Core Expressions: An Intermediate Representation for Expressions in C

Maksim Orlovich and Radu Rugina
Computer Science Department
Cornell University
Ithaca, NY 14853

Abstract. The paper presents a simple representation of C expressions and an algorithm that translates C expressions into this form. The representation can be used as a simple model when designing program analyses. At the same time, it can be used as a compiler intermediate representation to build actual analysis implementations. We describe a formulation of a flow-insensitive, but field-sensitive points-to analysis using this representation. We have implemented this intermediate representation and points-to analysis in CRYSTAL, our program analysis infrastructure. We find core expression very useful in providing a convenient model for program analysis and a simple intermediate compiler representation for C expressions.

1 Introduction

The C programming language offers a rich set of program constructs that give programmers the ability to write complex expressions and perform low-level, potentially unsafe manipulations. However, for a program analysis or compiler system, the rich set of primitives and their potential unsafety becomes an obstacle both in designing and in implementing program analyses.

To simplify the design, program analyses are usually specified for some simplified program model that captures just the key aspects of the computation. This allows a high-level description of the analysis, but leaves parts of the language unspecified uncovered – these parts are treated as “implementation details”. A typical example is pointer analysis, where most formulations assume that programs consist of just four kinds of assignments \( x = \& y \), \( x = y \), \( \* x = y \), and \( x = \* y \).

On the other side, to simplify the implementation, compiler infrastructures perform a certain amount of preprocessing and provide intermediate representations aimed to present a “cleaner” form of the program to the implementor. Typical program simplifications include the elimination of side-effects and implicit control-flow from expressions, and breaking complex expressions down into three-address code. Although they simplify the implementation task, usually there still is a gap between the program representation used in the analysis specification and the one provided by the infrastructure. Furthermore, program representations such as the three-address code introduce a temporary variables, which may affect the performance (possibly even the precision) of an analysis.

\footnote{1}{This work was supported in part by NSF grant CNS-0406345.}
This paper presents a representation for C expressions that is aimed at bridging the gap between analysis specifications and implementations. This representation is intended as a simple program model for analysis designers, and, at the same time, as an intermediate representation for analysis implementors. The proposed representation, which we call core expressions, describes program expressions using a set of six primitives: constants, symbolic addresses, dynamic allocation, binary operations, field offsets, and type-annotated pointer dereferences. Roughly speaking, this is a minimal set of low-level primitives that contain high-level information about field offsets and types. This representation is simple, it automatically provides a canonical representation of C expressions, it makes memory loads and stores explicit (in the form of dereferences), and doesn’t require breaking down large expression trees. We provide a simple algorithm that automatically translates C expressions into core expressions.

Three program representations that are close to ours are those used in SUIF [2], CIL [10], and OpenAnalysis [14]. All of them use expression tree representations and use a fairly small set of primitives. However, SUIF drops the high-level information about field offsets and type sizes (replacing them with machine-dependent constants). CIL and OpenAnalysis have address-of expressions. Finally, SUIF and CIL use cast expressions, but do not have dynamic allocation as a primitive expression. Overall, our set of primitives and the grammar of core expressions is simpler than those of the other representations.

The second contribution of this paper is a formulation of a pointer analysis using core expressions. This is a useful exercise that makes use of our representation. At the same time, it addresses the difficulties that aliasing and memory safety pose to a program analysis system for C. The analysis is field-sensitive – it uses the field and type information in our representation to distinguish between different structure fields. Using the analysis results, the system can: 1) answer pointer aliasing queries; and 2) identify safe structure types for memory blocks that the program accesses in a type-consistent manner. We have implemented this analysis in CRYSTAL, our program analysis infrastructure. We present experimental results collected from this implementation.

To summarize, this paper makes the following contributions:

– It presents a simple, concise representation of program expressions that can be used in analysis specifications and implementations;
– It gives a translation algorithm from C expression to core representation;
– It presents a formulation of a flow-insensitive, field-sensitive pointer analysis algorithm specified and implemented using core expressions; and
– It presents experimental results from an implementation of these techniques in our program analysis infrastructure.

The remainder of the paper is organized as follows. Section 2 presents core expressions and discusses the translation from C expressions. Section 3 presents the pointer analysis algorithm using core expressions. Next, Section 4 gives an overview of our infrastructure. Section 5 presents experimental results. Finally, we discuss related work in Section 6 and conclude in Section 7.
2 Core Expressions

The syntax for core expressions is as follows:

```
Core expressions  e ::= n | s | d | *t e | #f e | e1 ⊕ e2
```

- Numeric constants  n ∈ ℕ
- Symbolic addresses  s ∈ S
- Dynamic allocations  d ∈ D
- Structure fields  f ∈ F
- Types  t ∈ T
- Operators  ⊕

Numeric constants \( n \) include integers and floating-point numbers. It also includes compiler- and machine-dependent constants such as \texttt{sizeof} and \texttt{alignof}. Static addresses \( s \) include addresses of variables (locals and globals, scalars and arrays), and addresses of character strings. These represent symbolic addresses of memory blocks. Static addresses \( s \) also include function (code) addresses. Each dynamic address \( d \) denotes a dynamic memory allocation site. These include both heap allocation (e.g., \texttt{malloc}, \texttt{calloc}, etc) and stack allocation (\texttt{alloca}). During program execution, its value is the address of the newly allocated heap block.

A dereference expression \( *t e \) describes the contents of the memory block that \( e \) points to. The dereference is annotated with the type \( t \) of the target memory block. The size of the memory block is equal to the size of type \( t \). We abbreviate all pointer types as \( \text{ptr} \) since all pointers have the same size (and their target type is not relevant when accessing the memory location that holds the pointer). Field offset expression are of the form \( #f e \), where expression \( e \) must hold the address of the base of a structure that has a field \( f \). Then, \( #f e \) points to the \( f \) field of that structure. Finally, \( e1 ⊕ e2 \) models arbitrary operations, including pointer arithmetic, integer and floating-point arithmetic, bitwise manipulations, and comparisons. For multiplication, we use the symbol \( \times \), to distinguish it from pointer dereferences. We have omitted unary expressions, because their treatment is similar to binary expressions – both in the translation and in the pointer analysis.

Lvalues are expressions that refer to memory blocks [8]. Such expressions are explicit in the core expressions syntax: an expression \( e \) is an lvalue if and only if it is a dereference expression \( e = *t e' \). Therefore, each assignment statement can be represented in the form: \( *t e1 = e2 \), making it explicit that the left-hand side must be an lvalue.

Note that there is no address-of operator or variable symbol in the syntax of core expressions. Essentially, more complex C expressions that use the address-of operator can be reduced to core expressions that build these addresses starting from symbolic addresses and applying dereferences, field offsets and pointer arithmetic. Variables are replaced by dereferences of their symbolic addresses.

For readability purposes, we omit parentheses for expression without binary operations. Such expressions unambiguously associate to the right. For instance, \( *t \#f \#t \#s \) can only be parsed as \( *t (\#f (*t \#s)) \). Type annotations are sometimes dropped to make
expressions more readable. All of the omitted types are either pointers `ptr` or the integer type `int`.

2.1 Translating C Expressions

In this section we show how to translate C expressions into core expressions. We consider C expressions described by the following grammar:

\[
C \text{ expressions} \quad e ::= n \mid \text{str} \mid \text{fun} \mid d \mid x \mid \& e \mid * e \\
\quad \mid e.f \mid e\rightarrow f \mid e_1 \[ e_2 \] \mid e_1 \oplus e_2 \mid (t) e
\]

Variables \( x \in V \)  
String literals \( \text{str} \in \text{Str} \)  
Function symbols \( \text{fun} \in \text{Fun} \)

We do not consider expressions with side-effects (such as auto-increments) or implicit control-flow (such as short-circuit logical operations or conditional expressions). We assume that a preprocessing step transforms such expressions into equivalent sequences of assignments or nested if statements.

Figure 1 shows the algorithm that translates arbitrary C expressions into core expressions. The translation uses two mutually recursive translation procedures, \( \mathcal{R} \) and \( \mathcal{L} \).
\[ \# f e = e, \quad \text{if } f \text{ is the first field} \quad (18) \]
\[ e_1 + (\# f e_2) = (\# f e_1) + e_2 = \# f (e_1 + e_2) \quad (19) \]
\[ e + 0 = 0 + e = e \quad (20) \]
\[ e \times 0 = 0 \times e = 0 \quad (21) \]

Fig. 2. Simplification Rules

R. For a C expression \( e \), \( R[e] \) yields the core expression that represents the value of \( e \). If \( e \) is an lvalue, \( L[e] \) yields a core expression whose value is the address of the memory block that \( e \) represents. The evaluation \( L[e] \) yields an error exactly for those expressions that are not lvalues.

We briefly discuss a few points regarding these rules. The constructs that yield symbolic addresses are string literals, function symbols, and array variables. Their \( R \) evaluation yields their address. Taking the address of these expressions yields the same value, as shown by rules 2 - 4. However, none of these are lvalues and yield an error when evaluated using \( L \). The rules distinguish between array variables (rule 3) and non-array variables (rules 1 and 11) since the former represent symbolic addresses, but the latter represent lvalues. The evaluation of each lvalue expression always yields a dereference expression, as shown by rule 10. The dereference is annotated with the type of expression \( e \), derived from the type declarations. Type casts are discarded by the translation (rules 9 and 16), but they can affect the type annotations in rule 10.

In addition to the translation rules, we use the four simplification rules presented shown in Figure 2. The first rule shows that the offset of the first field is zero. The second rule pulls out field offsets when used from pointer arithmetic operations. The other two are standard algebraic equalities. These rules are applied during the recursive translations \( R \) and \( L \), but we present them separately for clarity.

The table below shows the translation of several standard C expressions into core expressions. For each expression \( e \), it shows the core expression \( R[e] \). The translation eliminates the address-of operator and distinguishes between lvalues (core expressions that start with a dereference) and non-lvalues (expressions that do not start with a dereference):

<table>
<thead>
<tr>
<th>C declaration</th>
<th>C expression e</th>
<th>Core expression ( R[e] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
<td>&amp;x</td>
<td>( s_x )</td>
</tr>
<tr>
<td>int x;</td>
<td>x</td>
<td>( * ) ( _{int} s_x )</td>
</tr>
<tr>
<td>int *x;</td>
<td>*x</td>
<td>( * _{ptr} s_x )</td>
</tr>
<tr>
<td>struct { .. int f; .. } x;</td>
<td>x.f</td>
<td>( * _{int} # _{int} s_x )</td>
</tr>
<tr>
<td>struct { .. int f; .. } x;</td>
<td>&amp; (x.f)</td>
<td>( # _{int} s_x )</td>
</tr>
<tr>
<td>struct { .. int f; .. } *x;</td>
<td>x-&gt;f</td>
<td>( * _{int} # _{ptr} s_x )</td>
</tr>
<tr>
<td>int i, x[10];</td>
<td>x[5]</td>
<td>( # _{ptr} s_x )</td>
</tr>
<tr>
<td>int i, *x;</td>
<td>&amp; (x[5])</td>
<td>( * <em>{int} (s_x + 5 \times size</em>{int}) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( (* <em>{int} s_x) + 5 \times size</em>{int} )</td>
</tr>
</tbody>
</table>
2.2 Canonical Representation

Core expressions provide a canonical representation of program expressions. In C, several expressions can denote the same value. For instance, C expressions \(x, \&(*x)\), and \(*(&x)\) have the same value. Similarly, \((&x)->f\) and \(x.f\) have the same value. In these examples, adjacent \(\&\) and \(*\) operators cancel out.

However, syntactic elimination of adjacent \(\&\) and \(*\) using rewrite rules cannot always be performed. For instance, consider the C expression \(e = \& \& * * x\) for some double pointer \(x\). Eliminating the innermost sequence \(\& *\) yields expression \(\& * x\). Applying the same transformation to the resulting expression yields \(e = x\). However, this is not correct, because \(\& * * x\) doesn’t have an l-value. Hence \(e\) is an invalid expression; it can’t be the same as the valid expression \(x\).

Here is another example. To determine that \((\&x)->f\) and \(x.f\) are equivalent, a rewriting-based transformation needs to rely on the fact that \(e->f\) can be rewritten as \((*e).f\) for any expression \(e\). However, this can lead to certain problems. Consider the following standard way to extract the offset of a field \(f\) in a structure \(S\) in a compiler-independent manner: \(\&((S*)0)->f\). Expanding this expression using the rewrite rule \(e->f \Rightarrow (*e).f\) would result in an expression that has an explicit null pointer dereference as a sub-expression (even though the evaluation of the overall expression does not dereference null).

Furthermore, expression equivalence might not involve adjacent \(\&\) and \(*\) operators. For instance, if \(p\) is a pointer to a structure with a first field int \(f\), then \(*((\text{int} *) p)\) and \(p->f\) are equivalent.

Our rules automatically derive canonical representations in each of the above examples. The translation automatically cancels adjacent \(\&\) and \(*\), but does not suffer from the problems mentioned above. For instance, the translations of C expressions \(x, \&(*x), \text{and } \&(*x)\) yield a unique core expression:

\[
\begin{align*}
R[x] &= *L[x] = *s_x \\
R[(\&(*x))] &= *L[*((\&x))] = *R[\&x] = *L[x] = *s_x \\
R[\&(*x)] &= L[*x] = R[x] = *L[x] = *s_x
\end{align*}
\]

However, the translation \(R[\&(*x)]\) fails because \(L[\&(*x)]\) yields an error. This indicates that \(\&(*x)\) is not an l-value. For the field offset example, the \(R\)-translation of C expression \(\&((\&x)0)->f\) yields core expression \#f 0.

2.3 Translating C Statements

Translating a C statement \(e_1 = e_2\) yields an assignment of core expressions:

\[ *_{\text{type}}(e_1)e'_1 = e'_2 \]

where \(e'_1 = L[e_1]\) is the address of the lvalue in the left-hand side (LHS), and \(e'_2 = R[e_2]\) is the value of the right-hand side (RHS). Similarly, each C function call statement \(e_0 = f_{\text{fun}}(e_1, ..., e_n)\) is translated to:

\[ *_{\text{type}}(e_1)e'_0 = e'_{f_{\text{fun}}(e'_1, ..., e'_n)} \]
where $e_0' = L[e_0]$, $e_{\text{fun}}' = R[e_{\text{fun}}]$, and $e_{i}' = R[e_i]$ for $i = 1..n$. If $e_{\text{fun}}' = s_{\text{fun}}$ is a function symbol, then this is a direct function call.

Consider the C expressions $x$, $*(&x)$, and $s(*x)$ discussed earlier. Of these, only the first two expressions can be used in the LHS of a C assignment. Expression $s(*x)$ cannot because its $L$-evaluation fails. However, all three expressions can be used in the RHS. This matches the desired behavior.

2.4 Identifying Memory Accesses

Identifying all of the memory locations that a C statement or expression reads or writes is straightforward with our representation, because each memory access is an explicit dereference expression. For each translated statement $*_{1} e_{1} = e_{2}$, expression $*_{1} e_{1}$ is the memory location being written; the memory reads are all of the dereference sub-expressions of $e_{1}$ and $e_{2}$.

Example. Consider the C assignment $x[i].a = y->b->c$, where $x$ is an array of structures of type $S$, $y$ is a pointer to a structure, $i$ is an integer, and all of $a$, $b$, and $c$ are pointer fields. Using core expressions, the translated statement is:

$$* #a(s_x + (* s_i) \times size(S)) = * #b * #c * s_y$$

where all of the type annotations are $ptr$, except the one on $s_i$, which is $int$. The leftmost dereference indicates a memory write; the remaining four dereferences are the memory reads.

2.5 Representing Core Expressions

As done in other infrastructures, program expressions can be efficiently represented using a graph structure that contains exactly one node for each distinct core expression and sub-expression. Each expression node keeps pointers to its sub-expression nodes. We refer to this structure as the expression graph. Expression equality is then performed using reference equality on the graph nodes.

3 Pointer Analysis using Core Expressions

We describe a formulation of a flow-insensitive, field-sensitive points-to analysis with unification constraints. The algorithm is based on the techniques originally proposed by Steensgaard [13, 12]. The contribution of this work is the formulation of the analysis using core expressions. The simplicity of core expressions simplifies the algorithm. By the expression translation from the previous section, the analysis automatically handles the complexity of arbitrary C expressions and assignments. However, it doesn’t explicitly deal with C constructs and it doesn’t require normalizing the program to a three-address intermediate form.
3.1 Basic Analysis

A simple points-to analysis algorithm can be developed if we make a few simplification assumptions (to be relaxed in the following subsections). Consider programs with structures and arrays, but assume that integers and pointers have the same size and are the only primitive types. Hence, all type annotations are `ptr` and `int` and we omit them. Assume no reads or writes of entire structures, and consider that each structure is accessed in a type-consistent way (but not necessarily consistent with its declared type). Finally, assume no procedures.

The algorithm builds equivalence classes on top of the nodes in the expression graph. Let $n(e)$ be the equivalence class representative of expression $e$ and denote by $n_1 \equiv n_2$ the fact that classes $n_1$ and $n_2$ have been unified. Initially, each expression is in its own equivalence class. Additionally, there are labeled edges between equivalence classes: for each expression $*e$ there is an edge labeled “$*$” from $n(e)$ to $n(*e)$; and for each expression $#f e$ there is an edge labeled “$f$” from $n(e)$ to $n(#f e)$. Let $n.l$ be the successor equivalence class of $n$ on field labeled $l$. Then, the algorithm is summarized by the following inference rules:

\[
\begin{align*}
  e_1 \oplus e_2 \in P & \quad n(e_1) \equiv n(e_2) \equiv n(e_1 \oplus e_2) \\
  *e_1 = e_2 \in P & \quad n(*e_1) \equiv n(e_2) \\
  n_1 \equiv n_2 & \quad n_1.l \equiv e_2.l
\end{align*}
\]

In words, expression nodes are unified with their children; left-hand and right-hand sides of assignments are unified; and unifications trigger recursive unifications on successor nodes for edges with the same label. To implement these rules, a set of unification constraints is constructed by scanning each statement once. Then, the last rule is enforced using a worklist approach: each time a unification is processed, the recursive unification constraint is placed in the worklist. As usual, unifications can be efficiently implemented using union-find structures ($\equiv$ and $n(\cdot)$ are essentially the union and find primitives).

The meaning of equivalence classes is that two expressions in the same class might have the same value. Recursive unifications show that if two addresses are the same, then field offsets from those addresses for the same field are also identical; and their target memory locations hold the same value because they represent the same memory location. The unifications for binary operations capture the fact that the result of a binary operation might yield the same result as one of the operands (e.g., add 0, multiply by 1, exclusive or with itself, etc).

Hence, after all unifications have been performed, expressions in different equivalence classes definitely have different values. We can use this information to answer aliasing queries: two memory accesses $*e_1$ and $*e_2$ refer to disjoint pieces of memory if their addresses $e_1$ and $e_2$ are in different equivalence classes. Note that the analysis doesn’t compute actual points-to sets, but instead uses the results to answer aliasing queries.

Example. Figure 3(a) shows a simple example that helps understanding how the analysis works. The left part shows a program fragment. The right part shows the corresponding expression graph. Here, $s_a$, $s_b$, $s_c$, and $s_x$ are the symbolic addresses of program variables $a$, $b$, $c$, and $x$. The dashed lines show the unifications that the analysis performs. For instance, the assignment $a=&(b->f)$ causes the unification of $*s_a$
Fig. 3. Example: code fragment and corresponding expression graph. Dashed lines indicate unifications.

with \#f \ast s_b. Two unifications (on field g and the successor * ) are recursively triggered by the assignment b->f=&c. The analysis can determine that \((\ast a) \rightarrow g\) may point to x, because the equivalence class of \(* \#g \ast * s_a\) contains the symbolic address \(s_x\).

3.2 Extensions

We now extend the algorithm to relax the simplifying assumptions about structures, type consistency, and procedure calls. We also show extensions meant to improve the precision of the algorithm.

Structures and Type Consistency. To handle types of different sizes, structure assignments, and type-inconsistent accesses, we extend the analysis as follows:

- First, we place a type tag on each equivalence class. The tag indicates how memory blocks in that class have been accessed. These tags form a flat lattice, consisting of all program types, augmented with a bottom type \(\bot\) (meaning that the memory block has not accessed, or there are no memory blocks in the class), and a top type \(\top\) (meaning type-inconsistent memory accesses). The meet of two different, non-bottom types yields the unknown type. Initially, each class \(n(*_t e)\) is tagged with its access type \(t\). Pointer types are all represented as \(ptr\). All of the non-dereference classes initially have bottom \(\bot\) type tags. When the algorithm unifies two classes with tags \(t_1\) and \(t_2\), it tags the unified class with type \(t_1 \sqcap t_2\).

- Second, we require that, for each class \(n\), all of its outgoing field and dereference edges are consistent. More precisely, the type of \(n\) must be \(ptr\), all of the outgoing fields must belong to the same structure type \(S\), and the class \(n.*\) must be tagged with type of the first field of \(S\). If these conditions are not met, the analysis collapses all of the fields of \(n\) (by unifying it with its each of its field successor classes), and sets the type of \(n\) to \(\top\). Structure consistency is enforced after each unification.

- Finally, the analysis disallows mixing pointer arithmetic (or other binary or unary operations) with field offsets. If a class \(n\) contains the operand or the result of a binary operation, then the analysis collapses all of the fields in \(n\) and sets the type of the resulting equivalence class to \(\top\).
Dereference edges are not type-annotated, so each class has a unique dereference successor. Unions are automatically handled because all of the fields are “first fields”. By Rule 18 union field offsets are eliminated, so the representation keeps just the base address of the union. Accesses to union fields all dereference this address, but with different target types. The analysis automatically merges the target nodes into a single class with an unknown $\top$ type to ensure a unique dereference successor class.

**Direct Procedure Calls.** Consider that, in addition to assignment statements, the program also includes procedure calls of the form $*e_0 = \text{fun}(e_1, \ldots, e_n)$, and return statements $\text{return } e$, where $\text{fun}$ is a function symbol and $e, e_0, \ldots, e_n$ are core expressions. Each call is modeled using assignments of actual arguments to parameters, and of the return value to $*e_0$:

$$*p_1 = e_1; \ldots; *p_n = e_n; \quad *e_0 = *\text{ret}$$

where $p_1, \ldots, p_n$ are the symbolic addresses of the formal parameters of $\text{fun}$, and $\text{ret}$ is a special variable that models the return address of $\text{fun}$. Each return statement $\text{return } e$ is modeled as $*\text{ret} = e$, where $\text{ret}$ is the return address of the enclosing function. We have omitted the arbitrary type annotations on dereferences for readability.

**Function Pointers.** Indirect function calls cannot be handled as above, because the target function is not known, and neither are the formal parameters. Handling function pointers requires augmenting nodes in the expression graph with function signatures. Assume that function calls are now of the form $*e_0 = e_{\text{fun}}(e_1, \ldots, e_n)$, where $e_{\text{fun}}$ is either a symbolic function address $s_{\text{fun}}$ (for direct calls), or an arbitrary expression $*e$ (for indirect function calls). The algorithm is extended as follows:

- each node representing a function address in the expression graph has a vector of core expressions $(p_0, \ldots, p_n, \text{ret})$ that represent the parameters and return values of the function. All of the other nodes have bottom $\bot$ signatures.
- Node unification is extended such that when two nodes are unified, the corresponding parameters and return expressions are also unified. Unifying a signature vector $v$ with a bottom signature yields $v$:
- Finally, for a function call $*e_0 = e_{\text{fun}}(e_1, \ldots, e_n)$, the algorithm looks up the signature vector of $e_{\text{fun}}$. If the vector of $e_{\text{fun}}$ is not bottom, it uses it contents to build actuals-to-formals assignments, as described above. Otherwise, it builds a fresh signature vector (with fresh variables), according to the number of arguments of the call. It then builds the actuals-to-formals assignments.

**Precision Improvements.** As observed in [13], a unification-based approach may create many spurious aliases due to unifications with variables that never hold pointer values. Consider, for instance, a program that uses expressions $p[i]$ and $q[i]$ in unrelated statement to access different arrays pointed to by $p$ and $q$ for the same index $i$. For this program, the algorithm presented so far unifies the two pointers and concludes that the two accesses might be aliased, because they use the same index $i$. Even worse,
all of the pointers that a program nullifies using assignments \( x = 0 \) end up in the same equivalence class, because they get unified with the node of constant 0.

We solve such problems using lazy directional unification constraints. Such constraints are ordered pairs \((e_1 \leftarrow e_2)\) of nodes that get unified lazily, only when the analysis determines that second node \(e_2\) might hold a pointer value. Such pairs are kept around in the unification worklist until there is evidence that \(e_2\) might hold a pointer value. At the end of the algorithm, the worklist might not be empty, but it contains only constraints that were never triggered (i.e., whose second component is never a pointer).

The algorithm is as follows:

– Build lazy constraints for assignments and binary operations. For an assignment \( *e_1 = e_2 \), the algorithm creates the lazy unification pair \(( *e_1 \leftarrow e_2)\) to capture the flow of values in the assignment. For a binary expression \( e = e_1 \oplus e_2 \), it creates the unification pairs \((e \leftarrow e_1)\) and \((e \leftarrow e_2)\).

– Tag each equivalence class with a boolean flag that indicates whether the class might contain pointer values. The flags activate the lazy unifications. Initially, the flags for addresses \(s\) and \(d\) are true, and all of the others are false. Later, when nodes are unified, the resulting class gets the logical or of their flags.

### 4 The Crystal Program Analysis System

We use core expressions as the intermediate representation of expressions in CRYSTAL, a program analysis infrastructure for C developed in our group. The system has been developed over the past year and is entirely written in Java. The goal of the system is to provide a clean and simple representation of the program, and to support standard analyses and analysis frameworks. The structure of the system consists of the following phases:

– **Parsing.** The system parses preprocessed source C files and builds Abstract Syntax Trees and symbol tables. For programs with multiple files, the system parses each of them in turn and loads all of their AST’s in memory to support whole-program analysis. Merging global type declarations is done via structural type equivalence. The subsequent phases proceed only after all files have been parsed and loaded. The system provides the option of serializing AST’s and symbol tables to disk, to avoid parsing in subsequent runs.

– **Simplification and CFG construction.** This phase first flattens inner procedure scopes, and collects all of the local symbols of each procedure in one single symbol table. Expression simplification eliminates side-effects (including assignments and calls) or embedded control-flow from expressions. The resulting expressions are then normalized to the core expression representation proposed in this paper. Then, the system builds one control-flow graph for each procedure. CFG nodes are either switch nodes, or statement nodes. Each statement is an assignment of core expressions, a procedure call, or a return.
– **Pointer analysis.** This phase runs the pointer analysis algorithm presented in Section 3, including the extensions for field-sensitive analysis and function pointers. Finally, the indirect function calls are resolved using the result of the alias analysis and a call graph is constructed.

## 5 Experimental Data

Our experimentation includes the C programs in the Spec CPU2000 benchmark suite [3]. These results were collected on a 3Ghz Pentium 4 machine with 1GB of memory, running Red Hat Enterprise Linux 3.

Table 1 shows several statistics about these benchmarks and our intermediate representation. The first two columns show the sizes of these benchmarks (in KLOC and numbers of functions). The following set of three columns shows the numbers of assignments, call statements, and return statements in the call graph. The rightmost six columns show statistics for core expressions. It breaks down the number of core expressions on their types. Most of the expressions are symbolic addresses (variable addresses and strings), dereferences, and arithmetic operations (unary and binary). Note that there are more dereference expressions than memory blocks, because different portions of the same block are being accessed using different expressions. The number of dynamic allocation expressions is very small, because of two reasons. First, some programs actually use little dynamic data. Second, many programs wrap the standard allocation routines into custom allocation functions that check if the allocation has succeeded.

Table 2 shows statistics collected from from the pointer analysis. The analysis running times are shown in the first column. For most applications, the analysis takes less than one second. A few of the larger applications require a several seconds. For **mesa** and **gcc**, the analysis is more expensive and takes about 2 and 5 minutes, respectively. We believe that these high running times are due to field-sensitivity.

### Table 1. Benchmark statistics

<table>
<thead>
<tr>
<th>Program</th>
<th>KLOC</th>
<th>Funcs</th>
<th>Statements</th>
<th>Core Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Assign</td>
<td>Call</td>
<td>Ret</td>
<td>Const</td>
</tr>
<tr>
<td>164.gzip</td>
<td></td>
<td></td>
<td></td>
<td>1900</td>
</tr>
<tr>
<td>175.vpr</td>
<td>16.9</td>
<td>272</td>
<td>3106</td>
<td>2006</td>
</tr>
<tr>
<td>176.gcc</td>
<td>205.7</td>
<td>2247</td>
<td>58972</td>
<td>22635</td>
</tr>
<tr>
<td>177.mesa</td>
<td>50.2</td>
<td>1106</td>
<td>18892</td>
<td>3589</td>
</tr>
<tr>
<td>179.art</td>
<td>1.2</td>
<td>26</td>
<td>443</td>
<td>154</td>
</tr>
<tr>
<td>181.mcf</td>
<td>1.9</td>
<td>26</td>
<td>523</td>
<td>78</td>
</tr>
<tr>
<td>183.equake</td>
<td>1.5</td>
<td>27</td>
<td>489</td>
<td>206</td>
</tr>
<tr>
<td>186.crafty</td>
<td>19.4</td>
<td>109</td>
<td>5331</td>
<td>2430</td>
</tr>
<tr>
<td>188.ammp</td>
<td>13.2</td>
<td>179</td>
<td>4516</td>
<td>1188</td>
</tr>
<tr>
<td>197.parser</td>
<td>10.9</td>
<td>324</td>
<td>3360</td>
<td>1761</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>61.8</td>
<td>1076</td>
<td>28605</td>
<td>8387</td>
</tr>
<tr>
<td>254.gap</td>
<td>59.4</td>
<td>854</td>
<td>18303</td>
<td>5939</td>
</tr>
<tr>
<td>255.vortex</td>
<td>52.6</td>
<td>923</td>
<td>14147</td>
<td>8525</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>4.6</td>
<td>74</td>
<td>1094</td>
<td>387</td>
</tr>
</tbody>
</table>

---

- **Pointer analysis.** This phase runs the pointer analysis algorithm presented in Section 3, including the extensions for field-sensitive analysis and function pointers. Finally, the indirect function calls are resolved using the result of the alias analysis and a call graph is constructed.

## 5 Experimental Data

Our experimentation includes the C programs in the Spec CPU2000 benchmark suite [3]. These results were collected on a 3Ghz Pentium 4 machine with 1GB of memory, running Red Hat Enterprise Linux 3.

Table 1 shows several statistics about these benchmarks and our intermediate representation. The first two columns show the sizes of these benchmarks (in KLOC and numbers of functions). The following set of three columns shows the numbers of assignments, call statements, and return statements in the call graph. The rightmost six columns show statistics for core expressions. It breaks down the number of core expressions on their types. Most of the expressions are symbolic addresses (variable addresses and strings), dereferences, and arithmetic operations (unary and binary). Note that there are more dereference expressions than memory blocks, because different portions of the same block are being accessed using different expressions. The number of dynamic allocation expressions is very small, because of two reasons. First, some programs actually use little dynamic data. Second, many programs wrap the standard allocation routines into custom allocation functions that check if the allocation has succeeded.

Table 2 shows statistics collected from from the pointer analysis. The analysis running times are shown in the first column. For most applications, the analysis takes less than one second. A few of the larger applications require a several seconds. For **mesa** and **gcc**, the analysis is more expensive and takes about 2 and 5 minutes, respectively. We believe that these high running times are due to field-sensitivity.
### Table 2. Pointer Analysis Statistics

<table>
<thead>
<tr>
<th>Program</th>
<th>Analysis Time (sec)</th>
<th># Equivalence Classes</th>
<th># Classes With Pointers</th>
<th># Classes Type-Consistent</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>0.08</td>
<td>4044</td>
<td>1605</td>
<td>881</td>
</tr>
<tr>
<td>175.vpr</td>
<td>0.17</td>
<td>9828</td>
<td>4782</td>
<td>1928</td>
</tr>
<tr>
<td>176.gcc</td>
<td>9.9</td>
<td>114032</td>
<td>52030</td>
<td>25544</td>
</tr>
<tr>
<td>177.mesa</td>
<td>10.14</td>
<td>34078</td>
<td>18623</td>
<td>4312</td>
</tr>
<tr>
<td>179.art</td>
<td>0.03</td>
<td>941</td>
<td>382</td>
<td>210</td>
</tr>
<tr>
<td>181.mcf</td>
<td>0.09</td>
<td>883</td>
<td>465</td>
<td>165</td>
</tr>
<tr>
<td>183.equate</td>
<td>0.08</td>
<td>1673</td>
<td>509</td>
<td>342</td>
</tr>
<tr>
<td>186.crafty</td>
<td>0.11</td>
<td>13534</td>
<td>4453</td>
<td>2846</td>
</tr>
<tr>
<td>188.ammp</td>
<td>0.16</td>
<td>12341</td>
<td>5029</td>
<td>3092</td>
</tr>
<tr>
<td>197.parser</td>
<td>0.15</td>
<td>8594</td>
<td>3966</td>
<td>1978</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>1.57</td>
<td>42379</td>
<td>22971</td>
<td>9368</td>
</tr>
<tr>
<td>254.gap</td>
<td>1.08</td>
<td>40244</td>
<td>15684</td>
<td>4262</td>
</tr>
<tr>
<td>255.vortex</td>
<td>1.85</td>
<td>39970</td>
<td>22314</td>
<td>11873</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>0.05</td>
<td>2656</td>
<td>862</td>
<td>576</td>
</tr>
</tbody>
</table>

The second column shows the total number of equivalence classes. Comparing this number against the total number of core expressions gives a rough estimate about how much unification has been performed. On average, the number of equivalence classes is 64% of the total number of equivalence classes, with a maximum of 94% for crafty.

The next column shows the number of equivalence classes that might hold pointer values. On average, only 43% of the classes might hold pointer values. The last column shows the number of equivalence classes for which the analysis has determine precise types. On average, the analysis determines that 23% of the equivalence classes are being used in a type-consistent manner. Keep in mind that the analysis does not attempt to compute a precise type for any block that is accessed via arrays or pointer arithmetic.

The large number of non-pointer equivalence classes suggests that the lazy unifications might bring significant improvements in precision. We have experimented with an analysis that performs unifications eagerly and the number of classes with precise types has significantly decreased for all benchmarks, on average by more than an order of magnitude.

### 6 Related Work

Many intermediate representations for the C language have been used in different compiler infrastructures. We discuss those that are closer related to ours. We also discuss related work in pointer analysis.

The SUIF infrastructure [2] is a widely-used research infrastructure for C and C++. Expressions are represented as trees of three-operand RISC-like instructions. This representation uses a larger set of primitives including several kinds of memory access instructions (load, store, copy, and memory copy). High-level information about field accesses is replaced by pointer arithmetic with compiler-specific constants. Although
the field information is available in the form of string annotations, accessing and using it is less convenient and is not type-checked.

CIL is a program analysis and transformation infrastructure [10] for C written in OCaml. It also uses expression trees and removes all side-effects and embedded control-flow from expressions. Expressions use a representation of lvalues as pairs of memory block addresses and offsets within those blocks. The mutually recursive definition of expressions and lvalues makes their expression grammar (and expression data structures) more complex than ours. CIL also provides a flow-insensitive points-to analysis for memory disambiguation. Their points-to analysis handles function pointers, but does not distinguish between structure fields.

OpenAnalysis is an intermediate representation-independent program analysis system developed recently [14]. The system assumes no particular IR. Instead, it uses a generic representation interface so that any compiler infrastructure that implements the interface can also use the OpenAnalysis functionality. The IR interface mainly consists of an abstraction for memory locations, and an abstraction for memory reference expressions. In our framework, the locations correspond to core expressions without dereferences or binary operators; the latter correspond to dereference core expressions.

We consider that our representation of expressions is simpler than all of those in SUIF, CIL, and OpenAnalysis. Unlike CIL and OpenAnalysis, our representation doesn’t have an address-of operator for arbitrary expressions. On the other hand, we use a primitive abstraction for heap allocation, a useful feature found in OpenAnalysis, but not in CIL and SUIF. Finally, we are not aware of published algorithms that show how C expressions are translated into the intermediate forms in CIL, SUIF, or OpenAnalysis.

Many other infrastructures use variants of the three-address code. For instance the McCAT compiler [7] uses an intermediate representation called SIMPLE, where every assignment refers to at most three variables. However, arbitrary numbers of field or array accesses are allowed, and the representation uses many different forms for assignments (depending on whether there are pointer dereferences in the LHS and RHS). Hence, such a representation requires an analysis to deal with a large number of possible cases.

A number of other compiler infrastructures (such as ROSE [1], C-Breeze [9], and the Ckit [5]) mainly provide an AST of the program, but give little or no support for a simpler representation of expressions and assignments in the program.

Finally, the pointer analysis presented in this paper is based on the flow-insensitive pointer analysis with unification constraints proposed by Steensgaard [13], and its subsequent enhancements to support structures and unions [12]. These algorithms are formulated using a program model that includes four kinds of pointer assignments ($x=y$, $x=y$, $\star x=y$, $x=\star y$), as well as field assignments ($x=y->f$) and arbitrary binary operations ($x=y \ op \ z$). The analysis presents specific analysis rules for each of these forms. Most of the other proposed analyses are described just in terms of the four kinds of pointer assignments (e.g., [11, 4, 6]). In contrast, our formulation does not require three-address code normalization, it treats all assignments uniformly, and automatically covers all of the complexity of C expressions.
7 Conclusions

We have presented core expressions, a simple representation for C expressions. The syntax of core expressions is minimal and well-suited for use in the design of program analyses. We have presented an algorithm that translates C expressions into core expressions. Hence, analyses formulated using core expressions automatically apply to the entire set of C language constructs. We have formulated a flow-insensitive points-to analysis using core expressions and implemented it in our program analysis infrastructure. Our experience shows that core expressions are a convenient program representation that bridges the gap between program analysis design and implementation.

References