Bidirectional Translation between a C-like Language and an Imperative λ-calculus

Guillaume Bertholon and Arthur Charguéraud

Inria & Université de Strasbourg, CNRS, ICube, France

Programming languages based on λ-calculus usually have simple semantics. Imperative variants of λ-calculus add mutable variables whose value can be retrieved through explicit get operations. This allows expressing imperative algorithms without complicating too much the semantics. Because it is easy to reason about programs written in imperative λ-calculi, such languages are good candidates to be used for code transformations in internal compilation passes.

However, many programmers are used to more standard imperative programming languages such as C, Java or Python. For them, the explicit handling of mutable variables is unfamiliar and may be hard to read.

In the context of OptiTrust, an interactive source-to-source code transformation framework, we want to manipulate code that is easy to reason about for internal transformations and that, at the same time, reads like C code familiar to programmers from the high performance community.

This paper shows that we can get the benefits of imperative λ-calculus for the transformations, while displaying to the user code in a C-like language. Concretely, we introduce a bidirectional translation between an internal imperative λ-calculus and a language with mutable variables and left-values.

1 Introduction

Many domains of computer science like numerical simulations or machine learning are limited by the available computing power. In those cases, performing the computations faster or with less energy unlocks new usages and can significantly reduce costs. However, modern hardware is complex, and the best performance cannot be reached by using general purpose optimizing compilers (such as GCC, Clang, or ICC) alone [VVAT03]. Indeed, writing a program fine-tuned for the targeted hardware usually creates significant speedups [KK22].

The community of domain specific language (DSL) for array computation, has developed tools for a more interactive compilation such as Halide [RKBA+13] or TVM [CMJ+18]. In such tools, the task of writing the functional behavior of the program is separated from the task of *scheduling* the computation to make it as efficient as possible. Even more interactive workflows such as the one proposed in Roly-poly [IRKF+21] can help visualize and improve Halide schedules. However, all these approaches are intrinsically limited to programs that can be expressed in their DSL.

To go beyond the limits of the DSL approach, we would like to build a fully interactive compiler that allows expressing arbitrary compilation choices on arbitrary programs. The

key ingredient of such a compiler is a good feedback loop for the programmer who may choose a transformation, immediately see the resulting intermediate code (or alternatively see a *diff* summarizing the changes made by the transformation), and repeat the process until the code delivers satisfying performance. In particular, this means that an interactive compiler must display the program after every transformation in a way that is as readable as possible.

Another key ingredient of any compiler (interactive or not) is to perform correct code transformations. The code that implements transformations should be simple and straightforward to avoid implementation bugs that would break the semantics of the transformed program. In summary, an interactive compiler should ideally manipulate a programming language that has a simple semantics for transformations, and that is easily read by programmers.

Yet, there is a tension between the readability of user-facing code and the desired simplicity of compiler intermediate representations. This paper shows that we can alleviate this tension by using two different programming languages and translate code between the two at any point. With this strategy, one of the languages is reserved for user interactions, and the second language is used internally by the code transformations. In practice, we have implemented this strategy in an interactive source-to-source compiler, namely OptiTrust [BCK+24].

From the user point of view, OptiTrust parses and displays code and diffs in a language we call *OptiC*. OptiC is designed to be very close to C, while avoiding most of the complications of the C standard. Our choice is motivated by the fact that most programmers from the high performance community can read C code fluently.

One key feature of the C language, that is shared with OptiC, is the concept of *left-value*, and the mutability of variables. In C, unless specified otherwise, variables have a memory address and their value is mutable. This means that the value of a variable may change between its definition and its invocation. For transformation this means that substituting a variable with its value requires checking whether such value was changed or not. Moreover, expressions that are in *left-value position*, like the subterm of an address-of operator (`&`) or on the left of an assignment (e.g. `=` or `+=`) do not have the same semantics as if they were in *right-value position*. Indeed, in left-value position, the expression evaluates to its address and not to its value. This difference of semantics of expressions in left-value position creates a burden for transformations that need to handle differently expressions depending on whether they appear as left- or right-values. Besides, mutable variables need special treatment in program verification tools that handle C code. For example, in VST [CBG+18], variables that are used as left-value (called addressable variables) cannot be used directly in assertions: they must be described using a dedicated heap predicate.

To avoid the overheads associated with mutability and left-values, OptiTrust works internally on an imperative λ-calculus, which we call Optiλ, better suited for transformations. All code transformations are performed on Optiλ. In Optiλ, like in OCaml, all variables are immutable, and their memory address cannot be retrieved. This makes reasoning about the code easier because all occurrences of a variable yield the same value, no matter their position. To express imperative algorithms, mutations are allowed but only through explicit operations on memory addresses. A memory cell at a given address can be modified with a primitive `set` operation, and its contents can be retrieved with a `get` operation.

Another potential benefit of separating the internal language is that we could more easily add support for other user-facing languages. Indeed, the implementation of the transformations, which operate on Optiλ, would remain the same.

The main contribution of this paper is the presentation of the bidirectional translation that is used inside OptiTrust to convert between the user-facing OptiC and the internal transformation-friendly Optiλ. A specificity of our translation is that it avoids spurious changes between the parsed OptiC code, and the OptiC code that is printed back by keeping some stylistic annotations on Optiλ terms. This means that a round-trip from OptiC back

to OptiC is almost the identity, which can help the user navigate in their own code when using OptiTrust. As far as we know, this is the first time a bidirectional translation between a language with left-values and an imperative λ-calculus is presented with such a form of round-trip property.

This bidirectional translation has been implemented in OCaml, inside the OptiTrust interactive compiler. The open-source implementation can be found at `https://github.com/charguer/optitrust`. The OptiTrust framework has been successfully used in several case studies [BCK+24, BCK+], which reproduce state-of-the-art performance on matrix multiplication, on an OpenCV blur, and on a particle-in-cell plasma simulation. For conciseness, this paper presents only the core of the languages OptiC and Optiλ. We omit loops, arrays, and structures. There features are nevertheless implemented in OptiTrust and exploited in the case studies.

In Section 2, we give an overview of how our bidirectional translation performs on some examples. In Section 3, we present a formal definition of the internal Optiλ language, along with its key design elements, and the underlying call-by-value semantics. Symmetrically, in Section 4, we describe the user-facing OptiC language and its associated semantics. In Section 5, we give the rules for the translation from OptiC to Optiλ and describe the correctness property proving that this translation preserves the semantics of the program. In Section 6, we give the rules for the reverse translation from Optiλ to OptiC, and describe not only the correctness property of the reverse translation, but also the round-trip property.

2 Overview

As said earlier, OptiTrust's internal imperative λ-calculus features let-bindings of variables that are always immutable, and encodes mutability by accesses behind pointers.

Let us present how our bidirectional translation between the internal imperative λ-calculus and the user-facing C-like language works on a few examples.

We start with a simple function `norm2`, that does not include any mutable variable. Immutable variables can be encoded as a simple let-binding when their address is never taken, as shown in the following example. For the purpose of typechecking and of computing reverse translations, the let-bindings introduced by the translation carry the type of the bound variable. Such types appear as subscript in the example Optiλ code below. Note that the translation is bidirectional, so given only the imperative λ-calculus term on the right, our tool is capable of reproducing the exact same C program on the left.

```
int norm2(int x, int y){
  const int xsq = x * x;
  const int ysq = y * y;
  const int res = xsq + ysq;
  return res;
}
```

```
let norm2 = fun(x : int, y : int) ↦ {
  let_int xsq = mul(x, x);
  let_int ysq = mul(y, y);
  let_int res = add(xsq, ysq);
  res
};
```

On the example above, the `return` instruction that appears at the end of the body of the function is translated into a terminal value at the end of a chain of let-bindings. As we explain later, in the syntax of our internal imperative λ-calculus, we exploit n-ary sequences instead of cascades of *let-in* constructs. Doing so makes it easier for programmers to target spans of contiguous instructions, and simplify the implementation of numerous transformations.

We say that a variable is *pure* if its definition is translated into a plain let-binding. Technically, a variable `x` can only be *pure* if there is no assignment operation on `x` and if the address of the variable `x` is never computed via the operator `&x`. Equivalently, a variable `x` can be *pure* if and only if `x` could have been declared with the modifiers `const register`, in the terminology of the C standard.

That said, the programmer may want to translate variables that can be pure into stack-allocated cells, to enable further code transformations. Hence, we need to rely on a keyword (or attribute) to indicate which variables should be translated without stack allocation. We could rely on `const register`, yet for brevity we decided that the keyword `const` alone would indicate the intention of the programmer to introduce a pure variable.

Let us now present an example involving impure variables. The function `norm2Acc`, shown below, computes the same value as `norm2`, yet using a mutable accumulator named `acc`.

```
int norm2Acc(int x, int y){
  int acc;
  acc = x * x;
  acc += y * y;
  return acc;

}
```

```
let norm2Acc = fun(x : int, y : int) ↦ {
  let_ptr(int) acc = stackCell_int();
  set(acc, mul(x, x));
  inplaceAdd(acc, mul(y, y));
  let_int res = get(acc);
  res
};
```

In that case, our translation replaces the mutable variable `acc` with the address to the stack space it occupies. The definition of the variable `acc` is replaced by an explicit allocation of a cell on the stack materialized as a call to stackCell. Then, all functions that modify a mutable variable such as set or inplaceAdd take their address as argument. When the value of a mutable variable is read, such as in the return statement, our translation inserts an explicit get operation. For reasons we explain later, our internal language syntactically only allows single variables as the result of a sequence, and therefore the translation of the return adds a new pure binding on a variable named `res`.

```
const int x = 3;                          ⟷ let_int x = 3;
f(x);                                     ⟷ f(x);

int z;                                    ⟷ let_ptr(int) z = stackCell_int();
z = 6;                                    ⟷ set(z, 6);
const int v = z;                          ⟷ let_int v = get(z);

int* const a = malloc(sizeof(int));       ⟷ let_ptr(int) a = heapCell_int();
*a = *a + 2;                              ⟷ set(a, get(a) + 2);
free(a);                                  ⟷ free(a);

int y = 5;                                ⟷ let_ptr(int) y = ref_int(5);
f(y);                                     ⟷ f(get(y));
y = y + 2;                                ⟷ set(y, get(y) + 2);
y += 4;                                   ⟷ inplaceAdd(y, 4);
y++;                                      ⟷ ignore(getThenIncr(y));

int* const p = &y;                        ⟷ let_ptr(int) p = y;
*p = *p + 2                               ⟷ set(p, get(p) + 2);

int* q = &y;                              ⟷ let_ptr(ptr(int)) q = ref_ptr(int)(y);
q = &z;                                   ⟷ set(q, z);
*q = *q + 2;                              ⟷ set(get(q), get(get(q)) + 2);
```

Figure 1. Example translations from C code into the OptiTrust's internal AST. We suppose that a function `void f(int)` is defined. We also suppose that variables marked as `const` are never modified and that their address is never taken.

Figure 1 presents additional examples illustrating our translations. The lines involving `x`

and `z` summarize the treatment of pure and impure variables. The lines involving `a` illustrate a heap allocated variable. A read operation `*a` is encoded as the function call `get(a)`, and an assignment `*a = v` is encoded as `set(a, v)`. Thus, heap-allocated variables and impure stack-variables are treated essentially the same way in our internal λ-calculus—with the main difference that stack-allocated variables are implicitly deallocated. The name of the primitive operation, whether stackCell or heapCell, is used to guide the reverse translation.

The lines of Figure 1 involving `y` illustrate the encoding of operators. The lines involving `p` show how we handle the address-of operator. Finally, the lines involving `q` show how our translation handles a mutable variable that stores pointers.

We next explain the last key ingredient of our bidirectional translation: the use of *annotations*. The issue stems from the fact that Optiλ features fewer language constructions than OptiC. For example, OptiC features the construct *if-then* without `else`, as well as C's ternary operator `b ? x : y`, whereas Optiλ only features a plain *if-then-else* construct. To enable going back from Optiλ to OptiC, we allow certain Optiλ terms to carry a *style* annotation. For example, we can annotate an *if-then-else* construct with the annotation ∅ to indicate that the *else* branch was absent in the input OptiC code. The example below shows additional examples of style annotations, with if-statements annotated using `&&` or `?:`, for keeping track of specific OptiC constructions.

```
void f(int* p){
  if (p && *p == 0){
    *p = 1;
  }
  if (p ? (*p == 2): false){
    *p = 3;
  } else {}
}
```

```
let f = fun(p : ptr(int)) ↦ {
  if (if^&& p then eq(get(p), 0) else false) then {
    set(p, 1)
  } else {}^∅
  if (if^?: p then eq(get(p), 2) else false) then {
    set(p, 3)
  } else {}
};
```

The key point is that a style annotation never alter the semantics of a construct. OptiTrust transformations do a best effort at preserving annotations. Yet, semantics preservation remains guaranteed even if any of the style annotation is dropped during a transformation.

3 Optiλ: an Internal, Imperative λ-calculus

As said in the introduction, the key feature of the Optiλ language is to have few constructions, with simple semantics. The aim is to simplify as much as possible the implementation of code transformations. At a high-level, Optiλ resembles the core of the OCaml language. In particular, variables are always immutable, and mutation involves explicit calls to heap-manipulating functions. We start by describing the specificities of Optiλ. We then present the formal syntax of the language and its big-step semantics.

Recall that this paper presents a subset of Optiλ and OptiC. We omit arrays, loops, and structures from the presentation. These features are nevertheless implemented in OptiTrust, and exploited in several case studies.

Sequences A sequence is a term that consists of a list of subterms with side effects or let-bindings, to be executed in order, and of a return value. A sequence is written $\{t_1; ...; t_n; r\}$, where each t_i could be of the form **let** $x = t$, and where r denotes a return value for the sequence—possibly the unit value. This presentation of sequences is similar to that found in, e.g., the Rust language. The expression r cannot perform side-effects; in our current implementation, the result value r is syntactically restricted to be either unit or a variable. We translate a statement of the form **return** t that appears in terminal position of a C function into "**let** $x = t$; x" where x is a fresh variable name.

Each sequence introduces a lexical scope, therefore when t_i is of the form **let** $x = t$, the variable x can be used in any t_j for $j > i$ but not after the closing brace. We impose in ASTs

of Optiλ the invariant that every function body consists of a sequence block, even if the sequence contains a single instruction.

In Optiλ, we enforce that all the instructions in a sequence have type unit (equivalent of `void` in C). To do so, we insert calls to the built-in function ignore around instruction that are not of type `void` in the C code.

Sequences in Optiλ may also include *ghost instructions*. A ghost instruction behaves, semantically, as a no-op. It guides, however, the typechecker of OptiTrust, typically by altering the way the memory state is described in the Separation Logic invariants. These invariants may be exploited for guiding code transformations, and for checking their correctness. A key interest of our design is that it allows placing instructions *after* the point at which the return value is computed. Doing so is specifically useful for ghost instructions that depend on the result value. From the perspective of our bidirectional translation, ghost instructions are treated exactly like regular function calls.

Manipulation of Heap and Stack Cells To account for heap-allocated data, OptiTrust provides the following standard primitive functions: heapCell for allocating an uninitialized cell on the heap, get for reading a cell, set for writing a cell, and free for freeing allocated cells. As usual, a read in an uninitialized memory cell is undefined behavior. Additionally, to account for stack-allocated variables, OptiTrust includes special functions. The operation stackCell() allocates a memory cell on the stack without initializing its contents. Its space is automatically reclaimed at the end of the surrounding sequence. The operation ref(t) also allocates a memory cell on the stack but initializes it with t. These two special operations are meant to occur as part of a let-binding, for example **let** x = ref(3), occurring directly within a sequence. A binding **let** x = ref(t) is strictly equivalent to **let** x = stackCell(); set(x, t). The two stack-allocation operators, apart from their implicit-free behavior, are treated like other primitive functions.

Unbounded integers In OptiTrust, the semantics is based on unbounded integers. The type int can accept infinitely large integers, like in Python. This greatly helps when proving properties about the code, and removes corner cases for arithmetic optimizations that should normally deal with possible overflows. In practice, such unbounded integers can have a significant performance cost, and therefore they should be eliminated at some point during the interactive compilation process. In future work, we plan to leverage function specifications and a value analysis to choose an actual bounded size for representing integers. To keep things simple, this paper does not include fixed size integer types in the grammar.

Other Language Constructs The other language constructs of Optiλ are standard. They include function abstraction, function calls, and conditionals. Our implementation accounts for a diversity of literal types. For simplicity, we consider in this paper only two kinds of literals: the metavariable b denotes a boolean literal (either true or false), and the metavariable n denotes an integer literal.

Other Primitive Operations Besides the aforementioned primitive operations for manipulating heap and stack cells, OptiTrust's internal language provides primitive functions that correspond to the arithmetic and boolean operators of the C language.

However, certain C operators do not behave like function calls and therefore must be encoded differently. Two such operators are the short-circuiting boolean operators `&&` and `||`. In our imperative λ-calculus, in order to limit the number of language constructions and keep simple semantics, we chose to disallow any operator that is not expressible as a call-by-value function call. Therefore, the C short-circuit operators cannot exist. However, this is not a real limitation since we can represent those short-circuiting operations with conditionals whenever the second argument is not a simple expression (in particular, if it might fail or perform side effects).

r :=	$\varnothing$ | x		result of a sequence
t :=	x		variables
	b | n		boolean values, and number values
	$\{t_1; ...; t_n; r\}$		sequence
	let $x = t$		variable definition
	fun$(a_1, ..., a_n) \mapsto t$		function definition
	$t_0(t_1, ..., t_n)$		function call
	if t_0 **then** t_1 **else** t_2		conditional
	add(t_1, t_2) | inplaceAdd(t_1, t_2)	...	primitive arithmetic operations
	ignore(t)		primitive to ignore a return value
	get(t) | set(t_1, t_2) | free(t)		primitive operations on memory cells
	stackCell() | ref(t) | heapCell()		allocations of memory cells

Figure 2. Grammar of Optiλ, the internal λ-calculus in OptiTrust. The grammar reserves space for annotations that are not shown.

Annotations In addition to the ghost instructions presented earlier, each subterm of an Optiλ program can carry a number of extra information that do not affect the semantics in the form of annotations. Currently, our internal AST carries the following information:

- The location of the subterm in the initial source code,
- User-placed marks allow referring to subterms by name during transformations,
- Separation logic contracts for functions and loops,
- Type information for all bindings, and for every subterm and operator,
- Style annotations to guide the reverse translation from Optiλ to OptiC.

As far as this paper is concerned, only the types on let-bindings and the style annotations are of interest.

Formal Syntax and Semantics Formally, the syntax of the Optiλ language is given by Figure 2.

A value can be either unit (written ()), a boolean, a number, a location in memory or a *function closure*. A function closure is written $\textbf{fun}^{\sigma}(a_1, ..., a_n) \mapsto t_f$, and captures the environment σ around the function definition. This capture allows function bodies to refer to variables defined in the scope surrounding the function definition. Note, however, that invoking a closure that performs a read at a location that had been captured by the closure may result in the program being stuck if this location has been freed (either explicitly or implicitly), or if it has never been initialized.

The semantics of the language is described using an environment, written σ, and a memory state, written μ. The environment maps the local variable names to their corresponding values. The memory state is a map from locations to either a value, written v, or the *uninitialized* token, written $\bot$. Thus, if a rule features a premise of the form $\mu(l) = v$, this premise captures the fact that the cell at address l has been initialized, and that is currently contains the value v. Our semantics consider that all datatypes occupy one cell slot in memory and that each address uniquely correspond to one of such cell slot. This restriction might be lifted later when introducing structures in the language.

The semantic judgement $t/^{\sigma}_{\mu} \Downarrow v/^{\sigma'}_{\mu'}$ signifies that term t in an environment σ and a memory state μ reduces to a value v and update the environment to σ' and the memory state to μ'.

$$\frac{}{x/^{\sigma}_{\mu} \Downarrow \sigma(x)/^{\sigma}_{\mu}}\ \textsc{Var} \qquad \frac{t/^{\sigma}_{\mu} \Downarrow v/^{\sigma}_{\mu'}}{(\textbf{let}\ x = t)/^{\sigma}_{\mu} \Downarrow ()/^{\sigma[x \mapsto v]}_{\mu'}}\ \textsc{Let}$$

$$\frac{t_0/^{\sigma_0}_{\mu_0} \Downarrow ()/^{\sigma_1}_{\mu_1} \qquad \{t_1; ...; t_n; r\}/^{\sigma_1}_{\mu_1} \Downarrow v/^{\sigma_1}_{\mu'}}{\{t_0; t_1; ...; t_n; r\}/^{\sigma_0}_{\mu_0} \Downarrow v/^{\sigma_0}_{\mu'}}\ \textsc{SeqInstr} \qquad \frac{v = \begin{cases} () & \text{if } r = \varnothing \\ \sigma(r) & \text{otherwise} \end{cases}}{\{r\}/^{\sigma}_{\mu} \Downarrow v/^{\sigma}_{\mu}}\ \textsc{SeqRes}$$

$$\frac{}{(\textbf{fun}(a_1, ..., a_n) \mapsto t_f)/^{\sigma}_{\mu} \Downarrow (\textbf{fun}^{\sigma}(a_1, ..., a_n) \mapsto t_f)/^{\sigma}_{\mu}}\ \textsc{Fun}$$

$$\frac{\begin{array}{c} t_0/^{\sigma}_{\mu_0} \Downarrow (\textbf{fun}^{\sigma_f}(a_1, ..., a_n) \mapsto t_f)/^{\sigma}_{\mu_1} \\ \forall i \in [1; n], \quad t_i/^{\sigma}_{\mu_i} \Downarrow v_i/^{\sigma}_{\mu_{i+1}} \qquad \sigma_c = \sigma_f[\overline{a_i \mapsto v_i}] \qquad t_f/^{\sigma_c}_{\mu_{n+1}} \Downarrow v/^{\sigma_c}_{\mu'} \end{array}}{t_0(t_1, ..., t_n)/^{\sigma}_{\mu_0} \Downarrow v/^{\sigma}_{\mu'}}\ \textsc{Call}$$

$$\frac{t_c/^{\sigma}_{\mu} \Downarrow \mathsf{true}/^{\sigma}_{\mu_c} \qquad t_t/^{\sigma}_{\mu_c} \Downarrow v/^{\sigma}_{\mu'}}{(\textbf{if}\ t_c\ \textbf{then}\ t_t\ \textbf{else}\ t_f)/^{\sigma}_{\mu} \Downarrow v/^{\sigma}_{\mu'}}\ \textsc{IfTrue} \qquad \frac{t_c/^{\sigma}_{\mu} \Downarrow \mathsf{false}/^{\sigma}_{\mu_c} \qquad t_f/^{\sigma}_{\mu_c} \Downarrow v/^{\sigma}_{\mu'}}{(\textbf{if}\ t_c\ \textbf{then}\ t_t\ \textbf{else}\ t_f)/^{\sigma}_{\mu} \Downarrow v/^{\sigma}_{\mu'}}\ \textsc{IfFalse}$$

$$\frac{t/^{\sigma}_{\mu} \Downarrow l/^{\sigma}_{\mu'} \qquad \mu'(l) = v}{\mathsf{get}(t)/^{\sigma}_{\mu} \Downarrow v/^{\sigma}_{\mu'}}\ \textsc{Get} \qquad \frac{t_1/^{\sigma}_{\mu_0} \Downarrow l_1/^{\sigma}_{\mu_1} \qquad t_2/^{\sigma}_{\mu_1} \Downarrow v_2/^{\sigma}_{\mu_2} \qquad l_1 \in \mathrm{dom}(\mu_2)}{\mathsf{set}(t_1, t_2)/^{\sigma}_{\mu_0} \Downarrow ()/^{\sigma}_{\mu_2[l_1 \mapsto v_2]}}\ \textsc{Set}$$

$$\frac{t/^{\sigma}_{\mu} \Downarrow v/^{\sigma}_{\mu'}}{\mathsf{ignore}(t)/^{\sigma}_{\mu} \Downarrow ()/^{\sigma}_{\mu'}}\ \textsc{Ignore} \qquad \frac{t_1/^{\sigma}_{\mu_0} \Downarrow n_1/^{\sigma}_{\mu_1} \qquad t_2/^{\sigma}_{\mu_1} \Downarrow n_2/^{\sigma}_{\mu_2} \qquad n' = n_1 + n_2}{\mathsf{add}(t_1, t_2)/^{\sigma}_{\mu_0} \Downarrow n'/^{\sigma}_{\mu_2}}\ \textsc{Add}$$

$$\frac{t_1/^{\sigma}_{\mu_0} \Downarrow l_1/^{\sigma}_{\mu_1} \qquad t_2/^{\sigma}_{\mu_1} \Downarrow n_2/^{\sigma}_{\mu_2} \qquad \mu_2(l_1) = n_1 \qquad n' = n_1 + n_2}{\mathsf{inplaceAdd}(t_1, t_2)/^{\sigma}_{\mu_0} \Downarrow n'/^{\sigma}_{\mu_2[l_1 \mapsto n']}}\ \textsc{InplaceAdd}$$

$$\frac{l \notin \mathrm{dom}(\mu)}{\mathsf{heapCell}()/^{\sigma}_{\mu} \Downarrow l/^{\sigma}_{\mu[l \mapsto \bot]}}\ \textsc{HeapCell} \qquad \frac{t/^{\sigma}_{\mu} \Downarrow l/^{\sigma}_{\mu'} \qquad l \in \mathrm{dom}(\mu')}{\mathsf{free}(t)/^{\sigma}_{\mu} \Downarrow ()/^{\sigma}_{\mu' \setminus l}}\ \textsc{Free}$$

$$\frac{l \notin \mathrm{dom}(\mu_0) \qquad \{t_1; ...; t_n; r\}/^{\sigma_0[x_0 \mapsto l]}_{\mu_0[l \mapsto \bot]} \Downarrow v/^{\sigma_0[x_0 \mapsto l]}_{\mu'} \qquad l \in \mathrm{dom}(\mu')}{\{\textbf{let}\ x_0 = \mathsf{stackCell}(); t_1; ...; t_n; r\}/^{\sigma_0}_{\mu_0} \Downarrow v/^{\sigma_0}_{\mu' \setminus l}}\ \textsc{StackCell}$$

$$\frac{\{\textbf{let}\ x_0 = \mathsf{stackCell}(); \mathsf{set}(x_0, t_0); t_1; ...; t_n; r\}/^{\sigma}_{\mu} \Downarrow v/^{\sigma}_{\mu'}}{\{\textbf{let}\ x_0 = \mathsf{ref}(t_0); t_1; ...; t_n; r\}/^{\sigma}_{\mu} \Downarrow v/^{\sigma}_{\mu'}}\ \textsc{Ref}$$

Figure 3. Semantics of the Optiλ internal language. Other arithmetic built-in functions follow the pattern of Add or InplaceAdd.

Note that only the let-binding instructions that add a new variable name to the context may alter the environment. For simplicity, we here consider a deterministic semantics. To smoothly handle non-determinismn we could use an omni-big-step semantics [CCEG23], or fall back to a standard small-step judgement.

The call-by-value, big-step semantics appears in Figure 3. When $\sigma(x)$ appears in a rule, there is an implicit assumption that $x \in \mathrm{dom}(\sigma)$. We write $\sigma[x \mapsto v]$ to create or update a binding from x to v. The operations $\mu(l)$ and $\mu[l \mapsto v]$ operate similarly on stores. We write $\mu \setminus l$ the store obtained by removing the binding on l from μ.

The attentive reader might notice that our semantics fixes the order of evaluation of the arguments of a function call from left to right. This choice is arbitrary. In practice, OptiTrust relies on a Separation Logic type-system (not shown in this paper), which ensures that the argument evaluation order cannot influence the results of computations.

4 OptiC: a C-Like, User-Facing Language

We strive to make the user-facing language of OptiTrust as close to C as possible, in order to make the tool accessible to most programmers.

Comparison with C Syntactically, OptiC is a subset of C, with a few extensions borrowed from C++. Our current implementation of OptiTrust parses OptiC code using Clang. Moreover, OptiTrust users can benefit from the C or C++ support of their IDEs to edit OptiC code.

Semantically, OptiC admits a simpler semantics than C. Supporting all the features of the C language would be extremely challenging. To see why, it suffices to contemplate the size of the Coq formalization of a significant subset of C [Kre15]. OptiC features fewer undefined behaviors than C, in particular with respect to evaluation order. Hence, it is incorrect to compile OptiC code using an arbitrary C-compliant compiler. Instead, either a prior translation to C is required, e.g. to bind intermediate expressions; or one should translate OptiC code directly into a lower-level language, such as CompCert's C-light or LLVM IR.

Strict order of evaluation for all operators In standard C, operators do not necessarily behave like calls to primitive functions. Indeed, the standard allows for a more liberal argument evaluation order. This is visible for pre/post-increment/decrement operators such as `i++`. For example, it is undefined whether the instruction `u = u++;` increments `u` or not. However, when written as call-by-value function calls, `set(&u, getAndIncr(&u))`, it is obvious that it cannot increment `u`. Since code transformations might accidentally produce code such as `u = u++`, and we do not want to treat operators in a special way, the OptiC language exposes fewer undefined behaviors than standard C. In that case, we impose that operators behave like function calls. This means that we do not guarantee any evaluation order of the arguments, but we ensure that all arguments performed all their side effects before the execution of an operator syntactically higher in the AST. This is not a problem when importing C code because this only restricts the number of possible behaviors of any given piece of code.

No integer overflows Like for the internal imperative λ-calculus, in our C-like language, we consider that type `int` is unbounded. This specifically alleviates the burden of undefined behaviors when signed overflow occurs. In OptiTrust, a transformation that chooses the width of an integer variable is responsible to insert assertions that prevent overflow or prove the absence of such overflow. We leave the support of bounded integers to future work.

Variables marked `const` and function arguments are pure As we saw in Section 2, it is important for our translation to understand whether a variable is pure or impure. Since transformations may react differently in presence of pure or impure variables, we want to syntactically distinguish variables we treat as pure. To reduce the amount of syntactic noise, our internal C-like language treats the keyword `const` as a request to handle the variable as a pure variable. Moreover, we made the design decision to always treat function arguments as pure variables. The mutation of function arguments is allowed in C, yet it is a rarely used feature, which can easily be avoided. We might, in future work, extend our translation to handle mutated arguments.

Function types and function variables In C, a programmer can write pointers to functions such as `int (*fptr)(int);`. This type can only accept functions that do not capture their surrounding environment. The semantics of the operators `&` and `*` when applied to C functions is not as simple as one may hope. In OptiC, we chose instead to use the C++ function types, such as `std::function<int(int)>`, which in this paper we write `fun<int(int)>`, assuming `fun` to be defined as an alias. A local variable with a function type may be either pure or impure. However, all functions declared using the syntax of C function definitions (e.g., `int f(){ return 42; }`) are represented as pure variables.

Syntax extensions for translating back from Optiλ OptitTrust transformations may generate ASTs in Optiλ with arbitrary shapes. In particular, the grammar of Optiλ allows function abstractions and sequences with a return value to appear anywhere as subterms in the AST. This flexibility does not exist in the standard C. Yet, we wish to be able to display to the OptiTrust user the corresponding AST in OptiC syntax. To that end, we consider standard extensions of the C language; such extensions would probably be already familiar to the OptiTrust user. For sequences, we consider the GNU C extensions[1], which supports the syntax `({` $u_1; ...; u_n; u_r;$ `})`, where u_r is an expression that corresponds to the return value. For functions, we borrow the C++ syntax for closures, written `[&](T_1 a_1, ..., T_n a_n){...}`. Moreover, we allow the nested functions from the GNU C extensions[2].

Unsupported C Features OptiTrust does not aim at covering all the features of the C language in the short term. Our current implementation does not handle `switch`, although we could presumably encode it using a cascade of if-statements. There is no support yet for `break`, `continue`, or non-terminal `return`. Besides, we do not treat `goto`s—and we only plan to consider a restricted form via a well-scoped block-exit contruct.

Semantics We equip the OptiC language with the big-step semantics. Similarly to the semantics described for Optiλ, the semantics of OptiC is also described using an environment and a memory state. The memory state, written m, maps locations to values. Such values also include unit, booleans, integers, locations and function closures this time written with a C++-like syntax $[s](T_1\, a_1, ..., T_n\, a_n)\, u_f$ when capturing an environment s. The environment, written s, maps variables to locations. This contrast with Optiλ, where environments map variables to values.

The semantics OptiC is described using two reduction judgements. $u/^{s}_{m} \Downarrow v/^{s'}_{m'}$ asserts that the term u, appearing as an instruction or as a right-value, reduces, in the environment s and the memory state m, to the value v, updating the environment to s' and the memory state to m'. The second judgement $u/^{s}_{m} \Downarrow^{\&} v/^{s'}_{m'}$ is similar but for evaluating expressions in left-value position; its output value v is always a location. The evaluation rules appear in Figure 4.

[1] `https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc/Statement-Exprs.html`
[2] `https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc/Nested-Functions.html`

$$\frac{}{x/^s_m \Downarrow m(s(x))/^s_m}\ \textsc{Var} \qquad \frac{}{x/^s_m \Downarrow^{\&} s(x)/^s_m}\ \textsc{LVar}$$

$$\frac{u_0/^{s_0}_{m_0} \Downarrow v_0/^{s_1}_{m_1} \qquad \{u_1;\ldots;u_n;q\}/^{s_1}_{m_1} \Downarrow v/^{s_1}_{m'}}{\{u_0;u_1;\ldots;u_n;q\}/^{s_0}_{m_0} \Downarrow v/^{s_0}_{m'}}\ \textsc{SeqCons}$$

$$\frac{q = (t;) \vee q = (\textbf{return}\, t;) \qquad t/^s_m \Downarrow v/^s_{m'}}{\{q\}/^s_m \Downarrow v/^s_{m'}}\ \textsc{SeqRes} \qquad \frac{q = \varnothing \vee q = (\textbf{return};)}{\{q\}/^s_m \Downarrow ()/^s_m}\ \textsc{SeqVoid}$$

$$\frac{}{([\&](T_1\, a_1, \ldots, T_n\, a_n)\, u_f)/^s_m \Downarrow ([s](T_1\, a_1, \ldots; T_n\, a_n)\, u_f)/^s_m}\ \textsc{FunClosure}$$

$$\frac{\begin{array}{c} u_0/^s_{m_0} \Downarrow ([s_f](T_1\, a_1, \ldots, T_n\, a_n)\, u_f)/^s_{m_1} \\ \forall i \in [1;n],\quad u_i/^s_{m_i} \Downarrow v_i/_{m'_{i+1}} \quad\wedge\quad l_i \notin \mathrm{dom}(m'_{i+1}) \quad\wedge\quad m_{i+1} = m'_{i+1}[l_i \mapsto v_i] \\ s_c = s_f[\overline{a_i \mapsto l_i}] \qquad u_f/^{s_c}_{m_{n+1}} \Downarrow v/^{s_c}_{m_q} \qquad m' = m_q \setminus \{\, l_i \,|\, i \in [1;n]\,\} \end{array}}{u_0(u_1, \ldots, u_n)/^s_{m_0} \Downarrow v/^s_{m'}}\ \textsc{Call}$$

$$\frac{u_c/^s_m \Downarrow \mathsf{true}/^s_{m_c} \qquad u_t/^s_{m_c} \Downarrow v/^s_{m'}}{(\mathsf{if}(u_c)\ u_t\ \mathsf{else}\ u_f)/^s_m \Downarrow ()/^s_{m'}}\ \textsc{IfTrue} \qquad \frac{u_c/^s_m \Downarrow \mathsf{false}/^s_{m_c} \qquad u_f/^s_{m_c} \Downarrow v/^s_{m'}}{(\mathsf{if}(u_c)\ u_t\ \mathsf{else}\ u_f)/^s_m \Downarrow ()/^s_{m'}}\ \textsc{IfFalse}$$

$$\frac{u/^s_m \Downarrow l/^s_{m'} \qquad m'(l) = v}{(*u)/^s_m \Downarrow v/^s_{m'}}\ \textsc{Get} \qquad \frac{u/^s_m \Downarrow v/^s_{m'}}{(*u)/^s_m \Downarrow^{\&} v/^s_{m'}}\ \textsc{LGet} \qquad \frac{u/^s_m \Downarrow^{\&} v/^s_{m'}}{(\&u)/^s_m \Downarrow v/^s_{m'}}\ \textsc{Address}$$

$$\frac{u_1/^s_{m_0} \Downarrow^{\&} l_1/^s_{m_1} \qquad u_2/^s_{m_1} \Downarrow v_2/^s_{m_2} \qquad l_1 \in \mathrm{dom}(m_2)}{(u_1 = u_2)/^s_{m_0} \Downarrow ()/^s_{m_2[l_1 \mapsto v_2]}}\ \textsc{Set}$$

$$\frac{t_1/^s_{m_0} \Downarrow n_1/^s_{m_1} \qquad t_2/^s_{m_1} \Downarrow n_2/^s_{m_2}}{(t_1 + t_2)/^s_{m_0} \Downarrow (n_1 + n_2)/^s_{m_2}}\ \textsc{Add}$$

$$\frac{t_1/^s_{m_0} \Downarrow^{\&} l_1/^s_{m_1} \qquad t_2/^s_{m_1} \Downarrow n_2/^s_{m_2} \qquad m_2(l_1) = n_1 \qquad n' = n_1 + n_2}{(t_1 \mathrel{+}= t_2)/^s_{m_0} \Downarrow n'/^s_{m_2[l_1 \mapsto n']}}\ \textsc{InplaceAdd}$$

$$\frac{l \notin \mathrm{dom}(m)}{\mathsf{malloc}(\mathsf{sizeof}(T))/^s_m \Downarrow l/^s_{m[l \mapsto \bot]}}\ \textsc{HeapCell} \qquad \frac{u/^s_m \Downarrow l/^s_{m'} \qquad l \in \mathrm{dom}(m')}{\mathsf{free}(u)/^s_m \Downarrow ()/^s_{m' \setminus l}}\ \textsc{Free}$$

$$\frac{l \notin \mathrm{dom}(m_0) \qquad \{u_1;\ldots;u_n;q\}/^{s_0[x_0 \mapsto l]}_{m_0[l \mapsto \bot]} \Downarrow v/^{s_0[x_0 \mapsto l]}_{m'} \qquad l \in \mathrm{dom}(m')}{\{T\, x_0; u_1;\ldots;u_n;q\}/^{s_0}_{m_0} \Downarrow v/^s_{m' \setminus l}}\ \textsc{StackCell}$$

$$\frac{\{T\, x_0; x_0 = u_0; u_1;\ldots;u_n;q\}/^s_m \Downarrow v/^s_{m'}}{\{T\, x_0 = u_0; u_1;\ldots;u_n;q\}/^s_m \Downarrow v/^s_{m'}}\ \textsc{InitStackCell}$$

$$\frac{\{\mathsf{fun}{<}T_0(T_1, \ldots, T_n){>}\, x = [\&](T_1\, a_1, \ldots, T_n\, a_n)\, u_f; u_1;\ldots;u_n;q\}/^s_m \Downarrow v/^s_{m'}}{\{T_0\, x(T_1\, a_1, \ldots, T_n\, a_n)\, u_f; u_1;\ldots;u_n;q\}/^s_m \Downarrow v/^s_{m'}}\ \textsc{FunDef}$$

Figure 4. Semantics of the OptiC user-facing language. The ternary conditional operator and the short-circuiting operators (not shown) follow a pattern similar to IfTrue and IfFalse. Other arithmetic built-in functions follow the pattern of Add or InplaceAdd. OptiC impose that the **return** keyword is only used in terminal position of a function body, and therefore acts like the result value of a sequence.

5 Translation from OptiC to Optiλ

This section describes the translation from the user-facing C-like language into OptiTrust's internal λ-calculus. We call this operation the *encoding*.

As exposed in the overview, the translation from the user-facing language to the internal λ-calculus crucially depends on the notion of pure variables. We can assume at this step that all pure variables are marked as `const` in the C-like language (recall that it corresponds to `const register` in C). This means that pure variables can be considered immutable and without an address.

The essence of the encoding process is to eliminate the notion of left-value by replacing impure variables with their stack-allocated address. Then, the encoding wraps the accesses to values of impure variables with a `get` operation. Such a process of elimination of the left-values is commonly found in the implementation of compilers. However, compilers in general are not concerned with supporting a reverse translation.

Figure 5 defines our translation from C to OptiTrust's internal language. We write $\lfloor u \rfloor$ the encoding of an OptiC term u, which could be a statement or an expression in right-value position. We write $\lfloor u \rfloor^{\&}$ the encoding of an OptiC term u appearing in left-value position.

During the encoding we build a global set Π that contains the identifiers of all variables marked as pure. Recall that in particular, this includes the names given in function definitions. Every occurrence of a variable that is not in Π becomes wrapped inside a call to `get`. Note that this encoding fails if the invariants imposed by the declared purity of a variable are not satisfied. Note also that this encoding adds style annotations to the Optiλ terms being produced.

We can show that this encoding preserves the semantics by showing that a simulation relation is preserved between the states of an OptiC program and its corresponding encoding in Optiλ.

In order to characterize the simulation, we need to relate values, environments and memory states of the two languages. This simulation involves a *relocation map*, written ρ, that binds locations from the OptiC memory to the corresponding locations in the Optiλ memory. Pure variables do not have an address in Optiλ, and therefore the locations of such variables do not occur in ρ. We say that a relocation map ρ is compatible with an environment s, a memory m and a pure variable set Π, and we write $\rho \ltimes (s, m, \Pi)$ when the domain of ρ is exactly the set of all the locations from m except those bound in s by a pure variable in Π. Formally:

$$\rho \ltimes (s, m, \Pi) \quad \Leftrightarrow \quad \mathsf{dom}(\rho) = \mathsf{dom}(m) \setminus \{l \mid \exists x \in \Pi, s(x) = l\}$$

We are now ready to define the encoding function, written $\lfloor \cdot \rfloor_\rho$. This encoding applies to values, environments, and memory states.

For values, the encoding function traverse the structure of values, until reaching a location or a closure. For these entities, the following two rules apply.

$$\lfloor l \rfloor_\rho = \rho(l)$$

$$\lfloor [s](T_1\, a_1, ..., T_n\, a_n)\, u_f \rfloor_\rho = \mathbf{fun}^{\lfloor s \rfloor_\rho}(a_1, ..., a_n) \mapsto \lfloor u_f \rfloor_\rho$$

An environment s of OptiC is translated into an environment σ of Optiλ by translating all the values it contains. The interesting case is that of a variable x in the OptiC store s that corresponds to a pure variable in Optiλ. In the store σ of Optiλ, the pure varialbe is bound to the translation of the value stored in OptiC memory m at the address $s(x)$, that is, to the value $m(s(x))$.

$$\lfloor s \rfloor_\rho = \left\{ \begin{array}{ll} x \mapsto \lfloor m(s(x)) \rfloor_\rho & \text{if } x \in \Pi \\ x \mapsto \lfloor s(x) \rfloor_\rho & \text{otherwise} \end{array} \;\middle|\; x \in s \right\}$$

A memory m of OptiC is translated into an environment m of Optiλ by translating all the values it contains. The entries that correspond to the representation of pure variables

$$
\begin{array}{lcl}
\lfloor u \rfloor^{\&} & = & t \quad \text{where } \lfloor u \rfloor \text{ is (guaranteed to be) of the form } \mathsf{get}(t) \\
\lfloor x \rfloor & = & \left| \begin{array}{ll} x & \text{if } x \in \Pi \\ \mathsf{get}(x) & \text{otherwise} \end{array} \right. \\
\lfloor b \rfloor & = & b \\
\lfloor n \rfloor & = & n \\
\lfloor u_1 + u_2 \rfloor & = & \mathsf{add}(\lfloor u_1 \rfloor, \lfloor u_2 \rfloor) \\
\lfloor \&u \rfloor & = & \lfloor u \rfloor^{\&} \\
\lfloor *u \rfloor & = & \mathsf{get}(\lfloor u \rfloor) \\
\lfloor u_1 \texttt{ = } u_2 \rfloor & = & \mathsf{set}(\lfloor u_1 \rfloor^{\&}, \lfloor u_2 \rfloor) \\
\lfloor u_1 \texttt{ += } u_2 \rfloor & = & \mathsf{inplaceAdd}\,(\lfloor u_1 \rfloor^{\&}, \lfloor u_2 \rfloor) \\
\lfloor u_0(u_1, ..., u_n) \rfloor & = & \lfloor u_0 \rfloor(\lfloor u_1 \rfloor, ..., \lfloor u_n \rfloor) \\
\lfloor T \texttt{ const } x = u \rfloor & = & \mathbf{let}_{\lfloor T \rfloor^{\mathsf{typ}}}\ x = \lfloor u \rfloor \qquad (x \in \Pi) \\
\lfloor T\, x = u \rfloor & = & \mathbf{let}_{\mathsf{ptr}(\lfloor T \rfloor^{\mathsf{typ}})}\ x = \mathsf{ref}_{\lfloor T \rfloor^{\mathsf{typ}}}(\lfloor u \rfloor) \qquad (x \notin \Pi) \\
\lfloor T\, x \rfloor & = & \mathbf{let}_{\mathsf{ptr}(\lfloor T \rfloor^{\mathsf{typ}})}\ x = \mathsf{stackCell}_{\lfloor T \rfloor^{\mathsf{typ}}}() \qquad (x \notin \Pi) \\
\lfloor u_0 \texttt{ ? } u_1 \texttt{ : } u_2 \rfloor & = & \mathbf{if}^{?:}\ \lfloor u_0 \rfloor\ \mathbf{then}\ \lfloor u_1 \rfloor\ \mathbf{else}\ \lfloor u_2 \rfloor \\
\lfloor \mathbf{if}\ u_0\ \mathbf{then}\ u_1\ \mathbf{else}\ u_2 \rfloor & = & \mathbf{if}\ \lfloor u_0 \rfloor\ \mathbf{then}\ \lfloor u_1 \rfloor\ \mathbf{else}\ \lfloor u_2 \rfloor \\
\lfloor \mathbf{if}\ u_0\ \mathbf{then}\ u_1 \rfloor & = & \mathbf{if}\ \lfloor u_0 \rfloor\ \mathbf{then}\ \lfloor u_1 \rfloor\ \mathbf{else}\ \{\}^{\varnothing} \\
\lfloor u_1 \texttt{ \&\& } u_2 \rfloor & = & \left| \begin{array}{ll} \mathsf{and}(\lfloor u_1 \rfloor, \lfloor u_2 \rfloor) & \text{if } u_1 \text{ and } u_2 \text{ are values} \\ \mathbf{if}^{\&\&}\ \lfloor u_1 \rfloor\ \mathbf{then}\ \lfloor u_2 \rfloor\ \mathbf{else}\ \mathsf{false} & \text{otherwise} \end{array} \right. \\
\lfloor u_1 \texttt{ || } u_2 \rfloor & = & \left| \begin{array}{ll} \mathsf{or}(\lfloor u_1 \rfloor, \lfloor u_2 \rfloor) & \text{if } u_1 \text{ and } u_2 \text{ are values} \\ \mathbf{if}^{||}\ \lfloor u_1 \rfloor\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}\ \lfloor u_2 \rfloor & \text{otherwise} \end{array} \right. \\
\lfloor \{u_1; ...; u_n; q\} \rfloor & = & \{\lfloor u_1 \rfloor^{\mathsf{void}}; ...; \lfloor u_n \rfloor^{\mathsf{void}}; \lfloor q \rfloor^{\mathsf{res}}\} \\
\lfloor u \rfloor^{\mathsf{void}} & = & \left| \begin{array}{ll} \lfloor u \rfloor & \text{if } u \text{ is of type } \texttt{void} \\ \mathsf{ignore}(\lfloor u \rfloor) & \text{otherwise} \end{array} \right. \\
\lfloor \varnothing \rfloor^{\mathsf{res}} & = & \varnothing \\
\lfloor \mathbf{return}; \rfloor^{\mathsf{res}} & = & \varnothing^{\mathsf{ret}} \\
\lfloor u; \rfloor^{\mathsf{res}} & = & \left| \begin{array}{ll} x & \text{if } \lfloor u \rfloor \text{ is a variable } x \\ \mathbf{let}^{\mathsf{res}}\ x = \lfloor u \rfloor;\ x & \text{otherwise} \end{array} \right. \\
\lfloor \mathbf{return}\, u; \rfloor^{\mathsf{res}} & = & (\lfloor u; \rfloor^{\mathsf{res}})^{\mathsf{ret}} \\
\lfloor T_0\, f(T_1\, a_1, ..., T_n\, a_n)\, u_f \rfloor & = & \mathbf{let}_{(\lfloor T_1 \rfloor^{\mathsf{typ}} \times ... \times \lfloor T_n \rfloor^{\mathsf{typ}}) \to \lfloor T_0 \rfloor^{\mathsf{typ}}}\ f = \mathbf{fun}(a_1, ..., a_n) \mapsto \lfloor u_f \rfloor \\
\lfloor [\&](T_1\, a_1, ..., T_n\, a_n)\, u_f \rfloor & = & \mathbf{fun}^{\square}(a_1, ..., a_n) \mapsto \lfloor u_f \rfloor \\
\\
\lfloor T* \rfloor^{\mathsf{typ}} & = & \mathsf{ptr}(\lfloor T \rfloor)^{\mathsf{typ}} \\
\lfloor \texttt{int} \rfloor^{\mathsf{typ}} & = & \mathsf{int} \\
\lfloor \texttt{bool} \rfloor^{\mathsf{typ}} & = & \mathsf{bool} \\
\lfloor \texttt{fun<}T_0\texttt{(}T_1, ..., T_n\texttt{)>} \rfloor^{\mathsf{typ}} & = & (\lfloor T_1 \rfloor^{\mathsf{typ}} \times ... \times \lfloor T_n \rfloor^{\mathsf{typ}}) \to \lfloor T_0 \rfloor^{\mathsf{typ}}
\end{array}
$$

Figure 5. Translation from OptiC to Optiλ. A global, precomputed set Π contains the identifiers of all *pure* variables—in OptiTrust, variables carry unique identifiers in addition to their names. Superscripts on λ-terms represent style annotations for translating back to OptiC.

are dropped. These entries can be identified by the fact that the corresponding addresses are not bound in the relocation map ρ.

$$\lfloor m \rfloor_\rho = \{\rho(l) \mapsto \lfloor m(l) \rfloor_\rho \mid l \in m \cap \rho\}$$

We next define the simulation relations, which we use to state and prove the theorem capturing the correctness of the encoding. The first relation relates pairs of stores and memories of the two languages. The second relation relates output configurations of the two languages.

$$(\sigma, \mu) \sim (s, m) \quad \Leftrightarrow \quad \exists \rho. \quad \rho \ltimes (s, m, \Pi) \quad \wedge \quad \sigma = \lfloor s \rfloor_\rho \quad \wedge \quad \mu = \lfloor m \rfloor_\rho$$

$$w/^{\sigma}_{\mu} \sim v/^{s}_{m} \quad \Leftrightarrow \quad \exists \rho. \quad \rho \ltimes (s, m, \Pi) \quad \wedge \quad \sigma = \lfloor s \rfloor_\rho \quad \wedge \quad \mu = \lfloor m \rfloor_\rho \quad \wedge \quad w = \lfloor v \rfloor_\rho$$

To simplify the correctness theorems, we assume that the semantics are *completed* with error-propagation rules, in such a way that a configuration is never stuck, but can always evaluate to an error. To that end, we follow the presentation of pretty-big-step semantics [Cha13] and introduce the notion of *outcome*. The evaluation judgement for Optiλ takes the form $t/^{\sigma}_{\mu} \Downarrow \omega$, where ω is either a final configuration or an error, written err. Likewise, the evaluation judgement OptiC takes the form $u/^{s}_{m} \Downarrow o$, where o denotes a final configuration or an error. We extend our simulation relation for output configurations to consider that error states are related, that is, $\mathsf{err} \sim \mathsf{err}$.

We are now ready to state the correctness theorem. For simplicity, we focus our attention on terminating programs only.

Proposition 1 (Correctness of the encoding). For any OptiC program u, if there exists:

- σ, μ, s, m such that $(\sigma, \mu) \sim (s, m)$,
- ω and o such that either $(u/^{s}_{m} \Downarrow o) \wedge (\lfloor u \rfloor/^{\sigma}_{\mu} \Downarrow \omega)$, or $(u/^{s}_{m} \Downarrow^{\&} o) \wedge (\lfloor u \rfloor^{\&}/^{\sigma}_{\mu} \Downarrow \omega)$,

then $\omega \sim o$ holds.

At this stage, we only give the sketch of a proof for this proposition. The intuition is that it can be proven by induction on the structure of the program u and then in each case, the reduction properties can be inverted to constrain ω and o enough to be able to either show that they both are errors, or that we can derive the expected conclusion. For the induction to go through, the relocation map ρ needs to be explicited, and one needs to argue that the relocation map involved for justifying the output simulation is an extension of the relocation map involved for justifying the input simulation. To get confidence in the theorem, such a proof ought to be checked using a proof assistant. We leave this task to future work.

6 Translation from Optiλ back to OptiC

This section defines the reciprocal translation, which we call *decoding*. Figure 6 defines this decoding operation. The notation $\lceil t \rceil$ denotes the decoding of an Optiλ term t. The notation $\lceil t \rceil^{*}$ denotes an auxiliary operation for decoding terms that will appear in left-value contexts. We express the correctness property of decoding similarly as for encoding.

Proposition 2 (Correctness of decoding). For any Optiλ program t, if there exists:

- σ, μ, s, m such that $(\sigma, \mu) \sim (s, m)$,
- ω such that $t/^{\sigma}_{\mu} \Downarrow \omega$,
- o such that either $\lceil t \rceil/^{s}_{m} \Downarrow o$, or $\lceil t \rceil^{*}/^{s}_{m} \Downarrow^{\&} o$,

then $\omega \sim o$ holds.

$$
\begin{array}{lcl}
\lceil t \rceil^{*} & = & \begin{cases} u & \text{if } \lceil t \rceil \text{ is of the form } \&u \\ *\lceil t \rceil & \text{otherwise} \end{cases} \\
\lceil x \rceil & = & \begin{cases} x & \text{if } x \in \Pi \\ \&x & \text{otherwise} \end{cases} \\
\lceil b \rceil & = & b \\
\lceil n \rceil & = & n \\
\lceil \mathsf{add}(t_1, t_2) \rceil & = & \lceil t_1 \rceil + \lceil t_2 \rceil \\
\lceil \mathsf{get}(t) \rceil & = & \lceil t \rceil^{*} \\
\lceil \mathsf{set}(t_1, t_2) \rceil & = & \lceil t_1 \rceil^{*} \texttt{ = } \lceil t_2 \rceil \\
\lceil \mathsf{inplaceAdd}\,(t_1, t_2) \rceil & = & \lceil t_1 \rceil^{*} \texttt{ += } \lceil t_2 \rceil \\
\lceil t_0(t_1, ..., t_n) \rceil & = & \lceil t_0 \rceil(\lceil t_1 \rceil, ..., \lceil t_n \rceil) \\
\lceil \mathbf{let}_{\mathsf{ptr}(\tau)}\ x = \mathsf{stackCell}() \rceil & = & \lceil \tau \rceil^{\mathsf{typ}}\, x; \qquad (x \notin \Pi) \\
\lceil \mathbf{let}_{\mathsf{ptr}(\tau)}\ x = \mathsf{ref}(t) \rceil & = & \lceil \tau \rceil^{\mathsf{typ}}\, x = \lceil t \rceil; \qquad (x \notin \Pi) \\
\lceil \mathbf{let}_{\tau}\ x = t \rceil & = & \lceil \tau \rceil^{\mathsf{typ}}\, \texttt{const}\, x = \lceil t \rceil; \qquad (x \in \Pi) \\
\lceil \mathsf{and}(t_1, t_2) \rceil & = & \lceil t_1 \rceil \texttt{ \&\& } \lceil t_2 \rceil \\
\lceil \mathsf{or}(t_1, t_2) \rceil & = & \lceil t_1 \rceil \texttt{ || } \lceil t_2 \rceil \\
\lceil \mathbf{if}^{\&\&}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ \mathsf{false} \rceil & = & \lceil t_1 \rceil \texttt{ \&\& } \lceil t_2 \rceil \\
\lceil \mathbf{if}^{||}\ t_1\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}\ t_2 \rceil & = & \lceil t_1 \rceil \texttt{ || } \lceil t_2 \rceil \\
\lceil \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ \{\}^{\varnothing} \rceil & = & \mathbf{if}\ \lceil t_0 \rceil\ \mathbf{then}\ \lceil t_1 \rceil \\
\lceil \mathbf{if}^{?:}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \rceil & = & \lceil t_0 \rceil \texttt{ ? } \lceil t_1 \rceil \texttt{ : } \lceil t_2 \rceil \\
\lceil \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \rceil & = & \begin{cases} \lceil t_0 \rceil \texttt{ ? } \lceil t_1 \rceil \texttt{ : } \lceil t_2 \rceil & \text{if type is not unit} \\ \mathbf{if}\ \lceil t_0 \rceil\ \mathbf{then}\ \lceil t_1 \rceil\ \mathbf{else}\ \lceil t_2 \rceil & \text{otherwise} \end{cases} \\
\lceil \{t_1; ...; t_n; \mathbf{let}^{\mathsf{res}}\ x = t_r; x^{\mathsf{ret}}\} \rceil & = & \{\lceil t_1 \rceil; ...; \lceil t_n \rceil; \mathbf{return}\, \lceil t_r \rceil;\} \\
\lceil \{t_1; ...; t_n; \mathbf{let}^{\mathsf{res}}\ x = t_r; x\} \rceil & = & \{\lceil t_1 \rceil; ...; \lceil t_n \rceil; \lceil t_r \rceil;\} \\
\lceil \mathsf{ignore}(t) \rceil & = & \lceil t \rceil \\
\lceil \{t_1; ...; t_n; r\} \rceil & = & \{\lceil t_1 \rceil; ...; \lceil t_n \rceil; \lceil r \rceil^{\mathsf{res}};\} \\
\lceil \varnothing^{\mathsf{ret}} \rceil^{\mathsf{res}} & = & \mathbf{return};\ \varnothing \\
\lceil \varnothing \rceil^{\mathsf{res}} & = & \varnothing \\
\lceil x^{\mathsf{ret}} \rceil^{\mathsf{res}} & = & \mathbf{return}\, x; \\
\lceil x \rceil^{\mathsf{res}} & = & x; \\
\lceil \mathbf{let}_{(\tau_1 \times ... \times \tau_n) \to \tau_0}\ f = \mathbf{fun}^{A}(a_1, ..., a_n) \mapsto t_f \rceil & = & \lceil \tau_0 \rceil^{\mathsf{typ}}\, f(\lceil \tau_1 \rceil^{\mathsf{typ}}\, a_1, ..., \lceil \tau_n \rceil^{\mathsf{typ}}\, a_n)\, \lceil t_f \rceil \qquad \text{if } \square \notin A \\
\lceil \mathbf{fun}^{\square}(a_{1\,\tau_1}, ..., a_{n\,\tau_n}) \mapsto t_f \rceil & = & [\&](\lceil \tau_1 \rceil^{\mathsf{typ}}\, a_1, ..., \lceil \tau_n \rceil^{\mathsf{typ}}\, a_n)\, \lceil t_f \rceil \\
\lceil \mathsf{ptr}(\tau) \rceil^{\mathsf{typ}} & = & \lceil \tau \rceil^{\mathsf{typ}} \texttt{*} \\
\lceil \mathsf{int} \rceil^{\mathsf{typ}} & = & \texttt{int} \\
\lceil \mathsf{bool} \rceil^{\mathsf{typ}} & = & \texttt{bool} \\
\lceil (\tau_1 \times ... \times \tau_n) \to \tau_0 \rceil^{\mathsf{typ}} & = & \texttt{fun<}\lceil \tau_0 \rceil^{\mathsf{typ}}\texttt{(}\lceil \tau_1 \rceil^{\mathsf{typ}}, ..., \lceil \tau_n \rceil^{\mathsf{typ}}\texttt{)>}
\end{array}
$$

Figure 6. Translation from OptiTrust's internal λ-calculus back to C. A global, precomputed set Π contains the identifiers of all *pure* variables—in OptiTrust, variables carry unique identifiers in addition to their names. Style annotation constraint are silently ignored if no rule matches.

As mentioned earlier, during the encoding, a number of style annotations are attached to the terms produced, in order to guide the decoding phase and ensure the round-trip property. Importantly, these annotations are always ignored by the semantics. It is therfore always safe to drop annotations in the OptiTrust AST. Ignoring style annotatins may even be necessary. Consdier for example a transformation that rewrites "t_0; **if** t_c **then** $\{t_1\}$ **else** $\{\}^{\varnothing}$" into "**if** t_c **then** $\{t_0; t_1\}$ **else** $\{t_0\}^{\varnothing}$", where the symbol $\varnothing$ in the input term indicates that the *else* branch was absent from the OptiC code. There, the resulting term is a nonempty *else* branch, hence the annotation $\varnothing$ must be discarded.

There is one limitation with the current style annotation system expressed by the notion of *spurious pattern*, which consists of an occurrence of an expression of the form `&*u` or `*&u`. Such spurious patterns are eliminated during the process of encoding a program and do not generate any annotation. Even though we could design a style annotation that preserves spurious patterns, we believe that it is not worth the extra work, because spurious patterns usually do not appear in human-written source programs.

We are now ready to state the round-trip property of our bidirectional translation.

Proposition 3 (Round-trip for OptiC programs). If u is an OptiC program that does not contain spurious patterns, and such that its encoding $\lfloor u \rfloor$ is well-defined, then decoding gives back the original program: $\lceil \lfloor u \rfloor \rceil = u$.

Like in the previous section, actual mechanized proofs of the decoding correctness and the round-trip property are left for future work.

Note that this roundtrip theorem is defined at the level of the ASTs. In our current implementation, comments, blank lines, indentation, and macro expansions, are not part of the AST. Hence, they are not preserved by our translations. Indentation is not much an issue for users using code formatters. Comments and blank lines could be attached to AST nodes, as style annotation, if need be. Macros would be trickier to handle. One could imagine a best-effort algorithm, which folds back macros for parts of the AST that have not been altered by the transformations.

7 Future Work

In this paper, we have presented the semantics of the languages manipulated by our interactive optimizing compiler OptiTrust. We have described how to seamlessly translate from one the other. We showed in previous works [BCK+24] that OptiTrust can be used to reach state-of-the-art performance starting from naive implementations.

This paper focuses on the bidirectional translation from a language the resembles C code. Presumably, the ideas and translation schemes described in this paper could be applied to build bidirectional translations from other user-facing languages, such as OCaml, Rust, OpenCL/Cuda, Java, etc. Once such a translation is defined for another language, all the code transformations in OptiTrust, which are expressed on the Optiλ language, become available for this new language. Going one step further, we also envision the possibility to translate between different user-facing languages.

References

[BCK+] Guillaume Bertholon, Arthur Charguéraud, Thomas Kœhler, Begatim Bytyqi, and Damien Rouhling. Optitrust: Producing trustworthy high-performance code via source-to-source transformations. [DRAFT].

[BCK+24] Guillaume Bertholon, Arthur Charguéraud, Thomas Kœhler, Begatim Bytyqi, and Damien Rouhling. Interactive source-to-source optimizations validated using static resource analysis. In *Proceedings of the 13th ACM SIGPLAN*

International Workshop on the State Of the Art in Program Analysis, pages 26–34, 2024.

[CBG+18] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W Appel. Vst-floyd: A separation logic tool to verify correctness of c programs. *Journal of Automated Reasoning*, 61:367–422, 2018.

[CCEG23] Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. Omnisemantics: Smooth handling of nondeterminism. *ACM Transactions on Programming Languages and Systems*, 45(1):1–43, 2023.

[Cha13] Arthur Charguéraud. Pretty-big-step semantics. In *European Symposium on Programming*, pages 41–60. Springer, 2013.

[CMJ+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*. USENIX Association, 2018.

[IRKF+21] Yuka Ikarashi, Jonathan Ragan-Kelley, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. Guided optimization for image processing pipelines. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–5, 2021.

[KK22] Vasilios Kelefouras and Georgios Keramidas. Design and implementation of 2D convolution on x86/x64 processors. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3800–3815, 2022.

[Kre15] Robbert Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, 2015.

[RKBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Conference on Programming Language Design and Implementation*, 2013.

[VVAT03] Manish Vachharajani, Neil Vachharajani, David I. August, and Spyridon Triantafyllis. Compiler optimization-space exploration. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, page 204, Los Alamitos, CA, USA, mar 2003. IEEE Computer Society.