iteration variables*, each of which uniquely identifies what would become a C for loop in the untransformed translation.

For example, in lines 3–5 of Fig. 2, we perform a gradient computation with two nested loops, named by iteration variables x and y, where x is the outer loop and y the inner loop. The iteration variables range between 0 and the provided cutoff, which is 10. This loop definition is simply syntactic sugar for two separate, nested for loop definitions.

In lines 9–14 of Fig. 3, we perform a matrix multiplication. Here we use non-constant expressions for the loop cutoffs, and nest an additional loop within the body of the inner loop.

For specifying transformations, an embedded DSL is also used, listing a series of transformations referencing iteration variables defined in the initial iteration statements. Some of these transformations may contain host language expressions as parameters. For example, the transformations shown in Fig. 2 perform a series of loop split and reorder operations that constitute a loop tiling. A more complex example that also uses the `parallelize`, `vectorize`, and `unroll` transformations to perform an optimized matrix multiplication can be found in Fig. 3

```
1.  int result[10][10];
2.  transform {
3.    for (unsigned x : 10, unsigned y : 10) {
4.      result[x][y] = x + y;
5.    }
6.  } by {
7.    split x into (unsigned x_outer,
8.                  unsigned x_inner : 4);
9.    split y into (unsigned y_outer,
10.                 unsigned y_inner : 4);
11.   reorder x_outer, y_outer, x_inner, y_inner;
12. }
```

Figure 2: An example program segment that performs a tiled gradient computation, using a composable language extension providing halide-like separate specification and transformation of iterative constructs.

These transformations are implemented by essentially 'threading' the AST from the initial specification through each transformation. This is done using a pair of synthesized and inherited attributes, `toTransform` and `transformed`, both with type `IterStmt`. Each transformation production decorates the incoming `IterStmt` with inherited attributes providing parameters for the transformations, and then uses a synthesized attribute specific to each transformation to reconstruct the transformed version of the tree, which is then passed out as `transformed`.

For `parallelize` and `vectorize`, the AST is simply reconstructed with the named abstract loop replaced with an explicit statement for a C loop, with an attached OpenMP pragma. The `split` transformation, such as in lines 7 and 8 of Fig. 2, works in a similar way by replacing a loop with a series of new, nested loops. When the loop cutoffs for the new and original loops are constant, the new cutoffs are calculated statically.

For `unroll`, the named loop is replaced by a series of C statements, each containing an initialization of the loop variable to a constant and the body of the loop. Note that the loop cutoff here must be a constant.

```
1.  void matmul(unsigned m, unsigned n, unsigned p, float a[m][p], float b[p][n], float c[m][n]) {
2.    transform {
3.      for (unsigned i : m, unsigned j : n) {
4.        c[i][j] = 0;
5.        for (unsigned k : p) {
6.          c[i][j] += a[i][k] * b[k][j];
7.        }
8.      }
9.    } by {
10.     split i into (unsigned i_outer, unsigned i_inner : (m - 1) / NUM_THREADS + 1);
11.     parallelize i_outer into (NUM_THREADS) threads;
12.     tile i_inner, j into (TILE_DIM, TILE_DIM);
13.     split k into (unsigned k_outer
14.                   unsigned k_unroll : UNROLL_SIZE,
15.                   unsigned k_vector : VECTOR_SIZE);
16.     unroll k_unroll;
17.     vectorize k_vector;
18.   }
19. }
```

Figure 3: An example program segment that performs a tiled gradient computation, using a composable language extension providing halide-like separate specification and transformation of iterative constructs.

This transformation can also be implemented similarly to `parallelize`, using a synthesized attribute to reconstruct the tree down to the point at which the new, unrolled version is generated.

Performing a `reorder` transformation is more complex; this requires an inherited attribute to tell the AST what loops are to be transformed, and a synthesized attribute to extract the names and cutoffs of these adjacent loops into a sort of environment. The new AST can then be constructed by a function that looks up each loop in order and reconstructs it around the body of the innermost loop. A `tile` transformation simply consists of a sequence of `split`s followed by a `reorder` transformation, as seen in Fig. 2. This can be accomplished by having this transformation forward to the corresponding sequence of component transformations

This extension could be improved in many ways with new analyses and constructs. For example, thread-launching and vector operation code could be generated explicitly without relying on OpenMP. Another important feature would be handling of synchronization and avoiding introduction of race conditions when parallelizing loops - this could be done by adding additional analyses, and auto-generating atomic operations and/or synchronization code.

3 White-space sensitive parsing: condition tables

This extension provides constructs, based on those of RSML^{-e} [9] and SCR* [5], that are useful for understanding complex boolean expressions. This demonstrates how embedded DSLs can specify their own white-space and comment conventions.

A single new marking terminal `table` is used to begin a table expression. Rows of the table are separated by newlines; the ignored white space between rows is only spaces, as indicated by the `layout` clause.

A condition table expression forwards to a boolean value that is computed as follows. For each column to the right of the colon, a conjunction of each expression to the left of the colon is formed, with the expression taken directly if the truth value is T, the expression being negated if the truth value is F, and the expression being replaced with 1 (true) if the truth value is `*`. The final value is computed as the disjunction of these conjunctions.

```
1.  concrete production tableRowSnoc_c
2.  top::TableRows_c ::= trowstail::TableRows_c
3.                       n::NewLine_t
4.                       trow::TableRow_c
5.  layout { Spaces_t }
6.  {
7.    top.ast = tableRowSnoc(trowstail.ast,
8.                           trow.ast) ;
9.  }
```

Figure 4: Line breaks are part of the context free syntax and identify the separation between table rows.

Figure 5 illustrates the use of condition tables through a simple example. Lines 6–8 use a condition table to specify the following policy. Suppose an apartment complex's pet policy allows 1) cats, 2) other animals weighing under 20 pounds, and 3) any other animal provided that an additional fee is paid. Each of these conditions is specified by the respective column of the table. The generated code avoids evaluating the expressions more than once by using GCC's statement expression. The result of the expression is then a boolean expression in disjunctive normal form.

```
1.  #include <stdio.h>
2.  int main(void) {
3.    int is_cat = 0;
4.    int weight = 16;
5.    int paid_extra_fee = 0;
6.    if (table { is_cat          : T F F
7.                weight < 20     : * T T
8.                paid_extra_fee : * * T })
9.       )  {
10.
11.
12.
13.
14.     printf("Pet is allowed.\n");
15.   } else {
16.     printf("Pet is NOT allowed.\n");
17.   }
18.   return 0;
19. }
```

```
1.  // expand <stdio.h>
2.  int main(void) {
3.    int is_cat = 0;
4.    int weight = 16;
5.    int paid_extra_fee = 0;
6.    if ({
7.      _Bool __table_cond_7 = is_cat;
8.      _Bool __table_cond_8 = weight < 20;
9.      _Bool __table_cond_9 = paid_extra_fee;
10.     (( __table_cond_7 && 1             && 1) ||
11.      (!__table_cond_7 && __table_cond_8 && 1) ||
12.      (!__table_cond_7 && __table_cond_8 &&
13.        __table_cond_9)); }) {
14.     printf("Pet is allowed.\n");
15.   } else {
16.     printf("Pet is NOT allowed.\n");
17.   }
18.   return 0;
19. }
```

Figure 5: Sample program utilizing Condition Tables extension, with generated code.

4 Templates

This extension introduces C++-style templated functions and types, to provide a mechanism for polymorphism other than `void*` pointers. The purpose of this extension is twofold. Firstly, this extension allows library writers and end users to be able to write type-generic code with better guarantees of type safety. Secondly, for extension writers, such as for the vector extension, are given a mechanism to easily implement parametric types and functions to be used for forwarding, in a simpler way than otherwise, such as what is done by the closure extension.

The concrete syntax of this extension is heavily inspired by C++. New function and struct definitions are implemented, mirroring C functions and structs, except prefixed by `template<params>`, as seen in Fig. 6. In addition, for template type aliases, such as on line 8 of Fig. 7, the C++11 syntax `using name<params> = type;` is also borrowed. For template instantiation, the syntax `template name<args>` for type expressions and `inst name<args>` for expressions is similar to C++, with the addition of the marking terminals. Note that Copper, using context-aware scanning, can automatically handle the issue of '`>>`' in template instantiations, which requires special handling in most other implementations.

```
1.  template<a> struct loc {
2.    a x;
3.    a y;
4.  };
5.
6.  template<a> a distance(template loc<a> p, template loc<a> q) {
7.    return (a)sqrt((p.x - q.x) * (p.x - q.x) + (p.y - q.y) * (p.y - q.y));
8.  }
```

Figure 6: An example definition of a templated struct that implements a location, and a function to compute the distance between two locations.

The semantics of templates are also based on C++. A templated struct defines a new name in the template namespace, next to templated function names. Template instantiation performs semantic checking and reports errors on the instantiated version of the declaration.

The implementation of templates utilizes many of the features described in Section 7 of the OOPLSA paper. A template declaration places a definition including the declaration and template parameters in a new environment namespace. An instantiation expression or type expression looks up the definition, checks that the correct number of argument types are given, then substitutes the arguments for the parameter names in the body of the function, and substitute the name for a mangled one based on the argument types, using the substitution mechanism. The substituted type or function declaration is then lifted to the global level using the lifting mechanism and avoiding duplicates, as described previously. Note that this means that only types declared at the global level can be used as template parameters, but in practice this is not a major restriction as declaring structs and typedefs within a function is not recommended anyway.

There are a number of ways this extension could further be improved. The only parameters that we currently implement are type parameters, but value parameters could be implemented in a similar way. Currently we do not support prototypes for templated structs or functions, this would be required to allow for mutual recursion. We also lack support for C++-style type inference of template parameters for templated functions, which could be implemented by adding a new analysis on TypeExpr to return a list of substitutions for a given type.

```
1.  template<k, v>
2.  struct treemap_s {
3.    k key;
4.    v value;
5.    template treemap_s<k, v> *left;
6.    template treemap_s<k, v> *right;
7.  };
8.
9.  using treemap<k, v> = template treemap_s<k, v>*;
10.
11. template<k, v>
12. template treemap<k, v> put(template treemap<k, v> map, k key, v value) {
13.   if (map == NULL) {
14.     template treemap<k, v> res = malloc(sizeof(template treemap_s<k, v>));
15.     res->key = key;
16.     res->value = value;
17.     res->left = NULL;
18.     res->right = NULL;
19.     return res;
20.   }
21.   else if (key < map->key) {
22.     map->left = inst put<k, v>(map->left, key, value);
23.     return map;
24.   }
25.   else if (key > map->key) {
26.     map->right = inst put<k, v>(map->right, key, value);
27.     return map;
28.   }
29.   else {
30.     map->key = key;
31.     map->value = value;
32.     return map;
33.   }
34. }
35.
36. template<k, v>
37. v get(template treemap<k, v> map, k key) {
38.   if (map == NULL) {
39.     fprintf(stderr, "Key not in map\n"); exit(1);
40.   }
41.   else if (key < map->key)
42.     return inst get<k, v>(map->left, key);
43.   else if (key > map->key)
44.     return inst get<k, v>(map->right, key);
45.   else
46.     return map->value;
47. }
```

Figure 7: A simple implementation of a templated tree map. Note that if `<` or `==` is undefined for type `k`, then an error will be raised in the instantiated code.

5 Vectors

This extension utilizes the templating extension and operator overloading to provide a vector type similar to C++ `std::vector` or lists in Python. This extension provides an alternative to statically sized arrays or explicitly managed allocated pointers.

The only new concrete syntax introduced by this extension is the vector type expression, `vector<type>`, and the vector constructor. This can be given a size, e.g. `vec<int> (4)`, or a literal, e.g. `vec<int> [1, 2, 3]`.

The semantics of operations on a vector are somewhat inspired by Python. The `+`, `+=`, `==`, `[]`, and `[]=` operations are overloaded via the mechanism described in Section 7.1 of the OOPLSA paper, with the expected semantics. ABLEC also supports the overloading of some constructs or combinations of constructs not traditionally part of operator overloading. The field access operator is overloaded to provide the fields `.size`, `.length` (same as `.size`), `.capacity`, and `.elem_size` (`sizeof(a)` where `a` is the vector element type). The member-call operation is overloaded to provide the fields `.append(elem)`, `.insert(index, elem)`, `.extend(other_vector)`, and `.copy()` (shallow copy). Examples of the use of these operators can be seen in Fig. 8.

A vector is implemented as a C `struct`, containing a pointer to information about the vector (size, capacity, etc.), and a double-pointer to the contents of the vector (needed to allow re-allocation). These pointers are garbage-collected using the Boehm garbage collection library [2]. Various vector operations are implemented as functions. Since different struct (and thus function) definitions are needed for every vector parameter type, the template extension, described in Section 4 of this paper, is also used. This allows the ABLEC code implementing vectors to be written in a header file as a list of templated declarations, instead of requiring these ASTs to be written manually as part of the Silver specification for the extension.

Another, similar extension is a string extension, which defines a string type similar to strings in Python. This is implemented by a struct containing a the length and a `const char*`. This extension introduces a new operator, `show()`, which is itself overloadable. We can implement an overload for this operator on vectors in the same way as for a host operator, when the sub-type also has `show()` implemented.

```
1.  vector<int> a = vec<int> [1, 2, 3];
2.  vector<int> b = vec<int> [4, 5, 6];
3.  vector<int> c = a + b;
4.
5.  // a becomes a new vector of a + b
6.  a += b;
7.
8.  // Update a to contain a + b
9.  a.extend(b);
10. b[1] += 7;
11. b.append(6);
12. // Shallow-copy of b
13. vector<int> d = b.copy();
14. b[2] = 7;
15. vector<int> e = vec<int>(3);
16. string res1 = show(c);
17. bool res2 = a == d;
```

(a)

```
1.  vector<int> sieve(int n) {
2.    vector<int> ints = vec<int>(n);
3.    for (int i = 0; i < n; i++)
4.      ints[i] = i;
5.
6.    vector<int> results = vec<int>[];
7.    for (int i = 2; i < n; i++) {
8.      if (ints[i] != -1) {
9.        for (int j = i * 2; j < n; j += i) {
10.         ints[j] = -1;
11.       }
12.       results.append(i);
13.     }
14.   }
15.
16.   return results;
17. }
```

(b)

Figure 8: (a) An example of some of the overloaded operations possible with vectors. (b) A function that generates a vector of all prime numbers less than n.

6 Type-safe enumeration

This extension uses type qualifiers to introduce a type-safe enumeration construct, based on the work of Dietl *et al.* [3]. Enumerated types in C are treated identically to integers. Consider an enumerated type

```
enum Day {
  MONDAY, TUESDAY, ...
};
```

Assigning an arbitrary integer to a variable of this type is not likely to be particularly meaningful, but is allowed. In contrast, C++ treats it as an error when an integer is used where an enumerated type is expected, but continues to allow enumerated types to be used where integers are expected; in other words, C++ treats enumerations as subtypes of integers. Support for type qualifiers in the host language allows us to obtain this type-safe behavior purely as a language extension.

A new marking terminal `enumeration` is introduced that is used in a similar manner as the existing `enum` in C, but forwards to declarations of qualified integers (line 1). For example,

```
enumeration Day {
  MONDAY, TUESDAY, ...
};
```

declares qualified integers such as

```
enum_day int MONDAY  = (enum_day int) 0;
enum_day int TUESDAY = (enum_day int) 1;
```

Arbitrary integers then cannot be assigned to variables of type `enum_day int`, which are declared using another new construct. A second use of the marking terminal `enumeration` specifies the corresponding qualified-integer type (lines 3,11).

```
1.  enumeration Day {
2.    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
3.  };
4.
5.  int foo(enumeration Day day)
6.  {
7.      /* we can implicitly cast enumeration Day to int but not vice versa */
8.      return day;
9.  }
10.
11. int main(int argc, char *argv[])
12. {
13.     enumeration Day day = MONDAY;
14.
15.     foo(day);
16.
17.     /* this would be a type error */
18.     //day = 5;
19.
20.     return 0;
21. }
```

Figure 9: Sample program utilizing the Enumeration extension.

7 Lambda-expressions

This extension introduces lambda closures, anonymous functions that capture the scope in which they were created. New syntax is introduced to create a closure and for a closure type expression:

```
int x = 1, y = 2;
closure<(int) -> int> fn = lambda (int a, int b) -> (x * a + y * b);
```

To apply a closure, the host function call syntax is overloaded:

```
int res = fn(3, 18);
```

As expected, lambda-expressions may capture variables that are currently in scope when they are created. This capture happens by value, so as such captured variables are `const` within the body of the lambda. If mutation of a value in the enclosing scope is required, then a pointer to this variable may be captured and modified instead, as seen in Fig. 10(a).

```
1.  int i = 0;
2.  int *i_ptr = &i;
3.  closure<() -> int> genInt =
4.    lambda () -> (*i_ptr++);
5.  genInt(); // 0
6.  genInt(); // 1
7.  genInt(); // 2
```

(a)

```
1.  template<a, b>
2.  vector<b> map(closure<(a) -> b> fn, vector<a> in) {
3.    vector<b> result = vec<b>(in.size);
4.    for (size_t i = 0; i < in.size; i++)
5.      result[i] = fn(in[i]);
6.    return result;
7.  }
```

(b)

Figure 10: (a) A closure that generates a unique integer each time it is applied. (b) A templated function that maps a closure over a vector. Here the `template` marking terminal corresponds to the template extension, the `vector` and `vec` marking terminals correspond to the vector extension, both of which are assumed to have also been included by the user in a composed translator.

Closures are implemented by a function and an environment that is passed as the first parameter to the function. The function then unpacks the contents of the environment into local variables and evaluates the body of the lambda. The function and struct that implements the environment are specific to each lambda-expression, and is placed at the global scope via the lifting mechanism.

The closure type is implemented by a struct containing a pointer to the environment struct and a function pointer. The struct that implements a closure of a specific type is generated based on the return and parameter types, for example the struct generated corresponding to the type `closure<(int, int) -> int>` is

```
struct _closure_builtin__signed_int__builtin__signed_int__builtin__signed_int__s {
  void *env;
  int (*fn)(void *env, int, int);
};
```

The closure type expression also uses the lifting mechanism to place this struct at the global scope when it has not already been defined.

Since the environment is stored as a pointer that must remain 'alive' after the enclosing function has terminated, we must use some form of dynamic memory allocation. To avoid needing to explicitly 'free' a closure when we are done with it (which may be impossible in some functional programming contexts), we again utilize the Boehm garbage collection library.

This extension is very useful when combined with others, for example as shown in Fig. 10(b). We have used this extension for the basis of other extensions, such as one for term rewriting.

Acknowledgments

This material is partially based upon work supported by the National Science Foundation (NSF) under Grant Nos. 1628929, 1047961, and 0905581. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF.

We also thank August Schwerdfeger for his continued work on Copper and support in the development of ableC.

References

[1] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on Java. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA '07)*, 2007.

[2] Hans-J. Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988.

[3] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 681–690, New York, NY, USA, 2011. ACM.

[4] Clemens Grelck and Sven-Bodo Scholz. SAC: a functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[5] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS)*, 1995.

[6] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. Reliable and automatic composition of language extensions to C. *PACM Progr. Lang.* 1, OOPSLA, Article 98 (October 2017), 29 pages. DOI: 10.1145/3133922

[7] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In *Proceedings of the 12th International Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 61–76. Springer, 2003.

[8] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4):32:1–32:12, July 2012.

[9] Jeffrey M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification based prototyping for embedded systems. In *Proceedings of the 7th ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *LNCS*, September 1999.

[10] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.