ACHILLES Resource Estimation - Walk Through

Kinan Dak Albab, Rawane Issa, Peter Flockhart, and Wyatt Howe

{babman, ra1issa, pflock, whowe}@bu.edu

Boston University

September 28, 2020

Abstract

This document provides a step-by-step walk through demonstrating how our resource estimation tool, carousels, analyzes a sample merge sort and deduplicate program, and produces numerical estimates for its overall network usage with the BGW protocol.

1 Introduction

We show the step-by-step flow that our resource estimation tool performs over the sample merge sort and deduplicate program. The sample program is shown in figure 1.

At a high level, the steps can be separated into the following categories:

1. **Configuration:** This includes inputting the program to the tool, as well as choosing the MPC protocol, and the resource whose usage will be analyzed when the program is run using that given protocol.
2. **Parsing:** The tool parses the program into its own language and protocol agnostic intermediate representation (IR).
3. **Algebraic Analysis:** The tool analyzes the parsed IR, each step of the analysis manipulates an algebraic symbolic system representing the estimated resource usage of the program. The final output consists of one or more algebraic relation, these relations may be simple, or may be recursive or co-recursive, depending on the analyzed program.
4. **Evaluation:** The tool evaluates the algebraic system for given concrete values of the parameters appearing in that system. The tool can output the result of the evaluation as numeric values or plots, according to the user's preference.

2 Configuration

The user is responsible for providing three pieces of input to the tool, as part of its configuration: (1) the input program (2) the MPC protocol (3) the resource to analyze.

```
// Entry into problem. Carousels uses this function to start estimating
fn merge_sort_dedup<T, P: Obliv>(a: &[Possession<T, P>])
-> Vec<Possession<T, P>>
{
    let n = a.len();
    if n > 1 {
        let m = n/2;
        merge_dedup(&merge_sort_dedup(&a[0..m]), &merge_sort_dedup(&a[m..n]))[0..n]
    } else {
        return a.to_owned()
    }
}
// The recursive function that does most of the work
fn merge_dedup<T, P: Obliv>(a: &[Possession<T, P>], b: &[Possession<T, P>])
-> Vec<Possession<T, P>>
{

    let alen = a.len();
    let blen = b.len();

    if alen < 1 { return b[0..blen].to_owned() }
    if blen < 1 { return a[0..alen].to_owned() }

    let p1 = a[0] < b[0];
    let p2 = a[0] > b[0];

    // For security we cannot leak if there is a duplicate or not
    // so we always add a padding value to maintain length
    let mut v = Vec::with_capacity(alen+blen);
    let first_el = obliv if p1 {
        a[0].clone()
    } else obliv if p2 {
        b[0].clone()
    } else {
        P::run(0)
    };
    v.push(first_el);

    // Recurse on the rest of the problem depending on if the first value in each array is
    // <, >, or ==
    let rest = obliv if p1 {
        merge_dedup(&a[1..alen], &b[0..blen])
    }
    else obliv if p2 {
        merge_dedup(&a[0..alen], &b[1..blen])
    }
    else {
        merge_dedup(&a[0..alen], &b[1..blen])
    };

    v.extend(rest);
    return v[0..alen+blen];
}
```

Our tool comes built in with cost definitions for several protocols: BGW, SPDZ, GC, and BGV. Users can extend the tool with additional protocol by providing their own custom configuration.

A protocol configuration is essentially a cost model over the supported language primitives. It defines a set of symbolic parameters and resources, and then provides a cost estimate for every primitive and resource, as a symbolic expression over these parameters.

2.1 Cost Model Configuration

We created cost models for the above supported protocols. Users wishing to extend the tool to support additional protocols can create such cost models for them using a variety of techniques:

1. **Experimentally:** Cost models builders can write programs using specific primitives of interest, and can benchmark these programs while recording their consumed resources of interest. This approach is especially powerful for low-level system resources, such as estimating the number of CPU cycles needed to garble a gate.

2. **Analytically:** Researchers can manually analyze primitives of a protocol, and can analytically bound its resource usage. This is particularly useful for well-studied high level resources, such as rounds of communication. Most researchers tend to publish such analytical bounds along side their protocols. This allows users of the tool to immediately use these bounds, after encoding them in the tool's format.

Another dimension to consider is what constitutes a language "primitive":

1. A minimalist approach entails only defining the cost model for the most minimal core of primitives needed by the MPC protocol, for example, + and * for BGW, or NOT, XOR and AND for GC.

2. A more expansive approach can provide the cost of other higher-level primitives as part of the tool's cost model configuration. For example, it may provide costs for comparison operators directly, or certain for matrix operations, or even complex data structures such as ORAM.

Our experiences show that there is an interesting balance here: while the first approach entails the least configuration burden, the second approach is more efficient in terms of the analysis time required by our tool. The first approach requires that the input program contain complete implementation of all operations used in it that were not considered part of the minimal set. For example, a program in BGW using the ¡ on two secrets, must contain the defintion of that operator in terms of + and *. This can dramatically increase the size of the analyzed program, as well as the complexity of the algerbraic system produced by our tool. On the other hand, the second approach requires more configuration effort as more primitives must be baked into the cost model, while allowing the input program to remain simple and concise, causing our tool to operate more efficiently.

We relied on a hybrid approach, where we baked in higher level operations into our configuration when they are commonly used, while leaving out rare operations that we only encountered in a few example programs. Note that our tool can be used to make this approach less tedious. The

implementation of an interesting higher level operation can be fed into our tool and analyzed separately, and the resulting symbolic system can be simplified and written directly into the cost model.

2.2 Cost Model Format

Our tool uses a simple JSON format to encode the cost model of a specific protocol.

The format begins by defining the relevant symbolic parameters and resources it is defined over.

```
parameters: [
    {symbol: "b", description: "the bit size of the field"},
    {symbol: "p", description: "the number of parties"},
],
metrics: [
    {
      title: "Network Bits",
      description: "Total number of bits sent by a party",
      type: "TotalMetric"
    },
    {
      title: "Network Rounds",
      description: "Number of network calls made by a party",
      type: "RoundMetric"
    },
    // ... <additional metrics> ...
  ]
```

Every resource is defined as a metric with a type. The type determines what set of program logic rules the tool uses later on when manipulating the algebraic system representing the resource usage of the program. We support to important metric types: total and round, as well as hybrid metric which consists of multiple components, each being either a total or a round metric (similar to a tuple).

Next, every primitive is identified and assigned a symbolic cost in the above paramters.

```
operations: [
    {
      rule: {
        nodeType: 'DirectExpression',
        match: '<type:number@D,secret:true>(\\*)<type:number@D,secret:true>'
      },
      value: {
        "Network Bits": "(p-1)*b",
        "Network Rounds": "1",
        // ... <other resources> ...
      }
```

```
    },
    // ... <additional primitives> ...
]
```

Note that the primitive is identifies during analysis time by matching on the rule pattern specified in its definition. For example, the above primitive is only applied to expressions on the form "x * y", where "x" and "y" are expressions of type "(number, secret)". The "@D" directive refers to any dependent type, meaning that we are not imposing any conditions on the expressions beyond that they are numeric.

The cost model configuration is only responsible for defining the cost of the primitive in isolation. The program logic encoded in the tool's implementation is responsible for aggregating the costs of various primitives used in the program.

For more complex operations, the cost can be defined as a function of the operation context, which includes the cost and type information of all its sub-expressions. For example, below we define the cost of a high level oblivious if/mux over arrays: depending on the value of the secret condition bit, the operation either returns the first or the second array obliviously. Specifically, this cost specification is applicable to the implementation of that mux operation as a loop over the two arrays, which at every iteration, applies a regular mux to the two entries for the two arrays currently pointed to by the iteration.

```
{
  rule: {
    nodeType: 'OblivIf',
    // e.g. (x == y ? arr1 : arr2)
    match: "<type:boolean@D,secret:true>" +
           "\\?<type:array@D,secret:true>:" +
           "<type:array@D,secret:true>"
  },
  value: {
    "Network Bits": function (_, _, _, childrenType, _) {
        // e.g. arrayLength = n
        const arrayLength = childrenType.ifBody.dependentType.length;
        // e.g. n * (p-1)*b
        return "(" + arrayLength + ") * (p-1) * b";
    },
    "Network Rounds": "1",
  }
}
```

3 Parsing

After the tool has been fed with the input program, protocol, and resource choices. The tool beings by parsing the program into its IR.

The tool is designed so that parsers are pluggable components. This way, the tool can support MPC programs written in a variety of libraries or languages. We implemented a parser for obliv-rust, which is used by our tool by default.

We require no restrictions on the parser, except that it exposes an API that takes the input program as a string, and produces the parsed IR object. Our tool's IR is relatively simple and generic. It does not make any language specific assumptions, and tries to be as agnostic as possible to any language internal details. Our IR is encoded via JSON. You can find a summary specification of our IR constructs at https://github.com/multiparty/carousels/blob/master/docs/ir.md.

3.1 Type-less IR

Because our IR is designed to be language agnostic. We designed our IR so that it possesses the minimal amount type information about the program. This is particularly important to be able to support dynamically typed languages (such as python). As the parser will not be able to produce such a typed-IR without performing a lot of type inference itself. In particular, the IR only specifies functions parameters and return types, and nothing else.

After our type-less IR object is parsed from a program. Our tool performs type inference on it. This way, the bulk of type inference in our tool is re-used regardless of the language being parsed. This is useful since this inference not only produces traditional type information (such as variable base types, dependent variable values and arrays lengths, etc), but also identifies secret and public values and expressions, and tracks them through out the program.

Obviously, we still need to do some tweaking to this type inference according to parsed language. For example, the expression x / y, where x and y are ints, may be typed as int in some languages (e.g. C) or float in others (e.g. Javascript). This can be conveyed to our tool by providing custom typing configuration, which is similar to the cost model configuration discussed above, but instead of assigning expressions costs, it assigns them types.

This has the added advantage of supporting library code, without having to input the library implementation to our tool. For example, imagine that the input code uses some complex graph library by making calls to some functions defined in it. Traditionally, static analysis tools will require access to the implementation of these functions, as they may need to analyze these implementation to fully characterize the program. However, our tool does not need that implementation, as long as the type and resource cost of these calls are specified using the typing and cost model configuration.

4 Algebraic Analysis

Our algebraic analysis consists of a single recursive traversal of the IR of the program. This is implemented as a visitor pattern, where we have a single function per IR node type (e.g. if statement, function call, binary expression, etc).

4.1 Dependent Types

Every function call in the recursive traversal corresponds to a (sub) expression in the IR of the program. Every call returns a type and an algebraic cost. The type is what our tool's type inference

engine and typing configuration deem is the expression type, including its base type (e.g number, array), a dependent type portion (e.g. the length of the array), and whether or not it is a secret. Both the dependent type portion, and the algerbraic cost, are symbolic expressions over program quantities (e.g. previous variables). Additionally, costs may depend on protocol parameters defined by the cost model, as well as type information. However, the types are not allowed to depend on costs.

We support the boolean, integer, float, array, range, matrix, string types. The boolean, integer, and float dependent types specify their values. The array and matrix dependent types specify their size. The range dependent type specifies its end points.

If our tool is unable to determine the dependent type of some expression, e.g. if the expression refers to some unknown and unspecified library function, the tool assigns these types fresh new symbolic parameters, which become part of what the user must provide values for during evaluation.

4.2 Scoped Contexts

Additionally, our recursive traversal maintains two contexts for variables: One mapping variables to their types, and the other mapping them to their costs. Contexts are scoped: they contain mappings for variables defined in the current scope or parent scopes. When a scope is fully analyzed, all variables belonging to it are removed from the context. Context properly overwrite and recover shadow variables that are defined in a nested scope with a matching name to a variable in a parent scope.

Furthermore, our recursive traversal maintains similar contexts for functions: The first maps a function to its signature type, and the other to their cost.

4.3 Function Abstractions

We support program with functions that are recursive or co-recursive. This means that a function that is being analyzed may call another function that we have not analyzed yet. However, we only want to analyze a single function at a time in isolation.

We address this by first assigning every function a dependent type and cost abstraction. These are similar to the dependent type and cost of an expression. However, they are only derived by looking at the function signature. Essentially, these are symbolic functions without a definition (yet).

Whenever our tool encounters a new scope. It initially finds all function definitions in it, analyzes their signatures, and produces abstractions for them, prior to it analyzing any other statements or function body.

Looking at our merge sort sample program, our tool begins by assigning the following abstractions to the two functions defined in it.

```
// <function<
//   (<array<number,length=n0>,secret:true>)
```

```
//   => <array<number,length=F0(n0)>,secret:true>
// >>
// Return Type Abstraction: F0(n0)
// Cost Abstraction: F1(m2, n0) : m2 is the cost of a
fn merge_sort_dedup<T, P: Obliv>(a: &[Possession<T, P>])
-> Vec<Possession<T, P>>

// <function<
//   (
//     <array<number,length=n5>,secret:true>,
//     <array<number,length=n8>,secret:true>
//   ) => <array<number,length=F2(n5, n8)>,secret:true>
// >>
// Return Type Abstraction: F2(n5, n8)
// Cost Abstraction: F3(m7, m10, n5, n8) : m7, m10 are costs of a, b
fn merge_dedup<T, P: Obliv>(a: &[Possession<T, P>], b: &[Possession<T, P>])
-> Vec<Possession<T, P>>
```

As shown above, the tool begins by coming up with the type signature of the function. The base types of the parameters and return types are taken from the IR. Any dependent type portion for function parameters (such as a list length) is assigned a fresh symbolic variable, for example the length of parameter a in merge_sort_dedup is set to be $n0$. The dependent type of the return type is set to a symbolic uninterpreted function of the parameters, for example, the size of the array returned by merge_sort_dedup is set to $F0(n0)$.

Similarly, the cost of a function is set to a symbolic uninterpreted function of the types and costs of its parameters. This is necessary, since the cost of the function may change as the parameter size changes. The cost of calling merge_sort_dedup with an array of size $n0$ that takes cost $m2$ to construct is assigned to $F1(m2, n0)$.

Loops: Our program handles for loops in a similar way to functions. It considers a loop to be logically equivalent to recursive function, that recurses on an iteration counter parameter. Loops do not have a return type abstraction, since they return no value, but they may modify variables defined outside the loop. Thus our tool creates a dependent type and cost abstractions for every such modified variable, as well as a cost abstraction for the loop itself. All these functions depend on the loop iteration counter, as well as the current scoped context.

4.4 Function Bodies

After coming up with abstractions for every function in the scope. Our tool begins analyzing function bodies, in order of definition, each as a distinct isolated scope. Our tool traverse the body one statement at a time. Each statement is analyzed by calling the appropriate visitor function per the statement type, which may call other functions to analyze the statements sub-expressions.

Each statement and sub-expression analyzed is first typed using our typing engine and typing configuration, then its cost is looked up in the cost model, by finding rules that apply to its type.

This cost that is looked up is considered to be the cost of that expression **in isolation**. The cost is further contextualized by aggregating it with the costs of the nested sub-expressions. This final aggregated cost is considered to be the entire cost of the statement or expression.

4.5 Aggregation Program Logics

We have covered the cost model in section 1 above. Aggregation of costs of sub-expressions is performed according to the resource metric type. In this walk-through, we are looking at network bandwidth, which has metric type "total". Intuitively, this is because network bandwidth is computed by "adding-up" the costs of all operations in a program. Another way to think about this is that network bandwidth is not context or order specific: for example, (x * y) * z + t has the same network bandwidth cost as (x * y) + (t * z), because they have the same number of operations. This is not true for other costs, such as network rounds, which have metric type "round". We show (a poriton of) our aggregation program logic rules for metric type "total" in figure 2, we also contrast them with the rules for metric type "round" shown in figure 3. We augment some rules corresponding to program constructs that are unused in our program (e.g. loops), have already been described in prose previously (function definitions), or are intuitive (e.g. ranges and slices over arrays) for the sake of brevity.

There are several important differences between the total and round aggregation logics:

1. **The cost context Γ:** in the total logic, Γ only contains mapping for function cost abstractions. It remains unchanged within a scope. However, in the round logic, it contains costs for variables. This is primarily caused by where the program is "charged" the cost of the value assigned to that variable.

   In the total logic, the cost is charged at assignment time only, to avoid double counting. However, in rounds, the cost is charged when the variable is used (the assignment is free), this does not suffer double counting because we rely on max instead of + for aggregation.

   In fact, careful examination of the round logic will reveal that the only costs ever added are either an expression cost with a primitive cost acquired from the cost model Σ (which does not double count because it is independent from the context by definition), or the sequent of the obliv-if rule. However, that rule is carefully designed to exclude the costs of variables appearing in the condition, by setting them to zero in the context when analyzing the statement branches.

2. **Function calls arguments costs:** Similarly, in the total logic, the cost of an argument to a function is charged by the function-call rule, as it gets added to the aggregate cost directly. On the other hand, in the round case, the cost of that argument is passed as an argument to the function abstraction, which is responsible for charging it.

4.6 Example: merge_sort_dedup

We demonstrate how a function body is recursively visisted and analyzed, using the cost model and the program logic. We will go through the merge_sort_dedup one step at a time, showing how

$$\frac{e : constant}{\Gamma \vdash e \to 0}(constant) \qquad \frac{e : variable}{\Gamma \vdash e \to 0}(variable) \qquad \frac{\Gamma \vdash e \to c}{\Gamma \vdash v := e \to c}(assignment)$$

$$\frac{\Gamma \vdash e_1 \to c_1,\ \Gamma \vdash e_2 \to c_2,\ \Sigma \vdash \oplus(e_1, e_2) \to c_o}{\Gamma \vdash e_1 \oplus e_2 \to c_1 + c_2 + c_o}(bin\text{-}exp)$$

$$\frac{\Gamma \vdash e_1 \to c_1,\ ..., \Gamma \vdash e_k \to c_k,\ \Gamma \vdash f \to c_f(m_1, ..., m_k, n_1, ..., n_k)}{\Gamma \vdash f(e_1, ..., e_k) \to c_1 + ... + c_k + c_f(0, ..., 0, |e_1|, ..., |e_k|)}(function\text{-}call)$$

$$\frac{\Gamma \vdash e \to c\ \Gamma \vdash e_1 \to c_1,\ ..., \Gamma \vdash e_k \to c_k,\ \Gamma \vdash f \to c_f(m, m_1, ..., m_k, n, n_1, ..., n_k)}{\Gamma \vdash e.f(e_1, ..., e_2) \to c + c_1 + c_2 + c_f(0, 0, ..., 0, |e|, |e_1|, ..., |e_k|)}(method\text{-}call)$$

$$\frac{\Gamma \vdash e \to c,\ \Gamma \vdash e_1 \to c_1,\ \Gamma \vdash e_2 \to c_2}{\Gamma \vdash \text{if } e\ \{e_1\} \text{ else } \{e_2\} \to c + (|e|\ ?\ c_1\ :\ c_2)}(if)$$

$$\frac{\Gamma \vdash e \to c,\ \Gamma \vdash e_1 \to c_1,\ \Gamma \vdash e_2 \to c_2,\ \Sigma \vdash e?e_1 : e_2 \to c_{oif}}{\Gamma \vdash \text{obliv if } e\ \{e_1\} \text{ else } \{e_2\} \to c + c_1 + c_2 + c_{oif}}(obliv\text{-}if)$$

$$\frac{\Gamma \vdash e_1 \to c_1,\ ...,\ \Gamma \vdash e_n \to c_n}{\Gamma \vdash e_1; ...; e_n \to c_1 + ... + c_n}(body)$$

Figure 2: Program Logic for Aggregating Costs of Metric Type "Total"

$$\frac{e : constant}{\Gamma \vdash e \to 0}(constant) \qquad \frac{e : variable}{\Gamma \vdash \Gamma[e]}(variable) \qquad \frac{\Gamma \vdash e \to c}{\Gamma[v \to c], \Gamma \vdash v := e \to 0}(assignment)$$

$$\frac{\Gamma \vdash e_1 \to c_1,\ \Gamma \vdash e_2 \to c_2,\ \Sigma \vdash \oplus(e_1, e_2) \to c_o}{\Gamma \vdash e_1 \oplus e_2 \to max(c_1, c_2) + c_o}(bin\text{-}exp)$$

$$\frac{\Gamma \vdash e_1 \to c_1,\ ...,\ \Gamma \vdash e_k \to c_k,\ \Gamma \vdash f \to c_f(m_0, ..., m_k, n_0, ..., n_k)}{\Gamma \vdash f(e_1, ..., e_k) \to c_f(c_1, ..., c_k, |e_1|, ..., |e_k|)}(function\text{-}call)$$

$$\frac{\Gamma \vdash e \to c\ \Gamma \vdash e_1 \to c_1,\ ...,\ \Gamma \vdash e_k \to c_k,\ \Gamma \vdash f \to c_f(m, m_1, ..., m_k, n, n_1, ..., n_k)}{\Gamma \vdash e.f(e_1, ..., e_k) \to c_f(c, c_1, ..., c_k, e, |e_1|, ..., |e_k|)}(method\text{-}call)$$

$$\frac{\Gamma \vdash e \to c,\ \Gamma[e \to 0] \vdash e_1 \to c_1,\ \Gamma[e \to 0] \vdash e_2 \to c_2}{\Gamma \vdash \text{if } e\ \{e_1\} \text{ else } \{e_2\} \to c + (|e|\ ?\ c_1\ :\ c_2)}(if)$$

$$\frac{\Gamma \vdash e \to c,\ \Gamma \vdash e_1 \to c_1,\ \Gamma \vdash e_2 \to c_2,\ \Sigma \vdash e?e_1 : e_2 \to c_{oif}}{\Gamma \vdash \text{obliv if } e\ \{e_1\} \text{ else } \{e_2\} \to max(c, c_1, c_2) + c_o}(obliv\text{-}if)$$

$$\frac{\Gamma \vdash e_1 \to c_1,\ ...,\ \Gamma[e_1 \to c_1, ...] \vdash e_n \to c_n}{\Gamma \vdash e_1; ...; e_n \to max(c_1, ..., c_n)}(body)$$

Figure 3: Program Logic for Aggregating Costs of Metric Type "Round"

the tool types and assigns costs to the various expressions in the body of that function.

Remember that the function signature has already been analyzed to create the function return and cost abstractions. During this creation, the tool assigned the function parameter "a" the type "¡array¡number,length=n0¿,secret:true¿¿.

The tool analyzes the function body one statement at a time in order. It begins with the first statement:

```
let n = a.len();
```

The tool first analyzes the expression then the assignment. Because $a.len()$ is a library function, it does not have a function abstraction in the current context. Our tool will attempt to look it up in the cost model. This reveals that this function has no cost, since it access the length of the array, which is public information. The content of the array is secret, but not its length. The tool will also look up the cost of a from the context, which will return 0 in the case of a cost of metric type total.

Putting these two costs together, the tool invoked the "method-call" program rule, resulting in an overall cost of 0.

```
// n has type: <number<value=n0>,secret:false>
// cost: 0
let n = a.len();
```

Next, the tool moves to the next statement, this is an if statement. The tool breaks it into three pieces, each analyzed independently, the condition, the then branch, and the else branch. Once all are analyzed, the tool aggregated the costs of the these pieces and produces the aggregate cost of the if statement, as per the if aggregation logic rule.

The tool begins by analyzing the condition. This is a binary expression over non-secret expressions. The tool finds the cost of the two operands using the variable and constant rules respectively (both are 0), then finds the cost of the operation itself by looking it up in the cost model.

Because this is a non-secret operation, we omit it from the cost model (we could alternatively add it there and explicitly set its cost to 0). Our tool treates all such omissions as zero costs. Therefore, the tool applies the binary expression logic rule on the three zero costs, and produces a final cost of zero for the condition.

```
// type: <boolean<value=(n0 > 1)>,secret:false>
// cost: 0
n > 1
```

The tool proceeds to the then branch, which consists of two statements. The tool analyzes these two statements in order, acquiring a cost per statement, then uses the body rule to aggregate the costs together.

The first statement is an assignment of a non-secret expression to a variable. The tool finds the expression cost to be zero by applying the same rules as in the previous steps, then applies the

assignment rule to set that statement's aggregate cost to zero.

```
// m has type: <number<value=floor(n0 / 2),secret:false>
// cost: 0
let m =
  // type: <number<value=floor(n0 / 2),secret:false>
  // cost: 0
  n / 2;
```

We use floor(n0 / 2) in the dependent type of m because of the typing configuration mentioned in section 3. Our builtin rust typing interpret / as integer division, rather than float division, to correctly model rust's semantics.

Next, the tool analyzes the second statement in the then branch. This one is a function call. The tool recursively analyzes the arguments to the call first, themselves a function call. The tool proceeds recursively to visit all sub-expressions:

```
// type: <number<value=0>,secret:false>
// cost: 0
0
// type: <number<value=floor(n0 / 2)>,secret:false>
// cost: 0
m
// type: <array<number,length=floor(n0 / 2) - 0>,secret:true>
// cost: 0
&a[0..m]
// type: <array<number,length=F0(floor(n0 / 2) - 0)>,secret:true>
// cost: F1(0, floor(n0 / 2) - 0)
&merge_sort_dedup(&a[0..m])

// similarly for the other arguments
// ...
// type: <array<number,length=F0(n0 - floor(n0 / 2))>,secret:true>
// cost: F1(0, n0 - floor(n0 / 2))
&merge_sort_dedup(&a[m..n])

// putting both arugments together with the abstractions of merge_dedup
// type: <array<number,length=F2(F0(floor(n0 / 2)), F0(n0 - floor(n0 / 2)))>,secret:true>
// cost: F3(0, 0, F0(floor(n0 / 2)), F0(n0 - floor(n0 / 2)))
```

Note how the function return and cost abstractions are utilizes when their respective function is called. The abstractions (although are uninterpreted) are instantiated with argument values deduced from the type and cost of the corresponding arguments to the function call. This particular step is the reason our resulting algebraic system will be recursive, in a way similar to how the input program itself is recursive.

Now the tool moves to the else branch. This one is another black-box rust library call. The tool looks it up in our built-in rust typing configuration for type information, and realizes it has no costs since it is omitted from the cost model.

```
// type: <array<number,length=n0>,secret:true>
// cost: 0
return a.to_owned();
```

Since all three components of the if statement have been analyzed, the tool now can apply the if aggregation logic rule to find the aggregate cost. The rule also applies a similar if rule in the typing engine to come up with the aggregate dependent type. We omitted the typing rules because they are standard and follow a similar structure to the cost aggregation rule.

```
// type: <array<number,length=(
//   n0 > 1
//   ? F2(F0(floor(n0 / 2)), F0(n0 - floor(n0 / 2)))
//   : n0
// )>,secret:true>
// cost: n0 > 1 ? F3(0, 0, F0(floor(n0 / 2)), F0(n0 - floor(n0 / 2))) : 0
if n > 1 {
    let m = n/2;
    merge_dedup(&merge_sort_dedup(&a[0..m]), &merge_sort_dedup(&a[m..n]))[0..n]
} else {
    return a.to_owned()
}
```

Because our dependent type and cost languages can encode equivalent program expression, we are able to life the if condition in the function body to an equivalent ternary operation in the dependent length and cost expressions. This allows us to estimate resource usage accurately, but at a cost in efficiency at the evaluation stage, since the tool has to evaluate more complicated algebraic equations. This is an interesting tradeoff that the users of the tool can control, as they can provide hints and annotations to override dependent type and cost expressions with simpler bounds, for example the maximum of the respective expressions from either branch.

Finally, now that all statements have been analyzed, the tool can apply the body rule again to find the overall cost of the function, and its counterpart typing rule to find the return type length expression. These two expressions are then used to interpret the cost and return type abstractions of this function.

```
// F0(n0) = n0 > 1
//   ? F2(F0(floor(n0 / 2)), F0(n0 - floor(n0 / 2)))
//   : n0
// F1(m2, n0) = n0 > 1 ? F3(0, 0, F0(floor(n0 / 2)), F0(n0 - floor(n0 / 2))) : 0
fn merge_sort_dedup<T, P: Obliv>(a: &[Possession<T, P>])
-> Vec<Possession<T, P>>
```

The tool then analyzes merge_dedup in a similar fashion, moving one statement at a time, visiting its sub-expressions recursively, and aggregating their costs and types. Until it finds the interpretation of merge_dedup's cost and return type abstractions.

```
// F2(n5, n8) = (n5 < 1 ? n8 - 0 : (n8 < 1 ? n5 : n5 + n8))
// F3(m7, m10, n5, n8) =
// n5 < 1
//  ? 0
//  : n8 < 1
//    ? 0
//    : (F2(n5, n8 - 1) + F2(n5 - 1, n8) + 6) * (p - 1) * b
//      + 6 * (b - 1 + 2) * (p - 1) * b
//      + 2 * F3(0, 0, n5 - 1, n8) + F3(0, 0, n5, n8 - 1)
fn merge_dedup<T, P: Obliv>(a: &[Possession<T, P>], b: &[Possession<T, P>])
-> Vec<Possession<T, P>>
```

This concludes the analysis phase. Now, the tool has computed an algebraic system consisting of 4 functions, each with its own interpretation. These functions are recursive, but are guaranteed to be well-defined and terminating, if the program itself is indeed terminating. Evaluating the appropriate algebraic function at some chosen arguments will yield the resource cost of the corresponding program function when run on inputs matching the chosen arguments. We discuss this more in the next section.

5 Evaluation

Now that the analysis is completed. The user has the ability to evaluate or plot the resource cost of the program. To do so, the user must provide the following input:

- The name of the function whose cost should be evaluated. Usually, this is the name of the main entry point to the program. In our example, this would be "merge_sort_dedup".
- The values of all parameters needed for evaluation. These can be either single values (to acquire a single numeric cost estimate), or ranges (for plots). The parameters here include the following:
  1. Any parameters defined by the cost model of the protocol. In our example, this include b and p, the field size and number of parties respectively.
  2. The dependent type portion of the arguments to the function to evaluate. This allows the user to evaluate the cost of the program for specific inputs, In our example. this includes the length of the array to sort and deduplicate.
  3. Any free symbolic parameters the tool created to represent types or costs of untypable expressions, specifically unknown black-box library calls. We do not have any such parameters in our example.
- Optionally, the user can choose to provide values for the cost parameters to the cost abstraction being evaluated. By default these are all set to zero. This may be useful if the user is interested in evaluating the resource usage of a function analyzed without its context.

Optionally, the user can provide alternative interpretation to any abstraction produced by the analysis. This may be useful in case the user manually simplified certain expressions that were not simplified automatically, or if the user manually solved some of the recurrences present in the algebraic system. This can also allow the user to find bounds on the resource cost by replacing abstractions with simpler bounds. These scenarios can increase the efficiency of the evaluation procedure at the cost of user interaction.

At first glance, it may seem that the algebraic system produced by our analysis is as complex as our program, and thus will require an asymptotically similar amount of time to evaluate as would the program to experimentally run on the same input parameters.

However, this is not the case. Our algebraic system certainly lifts a lot of the run-time checks and operation to the algebraic system (for the sake of increased accuracy). Nonetheless, the system can still be simplified statically, via constant folding, expression re-writing, and other simplification rules. We rely on a algebraic mathematics solver/library to preform such re-writing and simplification for us. Furthermore, we do not evaluate this system naively. We rely on memoization to make the evaluation as fast as possible.

Both re-writing and memoization succeed in making evaluating this algebraic system a lot faster than the corresponding program due to a similar underlying reason. When a function in the program is run on different inputs of similar size, the function has to run twice, without any caching or sharing of state. The bench-marking environment cannot determine what state can be cached or shared, or whether these different inputs will result in the same cost or not.

However, our tool operates in such a way that only the parameters that indeed influence the program (along some execution trace) appear in the resulting algebraic system. For example, the values of array to sort do not affect resource usage, and thus do not appear in the system. Therefore, while running the program on the two arrays [0, 1, 2] and [5, 6, 7] requires running the program twice, these two arrays are identical with respect to our algebraic system, since it only looks at the array size.

This is particularly relevant with recursive and iterative function, where a function or body of code is executed several times, on different inputs with similar characteristics with respect to our algebraic system. Since these execution can be memoized. Indeed this is the case for the merge sort program.

Consider $merge_dedup$, every call to it results in three recursive calls (two of which are identical). Each one of these calls decreases either one of the two arrays lengths. Carefully looking at this shows that the number of such calls is exponential. The depth of the smallest recursion path is n, and therefore the total number of calls is $\geq 2^n$, where n is the length of the smallest input array.

On the other hand, merge_dedup return and cost abstractions are both functions of the lengths of the two arrays, and not their exact values. Since neither arrays ever increase in size, every call has parameters in $[0, n_1] \times [0, n_2]$, so there are no more than $(n_1 + 1) \times (n_2 + 1)$ different combinations of parameters to these abstractions. Therefore, with memoization, we only need to evaluate $O(n_1 * n_2)$ calls, instead of $O(2^n)$.

Furthermore, assuming the user is interested in evaluating or plotting the resource cost at various input sizes, our tool keeps memoized entries from evaluating on one input when evaluating on other inputs, and when the same call is encountered again, the result for it can be just looked up in the cache. For our merge sort example, it means that evaluating the cost over a range of array sizes takes the same amount of time as evaluating the cost over only the largest of these sizes.