Optimizing Prestate Copies in Runtime Verification of Function Postconditions

Jean-Christophe Filliâtre, Clément Pascutto

Optimizing Prestate Copies in Runtime Verification of Function Postconditions

Jean-Christophe Filliâtre[1] and Clément Pascutto[1,2][0000−0002−5658−7731]

[1] Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France
[2] Tarides, 75005, Paris, France
jean-christophe.filliatre@cnrs.fr, clement@tarides.fr

Abstract. In behavioural specifications of imperative languages, postconditions may refer to the prestate of the function, usually with an `old` operator. Therefore, code performing runtime verification has to record prestate values required to evaluate the postconditions, typically by copying part of the memory state, which causes severe verification overhead, both in memory and CPU time.
In this paper, we consider the problem of efficiently capturing prestates in the context of Ortac, a runtime assertion checking tool for OCaml. Our contribution is a postcondition transformation that reduces the subset of the prestate to copy. We formalize this transformation, and we provide proof that it is sound and improves the performance of the instrumented programs. We illustrate the benefits of this approach with a maze generator. Our benchmarks show that unoptimized instrumentation is not practicable, while our transformation restores performances similar to the program without any runtime check.

Keywords: Runtime Assertion Checking · OCaml · Optimized Code Generation · Memory Management

1 Introduction

In behavioral specification languages for imperative languages, function postconditions may refer to the prestate of the function, typically using some `old` or `pre` operator, as in Eiffel [10], JML [3], or ACSL [2]. For instance, a function with a postcondition $x = \texttt{old}\ x + 1$ states that any call will increment the value of the variable x.

In order to perform runtime verification, one needs to be able to evaluate terms and predicates, such as `old` x above, after function calls. The prestate, which `old` refers to, does not exist anymore. As a consequence, code instrumentations have to record any value required for the evaluation of the predicates involving `old`. A correct yet naive solution consists in copying the whole prestate.

In this paper, we consider the problem of efficiently capturing prestates in the context of Gospel, a behavioural specification language for OCaml [5], and Ortac, a runtime assertion checking tool [6]. Ortac consumes a Gospel-annotated

OCaml module interface and produces an instrumented wrapper around the module implementation. It operates in a black-box fashion without inspecting the original implementation of the module.

Gospel, like OCaml, abstracts over the addresses of the values it manipulates and provides a structural equality to compare values, rather than a physical equality. While this makes the specifications easy to read and write, it makes copies immediately necessary for any term that contains mutable data, as recording their address is not sufficient to evaluate the term in the prestate: all the memory contents available from that address are necessary. For instance, if a is an array, the postcondition $a = \texttt{old}\ a$ states that the contents of array a has been restored to its prestate value. The physical address of the array is not relevant to the equality predicate. Because the function may have modified its contents, copying the whole array (in-depth, recursively) is necessary before we make the function call in the instrumented code.

Code instrumentation under these constraints can cause severe verification overhead, both in memory and CPU time. Moreover, OCaml memory management uses a garbage collector (GC), so programs do not explicitly allocate or free memory. Instead, the program triggers the GC whenever it needs additional memory. Each run incrementally traverses the memory to determine which chunks are still in use, possibly moves them and then frees the rest. Therefore, the copies introduced by Ortac induce a garbage collection overhead that adds to the memory allocation overhead. In fact, naive instrumentation not only results in high runtime verification overhead, but can also change the complexity of the algorithm, threatening its scalability.

In this work, we propose some methods to optimize the runtime verification of logical assertions containing `old` by reducing the subset of the memory that one needs to copy in order to compute these checks. We formalize the semantics of a subset of OCaml and Gospel and provide proof that these transformations are sound and improve the performance of the instrumented programs.

We start by introducing a reduced working language, along with a formalization of its semantics (Sec. 2). Then we propose some code transformations on this language to help reduce the verification overhead and allocations (Sec. 3) and show that these can be critical in practice through an example and benchmarks (Sec. 4). We conclude with related efforts toward more efficient runtime assertion checking (Sec. 5) and some insights on future work and perspectives (Sec. 6).

2 A Minimal Language with Contracts

In this section, we introduce a simple programming language to model the behaviour of Gospel-annotated OCaml code. We believe this language is generic enough to both enable detailed reasoning about semantics and memory models, and abstract away from OCaml and Gospel, so our techniques can be applied in other imperative programming languages where the same issues arise as well.

$$
\begin{array}{lll}
e ::= \texttt{()} & & \textit{unit} \\
\quad \mid n & & \textit{integer literal} \\
\quad \mid x & & \textit{variable} \\
\quad \mid \texttt{let } x_1, \ldots, x_n = e \texttt{ in } e & & \textit{variable binding} \\
\quad \mid e; e & & \textit{sequence} \\
\quad \mid (e, \ldots, e) & & \textit{tuple construction} \\
\quad \mid \pi_i(t) & & \textit{tuple getter} \\
\quad \mid \texttt{create } e\ e & & \textit{array construction} \\
\quad \mid e\texttt{.(}e\texttt{)} & & \textit{array getter} \\
\quad \mid e\texttt{.(}e\texttt{)} \leftarrow e & & \textit{array setter} \\
\quad \mid \texttt{length } e & & \textit{array length} \\
\quad \mid \texttt{copy } e & & \textit{deep copy} \\
\quad \mid \texttt{assert } e\ \{p\} & & \textit{logical assertion}
\end{array}
$$

$$
\begin{array}{lll}
p ::= t = t & & \textit{equality} \\
\quad \mid p \wedge p & & \textit{conjunction} \\
\quad \mid p \vee p & & \textit{disjunction} \\
\quad \mid \texttt{forall } i.\ t \leq i \leq t \rightarrow p & & \textit{universal} \\
\quad \mid \texttt{exists } i.\ t \leq i \leq t \wedge p & & \textit{existential}
\end{array}
$$

$$
\begin{array}{lll}
t ::= \texttt{()} & & \textit{unit} \\
\quad \mid n & & \textit{integer literal} \\
\quad \mid x & & \textit{variable} \\
\quad \mid t\texttt{.(}t\texttt{)} & & \textit{array getter} \\
\quad \mid \texttt{length } t & & \textit{array length} \\
\quad \mid \pi_i(t) & & \textit{tuple getter} \\
\quad \mid \texttt{old } t & & \textit{prestate reference}
\end{array}
$$

Fig. 1. Language syntax

Syntax. The syntax is available in Fig. 1. A program consists of an *expression*. The expression language e includes immediate values (integers and unit), as well as variables bound to immediate values or addresses in memory, that can contain mutable (arrays) or immutable (tuples) data, to reflect the variety of cases that occur in usual programming languages. Similarly to OCaml, the language does not expose any direct manipulation of addresses or any explicit memory management; allocations are implicitly made when creating a new array or tuple.

On top of these traditional programming constructs, our language provides an `assert` e $\{p\}$ instruction. This instruction models the postconditions of a specification language for logical *predicates* p. Predicates provide an equality predicate over *terms*, logical conjunction and disjunction, and existential and universal quantifiers. Finally, terms t contain immediate values, variables, and tuples and array accessors. They also feature the `old` operator that motivates this work, and which semantics is formalised in the next section.

As explained in the introduction, our main interest lies in the runtime verification of function postconditions. Although this language does not provide functions, we can model functions calls in simple scenarios of the form:

$$e_1;\ \texttt{assert}\ e_2\ \{p\}$$

In this scenario, the expression e_1 models the code that is executed prior to the function call. It sets up the memory prestate and introduces variables to refer to it. The predicate p is a postcondition to the function call, and expression e_2

models the call itself. We still operate in a black-box context, as predicate p has no access to the code e_2 itself, but solely to the resulting poststate.

Typing. Like OCaml, our language is statically typed, *i.e.* expressions and terms are statically assigned types before the evaluation. There are four primitive types τ in the system: unit (the type with only one value), integer, homogeneous arrays, and heterogeneous tuples.

$$\begin{array}{rll} \tau ::= & \texttt{unit} & \textit{(unit)} \\ | & \texttt{int} & \textit{(integer)} \\ | & \tau\ \texttt{array} & \textit{(array)} \\ | & \tau \times \tau \times \cdots \times \tau & \textit{(tuple)} \end{array}$$

We introduce a typing judgment $\Gamma \vdash e : \tau$ meaning that e has type τ in the typing environment $\Gamma ::= x \mapsto \tau$. The inference rules for this judgment are standard and should follow intuition; they are available in the appendix.

Semantics. In this section, we define a big-step semantics for our language. Program evaluations produce values v which can be the unit value, an integer, or an address in memory.

$$\begin{array}{rll} v ::= & \texttt{()} & \textit{unit} \\ | & n & \textit{integer} \\ | & a & \textit{address} \end{array}$$

Because our language is imperative, the evaluation of an expression may read or modify the *state* of the program at any point of the execution. Program states associate variables to values on one hand (function V), and addresses in memory to sequences of values that represent arrays or tuples (function M). Note that V is immutable as variables are immutable.

$$\begin{aligned} V &::= x \mapsto v \\ M &::= a \mapsto [v, v, \ldots, v] \\ S &::= V \times M \end{aligned}$$

For the sake of conciseness, we simplify the notation such that $S = (V, M)$ is always assumed, *e.g.* V (resp. V', resp. V_1) is the variable function associated to the state S (resp. S', resp. S_1) in the rest of the article.

We use notation $S, e \rightsquigarrow M', v$ to denote that the evaluation of the expression e in the state S succeeds and produces the value v in a new memory M'. The big-step evaluation rules are simple and also follow intuition; they are available in full in the appendix. We highlight a couple of rules here: E-CREATE and E-GET, which demonstrate how expressions can interact with the memory, by reading or allocating, and E-ASSERT, which shows how program expressions and logical

predicates interact with each other.

$$\frac{\begin{array}{cc} S, e_2 \rightsquigarrow M_2, v_2 & (V, M_2), e_1 \rightsquigarrow M_1, n \\ a \notin dom(M_1) & M' = M_1[a \mapsto [v_0 = v, v_1 = v, \ldots, v_{n-1} = v]] \end{array}}{S, \texttt{create}\ e_1\ e_2 \rightsquigarrow M', a} \quad \text{(E-Create)}$$

$$\frac{\begin{array}{cc} S, e_2 \rightsquigarrow S_2, n_0 & (V, M_2), e_1 \rightsquigarrow M_1, a \\ 0 \leq n_0 \leq n-1 & M_1(a) = [v_0, \ldots, v_{n-1}] \end{array}}{S, e_1\,.\,(e_2) \rightsquigarrow M_1, v_{n_0}} \quad \text{(E-Get)}$$

The `assert` e $\{p\}$ construct models the verification of the logical postcondition p of the code e.

$$\frac{S, e \rightsquigarrow M', v \qquad S, (V, M') \models p}{S, \texttt{assert}\ e\ \{p\} \rightsquigarrow M', \texttt{()}} \quad \text{(E-Assert)}$$

For its evaluation to succeed in state S, the evaluation of e in S must succeed and lead to a state S', and the predicate p must hold with prestate S and poststate V, M'. In the following, we define $S, S' \models p$. It is straightforward for most predicate constructs, but requires care to properly handle structural equality.

Predicate evaluation and equality semantics. An interesting specificity of the Gospel language is the semantics of its equality predicate. In fact, the logical domain of predicates and terms is not aware of addresses at all; we reason directly on the *contents* of the memory instead of their location. This follows OCaml's idioms, as addresses tend to be hidden to the developers and the standard library provides a polymorphic, structural equality. In particular, this means that comparing arrays `a` and `b` with the Boolean expression `a = b` will compare the contents of the arrays (recursively if necessary), rather than their addresses in memory.

Our programming language also gives this semantics to the equality predicate. Terms and predicates do not understand program values (which contain addresses); instead, they manipulate *logical values*, where addresses are recursively resolved to their contents (arrays or tuples).

$$\begin{array}{rll} lv ::= & \texttt{()} & \textit{unit} \\ | & n & \textit{integer} \\ | & [lv, lv, \ldots, lv] & \textit{array or tuple} \end{array}$$

We provide resolution rules to transition from *values* to *logical values* in a given memory. When v resolves to lv in memory M, we note $M, v \looparrowright lv$.

$$\frac{}{M, \texttt{()} \looparrowright \texttt{()}} \quad \text{(R-Unit)} \qquad \frac{}{M, n \looparrowright n} \quad \text{(R-Int)}$$

$$\frac{\begin{array}{c} M(a) = [v_0, v_1, \ldots, v_{n-1}] \\ M, v_0 \looparrowright lv_0 \qquad M, v_1 \looparrowright lv_1 \qquad \ldots \qquad M, v_{n-1} \looparrowright lv_{n-1} \end{array}}{M, a \looparrowright [lv_0, lv_1, \ldots, lv_{n-1}]} \quad \text{(R-Addr)}$$

The rule for the equality predicate is now straightforward: two terms are equal iff they evaluate to the same logical value. We note $[\![t]\!]_S^{S'}$ to denote the evaluation of the term t with prestate S and poststate S'.

$$\frac{[\![t_1]\!]_S^{S'} = lv_1 \qquad [\![t_2]\!]_S^{S'} = lv_2 \qquad lv_1 = lv_2}{S, S' \models t_1 = t_2} \quad \text{(P-Equal)}$$

The logical value resolution also lets us axiomatize the program function `copy` as follows. Any function that implements this specification qualifies for the soundness proofs we provide. We note $M \sqsubseteq M_c$ to denote that M is a subset of $M_c : \forall x, x \in dom(M) \implies x \in dom(M_c) \wedge M(x) = M_c(x)$.

Definition 1 (Copy axiomatization). *The evaluation of* `copy` *always succeeds:*

$$S, \texttt{copy}\ t \rightsquigarrow M_c, v'$$

with $M \sqsubseteq M_c$.

Moreover, the resulting value resolved to the same logical value as the copied one. In other words, if $S, t \rightsquigarrow M, v$ *and* $M, v \looparrowright lv$, *then*

$$M_c, v' \looparrowright lv$$

Term evaluation. When evaluating terms, the semantics is similar to the one of expressions, but they now apply to logical values rather than program values. The value resolution needs to be applied whenever a program variable is referenced in a term, so the evaluation now returns a logical value. Recall that terms are evaluated in the poststate of `assert` expressions, so we use S' to fetch the values in the context.

$$\frac{V'(x) = v \qquad M', v \looparrowright lv}{[\![x]\!]_S^{S'} = lv} \quad \text{(T-Var)}$$

Note that variables are immutable, *i.e.* when considering `assert` e $\{p\}$, expression e does not change the variable bindings for p (see E-Assert). In other words, $V = V'$, so picking one or the other does not make any difference. However, the memory M may be modified by e (*e.g.* when using assignment or `create`), so resolving the values in M' is crucial. Consider for instance `assert` $a.(0) \leftarrow 1$ $\{a = b\}$: the program values for variables a and b are the same in the pre- and poststate (the arrays are not moved in memory) but the contents of a has been modified and thus evaluating $a = b$ indeed requires the poststate.

While terms are generally evaluated in the poststate, the `old` operator lets you refer to the prestate. The semantics is expressed by evaluating the term in the prestate S only, rather than in the couple S, S'.

$$\frac{[\![t]\!]_S^S = lv}{[\![\texttt{old}\ t]\!]_S^{S'} = lv} \quad \text{(T-Old)}$$

Note that because of this semantic, `old` captures the logical values bound to variables in the prestate (arbitrarily big values), rather than the program values

(simple addresses). The other rules for the judgments $S, S' \models p$ and $[\![t]\!]_S^{S'}$ are straightforward and can be found in their full version in the appendix.

A consequence of rules T-Var and T-Old is that `old` can always be propagated downwards to the variables. For instance,

$$[\![\texttt{old}\ (x.(0) + y) - 1]\!]_S^{S'} = [\![(\texttt{old}\ x).(0) + \texttt{old}\ y - 1]\!]_S^{S'}$$

Surprisingly, this is not what we are going to do. We are rather going to do the exact opposite!

3 Capturing the Prestate

In this section, we present two program transformations that enable the evaluation of predicates involving prestate captures. Our transformations operate on constructs `assert` e $\{p\}$. To do so, they can inspect predicate p but not expression e. This constraint is consistent with our idea of modelling function calls with the `assert` construct. For the sake of simplicity, we suppose that `old` terms in p are not nested, *i.e.* in every term of the form `old` t, term t does not contain any other `old` operator. A quick transformation consisting in simply removing any `old` in those terms ensures this property and is trivially correct considering the rule T-Old we discussed previously.

We first discuss a transformation that introduces the copied data in the memory and discuss implementation tactics for an optimized `copy` function (Sec. 3.1). Then, we present a transformation of the predicates that reduces the memory space that needs to be copied (Sec.3.2).

3.1 Introducing Copies

Our first program transformation introduces the copies necessary for the execution of the terms containing `old`. This operation, which we note T_c, performs a morphism over the program expressions, and transforms the `assert` expressions so that the predicate does not contain any prestate reference anymore. We note $x_1, \ldots, x_n$ fresh variables (that are not bound in any state), so that we don't introduce collisions with existing data.

$$\begin{aligned} T_c(\ \texttt{assert}\ e\ \{p\}\ ) := \ & \texttt{let}\ \ x_1, \ldots, x_n = \texttt{copy}\ (t_1, \ldots, t_n)\ \ \texttt{in} \\ & \texttt{assert}\ T_c(e)\ \{p[\texttt{old}\ t_i \leftarrow x_i]\} \end{aligned}$$

$$\begin{aligned} T_c(\quad \texttt{length}\ e \quad) &:= \texttt{length}\ T_c(e) \\ T_c(\quad e_1\texttt{.(}e_2\texttt{)} \quad) &:= T_c(e_1)\texttt{.(}T_c(e_2)\texttt{)} \\ T_c(\quad \cdots \quad) &:= \cdots \textit{ (similarly for other constructs)} \end{aligned}$$

Instead, the terms t_i under `old` (note that these are also valid *expressions* since they do not contain other `old`) are evaluated in the program space, prior to the assertion, and their result is copied into the variables x_i. The `old` t_i terms in the predicate of the assertion are then substituted with the copied values x_i. For instance, consider the following example, where a and b are two arrays of arrays.

```
assert a.(0) <- b { length (old a.(0)) = (old a.(1)).(2) }
```

The program is transformed into the following one:

```
let x1 = copy (a.(0), a.(1)) in
assert a.(0) <- b { length x1 = x2.(2) }
```

Soundness. This transformation is sound, meaning that it leaves the program semantics (and in particular the validity of the `assert` constructs) unchanged.

Theorem 1 (T_c preserves the program semantics). *If a program e successfully evaluates to v_1 in S_0, which logically resolves to lv in the resulting state,*

$$S_0, e \rightsquigarrow M_1, v_1 \text{ and } M_1, v_1 \twoheadrightarrow lv$$

then for any state S'_0 such that $S_0 \sqsubseteq S'_0$, the transformed program $T_c(e)$ successfully evaluates to v'_1 in S'_0, which resolves to the same logical value lv in the resulting state,

$$S'_0, T_c(e) \rightsquigarrow M'_1, v'_1 \text{ and } M'_1, v'_1 \twoheadrightarrow lv$$

and we have $S_1 \sqsubseteq S'_1$.

Proof. We prove this theorem by induction on the number of `old` contained in the transformed expression e. Because of space limitations, we will only show the `assert` case here, as it contains the critical postcondition verification. The proofs for the other expression cases follow the same structure.

Case `assert`. We know $S_0, \texttt{assert}\ e\ \{p\} \rightsquigarrow M_1, v_1$ and $M_1, v_1 \twoheadrightarrow lv$ and $S_0 \sqsubseteq S'_0$.

Lemma 1 (Substitutions in predicates). *Given two program states such that $S_0 \sqsubseteq S_1$, in which v_0 and v_1 resolve to the same logical value, i.e.,*

$$M_0, v_0 \twoheadrightarrow lv \text{ and } M_1, v_1 \twoheadrightarrow lv,$$

binding a variable x to either v_0 or v_1 in a predicate evaluation does not change the validity judgment, that is

$$V_0[x \leftarrow v_0] \models p \implies V_0[x \leftarrow v_1] \models p.$$

The idea for the proof of this lemma is simple: since the predicate and term evaluation only manipulate logical values, substituting identical logical values does not change their evaluation.

We introduce $x_1, \ldots, x_n$ some fresh variables. We now prove the following three sub-goals:

$$S_0, \texttt{copy}\ (t_1, \ldots, t_n) \rightsquigarrow M_c, a \text{ with } M_c(a) = [v_{c_0}, v_{c_1}, \ldots, v_{c_n}] \tag{1}$$

$$(V_0[x_i \to v_{c_i}], M_c), T_c(e) \rightsquigarrow M'_1, () \tag{2}$$

$$(V_0[x_i \to v_{c_i}], M_c), M'_1 \models p[\texttt{old}\ t_i \leftarrow x_i] \tag{3}$$

We note lv_{c_i} the logical values corresponding to v_{c_i}: $M_c, v_i \twoheadrightarrow lv_{c_i}$.

1. This goal follows directly from our axiomatization of the `copy` function in Def. 1. We also get that $M_0 \sqsubseteq M_c$, and $lv_i = lv_{c_i}$, which we will use for the next goals.
2. For this goal, we apply the induction hypothesis on $T_c(e)$, as e contains strictly less `old` expressions than `assert` e $\{p\}$.
3. This result is obtained by applying Lem.1, since $lv_i = lv_{c_i}$ under Def. 1.

We may now apply the rule E-Assert to the goals (2) and (3) and get the following result:

$$(V_0[x_i \to v_{c_i}], M_c), \texttt{assert}\ T_c(e)\ \{p[\texttt{old}\ t_i \leftarrow x_i]\} \rightsquigarrow M_1', \texttt{()}$$

Now we can conclude the proof by applying the rule E-Let-In to this result, and the one provided by (1), which gives us

$$S_0', T_c(\texttt{assert}\ e\ \{p\}) \rightsquigarrow M_1', \texttt{()}$$

since

$$\begin{aligned} T_c(\texttt{assert}\ e\ \{p\}) := \texttt{let}\ & x_1, \ldots, x_n = \texttt{copy}\ (t_1, \ldots, t_n)\ \texttt{in} \\ & \texttt{assert}\ T_c(e)\ \{p[\texttt{old}\ t_i \leftarrow x_i]\} \end{aligned}$$

And $M_1', \texttt{()} \not\rightsquigarrow \texttt{()}$ trivially holds under R-Unit. □

Copying with sharing. A first optimization rises from the observation that the copied terms $t_1, \ldots, t_n$ may refer to overlapping portions of the memory. Therefore, deep-copying them recursively as described previously results in copying the same memory chunks multiple times. For instance, in the example described previously, the `assert` leads to copying $a.(0)$ and $a.(1)$ independently. However, these values may be aliases, which would lead to copying the underlying array twice.

In order to avoid duplicating the copies of memory chunks, we use a sharing-preserving implementation of `copy`: the underlying memory structure of the copied value is maintained in the copy, and shared values are only copied once. We can then copy all the required sub-terms simultaneously in a tuple, so aliases in the original data remain aliases in the copy. In practice, we use the OCaml serialization module `Marshal` to encode, and then immediately decode, the tuple of values.

Although the tuple construction introduces an extra allocation, this cost has shown to be negligible compared to the gain provided by the sharing preservation, both between sub-terms, and inside sub-terms themselves. This ensures that we copy shared chunks of memory only once, regardless of the aliasing configuration.

Copy of immutable data. The second optimisation of the `copy` function is provided by the type information. Recall that our language provides mutable arrays, but also immutable tuples. We perform an analysis based on the expression types in order to determine whether the evaluation of an expression results in

a mutable or an immutable value. A value is immutable iff its type consists of immediate values, or tuples of immutable types, that is, in other words, if its type does not involve arrays. For those types, the language does not provide any mutating functions, so the relevant memory cannot be changed by the evaluation of an expression.

$$\frac{}{immutable(\texttt{unit})}\ \text{(I-Unit)} \qquad \frac{}{immutable(\texttt{int})}\ \text{(I-Int)}$$

$$\frac{immutable(\tau_1) \quad immutable(\tau_2) \quad \ldots \quad immutable(\tau_n)}{immutable(\tau_1 \times \tau_2 \times \ldots \times \tau_n)}\ \text{(I-Tuple)}$$

When we apply the transformation, we use this inference to determine if a call to `copy` is required, *i.e.* if the value contains mutable components. If not, we simply bind the value in the prestate to a variable.

3.2 Moving `old` Upwards to Copy Less

When the prestate reference is a sub-term of the total term, it is interesting to consider *how* this sub-term will be used in the rest of the evaluation. Consider for instance the simple postcondition `length (old` a`)` when a is an array. After applying the transformation exposed in Sec. 3.1, the evaluation will copy the whole array a. Instead, we could have computed its length directly in the prestate and only remember this value for the evaluation in the poststate.

Although the semantics of the terms containing prestate references virtually pushes the `old` operator downwards to variables (see Sec. 2), the second transformation we propose actually suggests the opposite. Rather than considering what values we *need* to capture in the prestate in order to compute the poststate, we try to push as many computations as we can in the prestate, by determining which terms *cannot* be computed in the prestate.

This new transformation, which we note T_o, is meant to be applied before T_c. It starts from the existing `old` sub-terms and propagates the `old` operator upwards in the terms until it encounters a variable that refers to the poststate, with the exception of immutable values. It is defined in terms of the following rewriting rules, written $T_o(t) \sim t$, until no further rewriting is possible (Fig. 2). The rule (O-Var) only applies for program variables, not variables introduced by quantifiers[3].

Proof of soundness We prove that $\sim$ preserves the terms semantics. Since T_o follows this relation, T_o also preserves the program semantics.

[3] In Ortac, we perform a more aggressive transformation, which moves `old` beyond quantifiers when possible. In this paper, however, we keep the presentation simple by limiting the transformation to terms.

$$\frac{}{\texttt{()} \sim \texttt{old ()}}\ \text{(O-Unit)} \qquad \frac{}{n \sim \texttt{old}\ n}\ \text{(O-Int)}$$

$$\frac{t_1 \sim t_1' \qquad t_2 \sim t_2'}{t_1\texttt{.(}t_2\texttt{)} \sim \texttt{old}\ (t_1'\texttt{.(}t_2'\texttt{)})}\ \text{(O-Get)} \qquad \frac{t \sim \texttt{old}\ t'}{\pi_i(t) \sim \texttt{old}\ (\pi_i(t'))}\ \text{(O-Pi)}$$

$$\frac{t \sim \texttt{old}\ t'}{\texttt{length}\ t \sim \texttt{old}\ (\texttt{length}\ t')}\ \text{(O-Length)} \qquad \frac{}{\texttt{old}\ t \sim \texttt{old}\ t}\ \text{(O-Old)}$$

$$\frac{t \sim \texttt{old}\ t'}{\texttt{old}\ t \sim \texttt{old}\ t'}\ \text{(O-Old-Id)} \qquad \frac{\Gamma \vdash x : \tau \qquad \mathit{immutable}(\tau)}{x \sim \texttt{old}\ x}\ \text{(O-Var)}$$

Fig. 2. The $\sim$ relation defining T_o.

Theorem 2 ($\sim$ preserves the terms semantics). *For all states S and S' such that $S \sqsubseteq S'$,*

$$t \sim t' \implies [\![t]\!]_S^{S'} = [\![t']\!]_S^{S'}$$

Proof. The proof of this theorem is straightforward for most rules. The rules always add or move `old` operators, and these only modify the semantics of a term if it implies variables, as discussed in Sec. 2. Therefore, we only consider the proof for the case O-Var here.

Lemma 2 (*immutable* captures immutability). *If a term has an immutable type, then evaluating this term in the prestate only does not change its semantics: if $S \sqsubseteq S'$, then*

$$\frac{\Gamma \vdash t : \tau \qquad \mathit{immutable}(\tau)}{[\![t]\!]_S^{S'} = [\![t]\!]_S^S}$$

The idea of the proof for this lemma is provided by the constructions of the language. Recall that immutable types are the ones that do not involve arrays, but only immediate values and tuples. There is no construct that allows us to modify a tuple, and the type-checking ensures that one cannot use the array setter on an address corresponding to a tuple.

Let us note $lv = [\![x]\!]_S^{S'}$. We know $\Gamma \vdash t : \tau$ and $\mathit{immutable}(\tau)$. We can conclude by applying Lem 2 and T-Old.

$$\dfrac{\dfrac{\Gamma \vdash t : \tau \qquad \mathit{immutable}(\tau)}{[\![x]\!]_S^S = [\![x]\!]_S^{S'}}\ \text{Lem. 2}}{[\![\texttt{old}\ x]\!]_S^{S'} = [\![x]\!]_S^{S'}}\ \text{T-Old}$$

□

Proof of optimisation. It is important to note that, since the term constructions only access the program memory but never modify it, this transformation can only reduce (or leave unchanged) the memory space involved in the evaluation of the term. More precisely, the transformed program only requires to access a subset of the memory addresses it originally needed, and therefore the amount of copied data is no greater.

4 Example and Benchmarks

In this section, we use our optimisation techniques and apply them to an existing program, implemented in OCaml and annotated with Gospel specifications. We demonstrate that a simple program and specification are sufficient for this optimisation to be critical to the performance and usability of the program.

4.1 A Maze Generator

Our stress-test is a program that takes an integer n as input and generates a perfect, random maze on a n^2 square grid. The algorithm is as follows:

1. Create a list of all walls and create a set for each cell (each set contains just that one cell).
2. For each wall, in some random order,
   - if the cells divided by this wall belong to distinct sets,
     (a) remove the current wall from the list;
     (b) join the sets of the formerly divided cells.

The set of sets of cells maintain the connected components of the grid, so at the end of each iteration, we remove a wall iff it joins otherwise disconnected components. At the end of the iterations, there is only one remaining connected component; therefore, the remaining walls in the list constitute a perfect maze.

We implement the set of cells involved in this algorithm using a union-find data structure [1]. Our implementation of the union-find exposes the interface reproduced in Fig. 3, which we instrument to verify at runtime.

The type `t` is the type representing an instance of the data structure. Our module will be operating in place, so this type is mutable, which is reflected by the Gospel clause `ephemeral`.

The function `create` creates a fresh structure containing integers in singletons, `num_classes` returns the number of disjoint sets in the data structure, and `find` returns the representative element of a set. These three functions do not perform any effects (*i.e.* they do not modify the union-find structure), do not raise exceptions, and always terminate. Therefore, they are considered `pure` by Gospel and can be used to specify other functions further.

Finally, the function `union` performs the union of two sets in the structure. We will focus on this function in the rest of this benchmark. Its contract states that it can modify the data structure with the `modifies` clause. Because the type of union-find is mutable, and this function potentially modifies it, executing properties that refer to the old version of the structure will require copies, and the transformations we proposed in Sec. 3 are relevant in this example.

```
type t
(*@ ephemeral *)

val create : int -> t
(*@ uf = create n
    checks n >= 0 *)

val num_classes : t -> int
(*@ pure *)

val find : t -> int -> int
(*@ pure *)

val union : t -> int -> int -> unit
(*@ union uf i j
    modifies uf
    requires 0 <= i < size uf
    requires 0 <= j < size uf
    ensures num_classes uf <= num_classes (old uf)
    ensures find (old uf) i <> find (old uf) j
            -> num_classes uf = num_classes (old uf) - 1 *)
```

Fig. 3. Union-find module interface (`uf.mli`).

4.2 Runtime Verification with Ortac

We use Ortac to generate OCaml code that checks these contracts at runtime. More precisely, the generated implementation performs the following operations:

1. Check the preconditions and fail if they do not hold or raise an exception.
2. Evaluate the terms under `old` operators, and copy their values into fresh variables.
3. Call the function `union` and fail if it raises an exception.
4. Replace the terms precomputed in step 2 with their value in the postconditions and check them, then fail if they do not hold or raise an exception.

Unoptimised version. In the unoptimized version, the specifications are considered as they were written by the user. There are four occurences of the `old` operator, and all four of them refer to the old version of `uf`. The generated code is of the following form:

```
let union uf i j =
  if not (0 <= i <= size uf) then fail ();
  if not (0 <= j <= size uf) then fail ();
  let old_1, old_2, old_3, old_4 = copy (uf, uf, uf, uf) in
  (try union uf i j with _ -> fail ());
  if not (num_classes uf <= num_classes old_1) then fail ();
```

```
7   if not (not (find old_2 i <> find old_3 j)
8           || num_classes uf = num_classes old_4 - 1)
9   then fail ();
```

When the `copy` function preserves sharing (see Sec. 3.1), the copy operation on line 4 only copies `uf` once, and `old_1`, `old_2`, `old_3`, and `old_4` are aliases. This does not allocate memory for every occurence of `uf`, and does not re-explore the memory either. In fact, this is equivalent to just copying `uf` once in a fresh variable and using this variable for each occurrence.

Optimized version. In the optimised version, although the user can still write the specifications in the way that feels the most natural to them, `ortac` preprocesses the terms to propagate the `old` operator, as explained in Sec. 3.2. Ortac automatically rewrites the terms as if the user wrote the following postconditions:

```
1 ensures num_classes uf <= old (num_classes uf)
2 ensures old (find uf i <> find uf j)
3         -> num_classes uf = old (num_classes uf - 1)
```

This rewriting effectively moves to the prestate some computations previously executed in the poststate. Therefore, it only triggers a copy of the result of the computations (two integers and one Boolean in this case) instead of the context necessary for the execution (here, the whole union-find structure). The instrumentation generated by `ortac` now has the following form:

```
1  let union uf i j =
2    if not (0 <= i <= size uf) then fail ();
3    if not (0 <= j <= size uf) then fail ();
4    let old_1, old_2, old_3 = copy (
5      num_classes uf,
6      find uf i <> find uf j,
7      num_classes uf - 1)
8    in
9    (try union uf i j with _ -> fail ());
10   if not (num_classes uf <= old_1) then fail ();
11   if not (not old_2 || num_classes uf = old_3) then fail ();
```

4.3 Benchmarks

We run our maze generator with multiple values of n, and for each value, we gather the execution time, the number of garbage collections, and the cumulative amount of data copied by `copy` during the whole maze generation. We present the results in Fig. 4. These results show that naive instrumentations of the code make it impracticable for large values of n, which timed out after one hour of execution. On the other hand, the optimised version significantly reduces the cost of the verifications to a constant factor no larger than 2. This is permitted by the limited amount of data copied and limited use of the GC, which can be costly.

n	**Instrumentation**	**Time (s)**	**GC runs**	**Copies (MB)**
100	None	0.002 6	0	-
	No optimization	8.6	952	763
	Shared copies	2.0	260	190
	+ `old` propagation	0.006	0	0.038
200	None	0.012	0	-
	No optimization	120	13 333	12 207
	Shared copies	30	4 444	3 050
	+ `old` propagation	0.032	2	0.15
400	None	0.088	1	-
	No optimization	2 100	58 664	195 315
	Shared copies	680	31 860	48 829
	+ `old` propagation	0.19	2	0.61
800	None	0.46	4	-
	No optimization	∞	∞	∞
	Shared copies	∞	∞	∞
	+ `old` propagation	0.89	4	2.4
1600	None	2.2	5	-
	No optimization	∞	∞	∞
	Shared copies	∞	∞	∞
	+ `old` propagation	3.9	5	9.8
3200	None	11	5	-
	No optimization	∞	∞	∞
	Shared copies	∞	∞	∞
	+ `old` propagation	19	6	39

Fig. 4. The results were obtained by running our benchmarks on an i7-1165G7 @ 2.80GHz CPU, with 16GB of RAM using the OCaml 4.14.0 compiler. Each value is obtained as the average of 10 runs.

About complexity. Recall that the maze generation calls `union` until there is only one remaining set (*i.e.* exactly $n^2 - 1$ times), so its complexity when invoked with size n is $O(n^2 \times uf(n))$, where $uf(n)$ is the complexity of `union`. When union-find is properly implemented, $uf(n) = O(\alpha(n)) \approx O(1)$, so the complexity of the maze generation is $O(n^2)$ in the un-instrumented version.

However, when copying the entire union-find structure (no optimisation and shared copies only), the instrumented `union` now needs to copy a structure of size n^2. This makes the total maze generation complexity $O(n^4)$, which is not practicable. Finally, the `old` propagation optimisation does not require copying this much data but instead copies a fixed amount at each call (two integers and one Boolean), so the original complexity of the program is restored.

5 Related Work

The efficient evaluation of `old` terms in runtime assertion checking is a well-known and difficult problem, for which there is still room for improvement. In the general case, most tools copy the whole memory state before the call to the function [7, 11], while acknowledging the flaws of this approach.

ACSL [2] generalises the `old` feature by introducing an `\at(t, L)` operator, that lets the user specify arbitrary locations `L` in the code, rather than restricting it to the function prestate. This leads to possibly worse performance issues with even more states being captured. While initial implementations of E-ACSL [13] used to only perform a shallow copy of the variable contents, which is incorrect in most cases, more recent implementations provide a hybrid method to reduce the copied memory space [12], but this approach has not been detailed yet.

It is also worth mentioning that, as noted in [4], in the presence of preconditions, the evaluation and copy of the `old` terms are meant to be guarded by these preconditions. Accordingly, Ortac only evaluates those once the corresponding preconditions are successfully verified.

Previous work have also explored other optimizations for runtime assertion checkers, for instance providing efficient representation of integers [8] or improving the verification of `modifies` clauses [9]. Regarding the former, Ortac benefits from `zarith`, which only switches to arbitrary-precision integers when machine integers are not large enough. Regarding the latter, Ortac assumes that user-provided `modifies` clauses are correct and even uses them to optimize the copies. Verifying such clauses is still future work for Ortac.

6 Conclusion and Future Work

In this paper, we have presented the optimizations performed by Ortac, a runtime assertion checking tool for OCaml, to mitigate the cost of copying prestate values in postcondition verification. We showed the benefits of this approach with proof and a practical evaluation.

This paper simplified the programming and logical languages compared to the actual implementation in OCaml and Gospel to make the presentation amenable. Ortac goes beyond this paper. First, it moves `old` upwards in predicates as well, including quantifiers, local variables, and user-defined predicates and functions. Second, Gospel includes a `modifies` clause, which Ortac uses to know whether it can move a `old` beyond a program variable. When doing so, Ortac assumes that all aliases in input variables correctly appear in user-provided `modifies` clauses.

There are several perspectives to extend this work. First, OCaml values do not carry any type information at runtime. Therefore, the copy function cannot use the information that a strict subterm has some immutable type to avoid the copy and keep a pointer to the original value instead. In the future, we plan to implement a smarter type-directed copy function in Ortac, which will save even more space. Second, we have assumed in this paper that the evaluation of a logical term does not allocate memory. This hypothesis is a key in the proof

of optimization of T_o. However, several logical functions in the Gospel standard library do allocate memory in practice. We plan to evaluate heuristics to resolve the trade-off between moving `old` upwards and then allocating because of the function and stopping there at the cost of copying more.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.: Data Structures and Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1983)
2. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C specification language.
3. Chalin, P., Kiniry, J., Leavens, G., Poll, E.: Beyond assertions: Advanced specification and verification with JML and ESC/Java2. vol. 4111, pp. 342–363 (11 2005)
4. Chalin, P., Rioux, F.: Jml runtime assertion checking: Improved error reporting and efficiency using strong validity. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008: Formal Methods. pp. 246–261. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
5. Charguéraud, A., Filliâtre, J.C., Lourenço, C., Pereira, M.: GOSPEL — Providing OCaml with a Formal Specification Language. In: FM 2019 - 23rd International Symposium on Formal Methods. Porto, Portugal (Oct 2019)
6. Filliâtre, J.C., Pascutto, C.: Ortac: Runtime assertion checking for ocaml (tool paper). In: Feng, L., Fisman, D. (eds.) Runtime Verification. pp. 244–253. Springer International Publishing, Cham (2021)
7. Kosiuczenko, P.: An abstract machine for the old value retrieval. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) Mathematics of Program Construction. pp. 229–247. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
8. Kosmatov, N., Maurica, F., Signoles, J.: Efficient runtime assertion checking for properties over mathematical numbers. In: Deshmukh, J., Ničković, D. (eds.) Runtime Verification. pp. 310–322. Springer International Publishing, Cham (2020)
9. Lehner, H., Müller, P.: Efficient runtime assertion checking of assignable clauses with datagroups. In: Rosenblum, D.S., Taentzer, G. (eds.) Fundamental Approaches to Software Engineering. pp. 338–352. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
10. Meyer, B.: Applying "Design by Contract". Computer **25**(10), 40–51 (Oct 1992)
11. Petiot, G., Botella, B., Julliand, J., Kosmatov, N., Signoles, J.: Instrumentation of annotated c programs for test generation. In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation. pp. 105–114. Institute of Electrical and Electronics Engineers Inc., Victoria, Canada (Sep 2014)
12. Signoles, J.: The E-ACSL Perspective on Runtime Assertion Checking. In: International Workshop on Verification and mOnitoring at Runtime EXecution. VORTEX 2021: Proceedings of the 5th ACM International Workshop on Verification and mOnitoring at Runtime EXecution, (online), Denmark (Jul 2021)
13. Signoles, J., Kosmatov, N., Vorobyov, K.: E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper). In: RV-CuBES (2017)

A Term Semantics

In the following, $[\![t]\!]_S^{S'}$ denotes the logical value of term t in prestate S and the poststate S'.

$$\frac{}{[\![\texttt{()}]\!]_S^{S'} = \texttt{()}} \quad \text{(T-Unit)} \qquad \frac{}{[\![n]\!]_S^{S'} = n} \quad \text{(T-Int)}$$

$$\frac{V(x) = v \qquad M, v \looparrowright lv}{[\![x]\!]_S^{S'} = lv} \quad \text{(T-Var)}$$

$$\frac{[\![t]\!]_S^{S'} = [lv_0, lv_1, \ldots, lv_{n-1}]}{[\![\texttt{length } t]\!]_S^{S'} = n} \quad \text{(T-Length)} \qquad \frac{[\![t]\!]_S^{S'} = (lv_1, lv_2)}{[\![\texttt{fst } t]\!]_S^{S'} = lv_1} \quad \text{(T-Fst)}$$

$$\frac{[\![t]\!]_S^{S'} = (lv_1, lv_2)}{[\![\texttt{snd } t]\!]_S^{S'} = lv_2} \quad \text{(T-Snd)} \qquad \frac{[\![t]\!]_S^{S} = lv}{[\![\texttt{old } t]\!]_S^{S'} = lv} \quad \text{(T-Old)}$$

B Predicate Semantics

In the following, $S, S' \models p$ means that predicate p holds in state S and prestate S'.

$$\frac{[\![t_1]\!]_S^{S'} = lv_1 \qquad [\![t_2]\!]_S^{S'} = lv_2 \qquad lv_1 = lv_2}{S, S' \models t_1 = t_2} \quad \text{(P-Equal)}$$

$$\frac{S, S' \models p_1 \qquad S, S' \models p_2}{S, S' \models p_1 \wedge p_2} \quad \text{(P-And)} \qquad \frac{S, S' \models p_1}{S, S' \models p_1 \vee p_2} \quad \text{(P-Or-Left)}$$

$$\frac{S, S' \models p_2}{S, S' \models p_1 \vee p_2} \quad \text{(P-Or-Right)}$$

$$\frac{\begin{array}{c}[\![t_1]\!]_S^{S'} = n_1 \qquad [\![t_2]\!]_S^{S'} = n_2 \\ \forall j, n_1 \leq j \leq n_2 \implies S(V[i \to j], M), (V'[i \to j], M') \models p\end{array}}{S, S' \models \texttt{forall } i, t_1 \leq i \leq t_2 \to p} \quad \text{(P-Forall)}$$

$$\frac{\begin{array}{c}[\![t_1]\!]_S^{S'} = n_1 \qquad [\![t_2]\!]_S^{S'} = n_2 \\ \exists j, n_1 \leq j \leq n_2 \wedge (V[i \to j], M), (V'[i \to j], M') \models p\end{array}}{S, S' \models \texttt{exists } i, t_1 \leq i \leq t_2 \wedge p} \quad \text{(P-Exists)}$$

C Program Semantics

In the following, $S, e \rightsquigarrow M', v$ means that the evaluation of the expression e in the state S succeeds and produces the value v in a new memory M'.

$$\frac{}{S, n \rightsquigarrow M, n} \quad \text{(E-Int)} \qquad \frac{}{S, \texttt{()} \rightsquigarrow M, \texttt{()}} \quad \text{(E-Unit)}$$

$$\frac{V(x) = v}{S, x \rightsquigarrow M, v} \quad \text{(E-Var)}$$

$$\frac{\begin{array}{c} S, e_1 \rightsquigarrow M_1, a \qquad M_1(a) = [v_0, v_1, \ldots, v_n] \\ (V[x_i \to v_i], M_1), e_2 \rightsquigarrow M_2, v \end{array}}{S, \texttt{let } x_1, x_2, \ldots, x_n = e_1 \texttt{ in } e_2 \rightsquigarrow M_2, v} \quad \text{(E-Let-In)}$$

$$\frac{S, e_1 \rightsquigarrow M_1, \texttt{()} \qquad (V, M_1), e_2 \rightsquigarrow M_2, v_2}{S, e_1; e_2 \rightsquigarrow M_2, v_2} \quad \text{(E-Seq)}$$

$$\frac{\begin{array}{c} S, e_n \rightsquigarrow M_n, v_n \qquad \ldots \qquad (V, M_3), e_2 \rightsquigarrow M_2, v_2 \\ (V, M_2), e_1 \rightsquigarrow M_1, v_1 \\ a \notin dom(M_1) \qquad M' = M_1[a \mapsto [v_1, v_2, \ldots v_n]] \end{array}}{S, (e_1, e_2, \ldots, e_n) \rightsquigarrow M', a} \quad \text{(E-Tuple)}$$

$$\frac{S, e \rightsquigarrow M', a \qquad M'(a) = [v_0, v_1, \ldots, v_{n-1}]}{S, \pi_i(e) \rightsquigarrow M', v_i} \quad \text{(E-Pi)}$$

$$\frac{\begin{array}{c} S, e_2 \rightsquigarrow M_2, v \qquad (V, M_2), e_1 \rightsquigarrow M_1, n \\ a \notin dom(M_1) \qquad M' = M_1[a \mapsto [v, v, \ldots, v]] \end{array}}{S, \texttt{create } e_1 \ e_2 \rightsquigarrow M', a} \quad \text{(E-Create)}$$

$$\frac{\begin{array}{c} S, e_2 \rightsquigarrow S_2, n_0 \qquad (V, M_2), e_1 \rightsquigarrow M_1, a \\ M_1(a) = [v_0, \ldots, v_{n-1}] \qquad 0 \leq n_0 < n \end{array}}{S, e_1\texttt{.(}e_2\texttt{)} \rightsquigarrow M_1, v_{n_0}} \quad \text{(E-get)}$$

$$\frac{\begin{array}{c} S, e_3 \rightsquigarrow M_3, v \qquad (V, M_3), e_2 \rightsquigarrow M_2, n_0 \qquad (V, M_2), e_1 \rightsquigarrow M_1, a \\ 0 \leq n_0 < n \qquad M_1(a) = [v_0, v_1, \ldots, v_n] \\ M' = M_1[a \mapsto [v_0, \ldots, v_{n_0-1}, v, v_{n_0+1}, \ldots, v_{n-1}]] \end{array}}{S, e_1\texttt{.(}e_2\texttt{)} \leftarrow e_3 \rightsquigarrow M', \texttt{()}} \quad \text{(E-set)}$$

$$\frac{S, e \rightsquigarrow M', a \qquad M'(a) = [v_0, \ldots, v_{n-1}]}{S, \texttt{length } e \rightsquigarrow M', n} \quad \text{(E-length)}$$

$$\frac{S, e \rightsquigarrow M', v' \qquad M', v' \looparrowright lv \qquad (M'' \backslash M'), v'' \looparrowright lv}{S, \texttt{copy } e \rightsquigarrow M'', v''} \quad \text{(E-copy)}$$

$$\frac{S, e \rightsquigarrow M', \texttt{()} \qquad S, (V, M') \models p}{S, \texttt{assert } e \ \{p\} \rightsquigarrow M', \texttt{()}} \quad \text{(E-assert)}$$

D Typing rules

In this section, rules are common for the shared subset of constructs between terms and program expressions. For the sake of clarity, we do not repeat those rules. We note $\Gamma \vdash p$ to denote that the predicate p is well typed in the environment Γ. The rules for this judgment are standard and omitted.

$$\frac{}{\Gamma \vdash n : \texttt{int}} \quad (\textsc{Ty-Int}) \qquad \frac{}{\Gamma \vdash \texttt{()} : \texttt{unit}} \quad (\textsc{Ty-Unit})$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad (\textsc{Ty-Var})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2 \times \ldots \times \tau_n \qquad \Gamma, x_i \mapsto \tau_i \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x_1, x_2, \ldots, x_n = e_1\ \texttt{in}\ e_2 : \tau} \quad (\textsc{Ty-Let-In})$$

$$\frac{\Gamma \vdash e_1 : \texttt{unit} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1; e_2 : \tau} \quad (\textsc{Ty-Seq})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \ldots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash (e_1, e_2, \ldots, e_n) : \tau_1 \times \tau_2 \times \ldots \times \tau_n} \quad (\textsc{Ty-Tuple})$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \times \ldots \times \tau_n}{\Gamma \vdash \pi_i(e) : \tau_i} \quad (\textsc{Ty-Pi})$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{create}\ e_1\ e_2 : \tau\ \texttt{array}} \quad (\textsc{Ty-Create})$$

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{array} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\texttt{.(}e_2\texttt{)} : \tau} \quad (\textsc{Ty-Get})$$

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{array} \qquad \Gamma \vdash e_2 : \texttt{int} \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1\texttt{.(}e_2\texttt{)} \leftarrow e_3 : \texttt{unit}} \quad (\textsc{Ty-Set})$$

$$\frac{\Gamma \vdash e : \tau\ \texttt{array}}{\Gamma \vdash \texttt{length}\ e : \texttt{int}} \quad (\textsc{Ty-Length}) \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{copy}\ e : \tau} \quad (\textsc{Ty-Copy})$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{old}\ e : \tau} \quad (\textsc{Ty-Old}) \qquad \frac{\Gamma \vdash e : \texttt{unit} \qquad \Gamma \vdash p}{\Gamma \vdash \texttt{assert}\ e\ \{p\} : \texttt{unit}} \quad (\textsc{Ty-Assert})$$