A practical Separation Logic typechecker

Guillaume Bertholon Arthur Charguéraud

Inria & Université de Strasbourg

June 10th, 2024

High-level specification of the typechecking algorithm

Input of the typechecker

- Program written in an imperative λ-calculus
- Annotated with:
  - function contracts
  - loop invariants
  - ghost code for shifting view on resources

Output of the typechecker

- Checks the validity of provided contracts
- Annotates the program with:
  - at every program point: the resources available
  - on every subterm: the set of resources it uses/consumes/produces

Usefulness of a SL typechecker

Objective 1: Code verification

Check properties on the code by checking that contracts are satisfied.
Contracts can be:

- Lightweight:
  Only about ownership and shape of the data
  A bit like typing of Rust
- Full functional correctness:
  Like what is done in tools such as VeriFast or Why3
- Anywhere between the two

Objective 2: Static information for the compiler

Get useful information for further compilation passes.
In that regard, similar to Abstract Interpretation but:

- More expressivity and more complete analysis
- At the price of being more demanding for the user

Challenges of a practical SL typechecker

- Challenge 1: Minimize the resource annotation effort for the user
- Challenge 2: Build summaries useful for compilation

  Ex: Be able to tell if a resource is never written on a scope
- Challenge 3: Check that non-determinism is harmless:
  - evaluation order of subexpressions
  - parallel loop execution
- Challenge 4: Keep the performance cost of analysis low

Structure of this talk

Section 1

Basic rules of the typechecker

Syntactic conventions

- Our implementation of the typechecker takes annotated C code as input
- Internally, we encode C into an internal imperative λ-calculus. In particular:
  - There is no notion left-value in the internal representation
  - Operators are regular function
- Our typechecker could theoretically also be used on different programming languages

Function contracts

Example of function annotation

```
float array_write(float* A, int i, float v) {
  __requires("n: int, H: in_range(i, 0..n)");
  __modifies("A ⤳ Array(n)");
  ⟨A : ptr_float, i : int, v : float, n : int, H : i ∈ 0..n | A ⤳ Array(n)⟩
  A[i] = v;
  ⟨A : ptr_float, i : int, v : float, n : int, H : i ∈ 0..n | A ⤳ Array(n)⟩
  ⇒ A ⤳ Array(n)
}
```

`A[i] = v` is internally encoded as $set(A \boxplus_{float} i, v)$

Definition of function contracts

type	pre-condition	post-condition
pure	`__requires`	`__ensures`
linear	`__consumes`	`__produces`

Each clause can contain multiple resources optionally named.
`__modifies` corresponds to both `__consumes` and `__produces` for the same resources.

Variables inside `__requires` clauses scope over the rest of the contract.
Variables inside `__ensures` clauses scope over the `__produces` scopes.

```
__requires("n1: int, n1 > 0");
__consumes("A ⇝ Array(n1)");
__ensures("n2: int, n2 < n1");
__produces("A ⇝ Matrix2(n1, n2)");
```

Resource contexts

Pure resources

Variable (code or ghost)	`x: int, y: ℤ`
Pure arithmetic fact	`H: x < 2 * y`
Alias definition	`z: int := 1 + y`

Linear resources

Memory cell	`x ⇝ Cell`
Sequence of contiguous memory cells	`for i in 0..n -> A[i] ⇝ Cell`

Two dimensional matrix (in C layout)	`A ⇝ Matrix2(m, n)`
	`:= for i in 0..m -> for j in 0..n ->`
	`A[n * i + j] ⇝ Cell`

Read-only resource	`RO(a, H)`
Write-only resource	`Uninit(H)`

Typing jugements

Our typechecker verifies Hoare triples of the form:

$$\{\langle E \mid F \rangle\}\ t^{\Delta}\ \{\langle E' \mid F' \rangle\}$$

Input

- t is the typechecked term
- E is the pure context available before typechecking t.
- F is the linear context available before typechecking t.

Output

- t^{Δ} is the term t decorated with usage summaries
- E' is the pure contexts after the execution of t (contains resources from E except those no longer relevant).
- F' is the linear contexts after the execution of t.

Our algorithm computes resources available at each program point with a top-down pass.
Our algorithm computes the summaries in a bottom-up pass.

Typing sequences

$\langle a : int,\ p : ptr \mid p \rightsquigarrow \mathrm{Cell} \rangle$

$\textbf{let}\ x =$

$get(p)$

$;$

$ref\,(a + x)$

Typing sequences

$\langle a : int, p : ptr \mid p \rightsquigarrow \mathrm{Cell} \rangle$

let x =

$\langle a : int, p : ptr \mid p \rightsquigarrow \mathrm{Cell} \rangle$

$get(p)$

;

$ref\,(a + x)$

Typing sequences

$\langle a : int, p : ptr \mid p \rightsquigarrow \mathrm{Cell} \rangle$

let x =

$\langle a : int, p : ptr \mid p \rightsquigarrow \mathrm{Cell} \rangle$

$get(p)$

$\langle a : int, p : ptr, \mathbf{res} : int \mid p \rightsquigarrow \mathrm{Cell} \rangle$

;

$ref\,(a + x)$

Special variable **res** denotes the result value of a term.

Typing sequences

$\langle a : int,\ p : ptr \mid p \rightsquigarrow \mathsf{Cell} \rangle$

let x =

$\quad\langle a : int,\ p : ptr \mid p \rightsquigarrow \mathsf{Cell} \rangle$

$\quad get(p)$

$\quad\langle a : int,\ p : ptr,\ \mathbf{res} : int \mid p \rightsquigarrow \mathsf{Cell} \rangle$

;

$\langle a : int,\ p : ptr,\ x : int \mid p \rightsquigarrow \mathsf{Cell} \rangle$

$ref(a + x)$

Special variable **res** denotes the result value of a term.

Typing sequences

$$\langle a : int, p : ptr \mid p \leadsto \text{Cell} \rangle$$

$$\textbf{let } x =$$

$$\quad \langle a : int, p : ptr \mid p \leadsto \text{Cell} \rangle$$

$$\quad get(p)$$

$$\quad \langle a : int, p : ptr, \textbf{res} : int \mid p \leadsto \text{Cell} \rangle$$

;

$$\langle a : int, p : ptr, x : int \mid p \leadsto \text{Cell} \rangle$$

$$ref(a + x)$$

$$\langle a : int, p : ptr, x : int, \textbf{res} : ptr \mid p \leadsto \text{Cell}, \textbf{res} \leadsto \text{Cell} \rangle$$

Special variable **res** denotes the result value of a term.

Typing rules for sequences

VAL

$$\frac{\Gamma.\mathrm{pure} \vdash v : \tau}{\{\Gamma\}\ v\ \{\Gamma \star [\mathbf{res} : \tau := v]\}}$$

LET

$$\frac{\{\Gamma_0\}\ t\ \{\Gamma_1\} \qquad \Gamma_2 = \mathrm{Rename}\{\mathbf{res} := x\}(\Gamma_1)}{\{\Gamma_0\}\ \mathbf{let}\ x = t\ \{\Gamma_2\}}$$

SEQ

$$\frac{\begin{array}{c}\forall i \in [1, n].\quad x_i \text{ fresh} \quad \wedge \quad \{\Gamma_{i-1}\}\ t_i\ \{\Gamma'_i\} \quad \wedge \quad \Gamma_i = \mathrm{Rename}\{\mathbf{res} := x_i\}(\Gamma'_i) \\ \Gamma_r = \begin{cases}\mathrm{Rename}\{x_i := \mathbf{res}\}(\Gamma_n) & \text{if } t_i \text{ is of the form ``}\mathbf{let\ res} = t'_i\text{''} \\ \Gamma_n & \text{otherwise}\end{cases}\end{array}}{\{\Gamma_0\}\ \left(t_1; \ldots; t_n\right)\ \{\Gamma_r\}}$$

Typing function calls

Function definition

```
void f(float* A, int i) {
  __requires("m: int, n: int")
  __requires("Pi: in_range(i, 0..n)");
  __modifies("A ~> Matrix2(m, n)");
  ...
}
```

$$\gamma = \begin{cases} \text{pre.pure} = \begin{array}{l} A : ptr,\ i : int, \qquad \text{(arguments)} \\ m : int,\ n : int,\ Pi : i \in 0..n \end{array} \\ \text{pre.linear} = HA : A \leadsto Matrix2(m, n) \\ \text{post.pre} = \varnothing \\ \text{post.linear} = HA' : A \leadsto Matrix2(m, n) \end{cases}$$

Function call

```
...
f(M, 12); __with("m := 24");
...
```

$$\langle f : func,\ Sf : Spec(f, \gamma),\ M : ptr \mid HM : M \leadsto Matrix2(24, 32)\rangle$$

$$f(M, 12)_{[m:=24],[HA' \mapsto HM']}$$

$$\langle f : func,\ Sf : Spec(f, \gamma),\ M : ptr \mid HM' : M \leadsto Matrix2(24, 32)\rangle$$

Syntactically: $A := M$, $i := 12$, $m := 24$, by unification: $n := 32$, $HA := HM$, arithmetic check: Pi

Typing rules for functions

Fun

$$\frac{\{[\Gamma_0.\mathsf{pure}] \star \gamma.\mathsf{pre}\}\ t\ \{\Gamma_1\} \qquad \Gamma_1 \Rightarrow \gamma.\mathsf{post} \qquad T_f = \ulcorner(T_1, \ldots, T_n) \to T_r\urcorner}{\{\Gamma_0\}\ \left(\mathbf{fun}(a_1 : T_1, \ldots, a_n : T_n)_\gamma : T_r \mapsto t\right)\ \{\Gamma_0 \star [\mathbf{res} : T_f, \mathsf{Spec}(\mathbf{res}, \gamma)]\}}$$

App

$$\frac{\begin{array}{c}\Gamma_0 \ni \mathsf{Spec}(f, \gamma) \qquad [a_1, \ldots, a_n] = \mathsf{Args}(\gamma) \\ \mathbf{Some}\ (\sigma', \Gamma_f) = \Gamma_0 \ominus \mathsf{Specialize}_{\Gamma_0}\{\overline{a_i := x_i}^{i\in[1,n]}, \sigma\}(\gamma.\mathsf{pre}) \\ \mathrm{dom}(\rho) = \mathrm{dom}(\gamma.\mathsf{post}) \qquad \mathrm{im}(\rho) \cap \mathrm{dom}(\Gamma_0) = \varnothing \\ \Gamma_q = \mathsf{CloseFracs}(\Gamma_f \star \mathsf{Rename}\{\rho\}(\mathsf{Subst}\{\overline{a_i := x_i}^{i\in[1,n]}, \sigma, \sigma'\}(\gamma.\mathsf{post})))\end{array}}{\{\Gamma_0\}\ f(x_1, \ldots, x_n)_{\sigma,\rho}\ \{\Gamma_q\}}$$

Section 2

Read-only and write-only permissions

Read-only resources

```
// Asks two read-only resources
void g(int* x, int* y) {
    __reads("x ~> Cell, y ~> Cell");
    printf("%d\n", *x + *y);
}
```

```
void f(int* x) {
    __modifies("x ~> Cell");
    g(x, x); // We need two copies of x
             // in read-only mode
    *x += 1; // Here, we need to write
             // in x
}
```

We use the standard technique in Separation Logic: αH is the fraction α of the resource H.

$$H = 1H \qquad \alpha H = (\alpha - \beta)H \star \beta H \qquad \text{where } \alpha \in]0;1] \text{ and } \beta \in]0;\alpha[$$

`__reads("H")` is equivalent to the combination
`__requires("a: frac"); __modifies("RO(a, H)")` for a fresh a.

Fractions could be generalized to reason about operations on lock-free concurrent datastructures

Semi-automatic management of fractional permissions

```
void g(int* x, int* y) {
    __requires("a: frac, b: frac");
    __modifies("RO(a, x ⇝ Cell)");
    __modifies("RO(b, y ⇝ Cell)");
    printf("%d\n", *x + *y);
}
```

```
void f(int* x) {
    __modifies("x ⇝ Cell");

    // Need to provide:
    // RO(?a, x ⇝ Cell), RO(?b, x ⇝ Cell)
    g(x, x);

    // Need to provide:
    // x ⇝ Cell
    *x += 1;
}
```

Semi-automatic management of fractional permissions

```
void g(int* x, int* y) {
    __requires("a: frac, b: frac");
    __modifies("RO(a, x ⇝ Cell)");
    __modifies("RO(b, y ⇝ Cell)");
    printf("%d\n", *x + *y);
}
```

```
void f(int* x) {
    __modifies("x ⇝ Cell");
    x ⇝ Cell


    g(x, x);



    *x += 1;
}
```

Semi-automatic management of fractional permissions

```
void g(int* x, int* y) {
    __requires("a: frac, b: frac");
    __modifies("RO(a, x ⇝ Cell)");
    __modifies("RO(b, y ⇝ Cell)");
    printf("%d\n", *x + *y);
}
```

```
void f(int* x) {
    __modifies("x ⇝ Cell");
    x ⇝ Cell
    (1 − α)(x ⇝ Cell) ⋆ α(x ⇝ Cell)

    g(x, x);



    *x += 1;
}
```

Automatically carve subfractions when needed

Semi-automatic management of fractional permissions

```
void g(int* x, int* y) {
    __requires("a: frac, b: frac");
    __modifies("RO(a, x ⇝ Cell)");
    __modifies("RO(b, y ⇝ Cell)");
    printf("%d\n", *x + *y);
}
```

```
void f(int* x) {
    __modifies("x ⇝ Cell");
    x ⇝ Cell
    (1 − α)(x ⇝ Cell) ⋆ α(x ⇝ Cell)
    (1 − α − β)(x ⇝ Cell) ⋆ α(x ⇝ Cell) ⋆ β(x ⇝ Cell)
    g(x, x);


    *x += 1;
}
```

Automatically carve subfractions when needed

Semi-automatic management of fractional permissions

```
void g(int* x, int* y) {
    __requires("a: frac, b: frac");
    __modifies("RO(a, x ⇝ Cell)");
    __modifies("RO(b, y ⇝ Cell)");
    printf("%d\n", *x + *y);
}
```

```
void f(int* x) {
    __modifies("x ⇝ Cell");
```

$x \rightsquigarrow \mathrm{Cell}$

$(1-\alpha)(x \rightsquigarrow \mathrm{Cell}) \star \alpha(x \rightsquigarrow \mathrm{Cell})$

$(1-\alpha-\beta)(x \rightsquigarrow \mathrm{Cell}) \star \alpha(x \rightsquigarrow \mathrm{Cell}) \star \beta(x \rightsquigarrow \mathrm{Cell})$

```
    g(x, x);
```

$(1-\alpha-\beta)(x \rightsquigarrow \mathrm{Cell}) \star \alpha(x \rightsquigarrow \mathrm{Cell}) \star \beta(x \rightsquigarrow \mathrm{Cell})$

```
    *x += 1;
}
```

Automatically carve subfractions when needed

Semi-automatic management of fractional permissions

```
void g(int* x, int* y) {
    __requires("a: frac, b: frac");
    __modifies("RO(a, x ~> Cell)");
    __modifies("RO(b, y ~> Cell)");
    printf("%d\n", *x + *y);
}
```

```
void f(int* x) {
    __modifies("x ~> Cell");
    x ~> Cell
    (1 − α)(x ~> Cell) ⋆ α(x ~> Cell)
    (1 − α − β)(x ~> Cell) ⋆ α(x ~> Cell) ⋆ β(x ~> Cell)
    g(x, x);
    (1 − α − β)(x ~> Cell) ⋆ α(x ~> Cell) ⋆ β(x ~> Cell)
    (1 − α)(x ~> Cell) ⋆ α(x ~> Cell)

    *x += 1;
}
```

Automatically carve subfractions when needed

After calls, repeatedly apply the CloseFracs rewriting rule:

$$(\alpha - \beta_1 - \ldots - \beta_n)H \star (\beta_i - \gamma_1 - \ldots - \gamma_m)H = (\alpha - \beta_1 - \ldots - \beta_{i-1} - \gamma_1 - \ldots - \gamma_m - \beta_{i+1} - \ldots - \beta_n)H$$

Semi-automatic management of fractional permissions

```
void g(int* x, int* y) {
    __requires("a: frac, b: frac");
    __modifies("RO(a, x ~> Cell)");
    __modifies("RO(b, y ~> Cell)");
    printf("%d\n", *x + *y);
}
```

```
void f(int* x) {
    __modifies("x ~> Cell");
    x ~> Cell
    (1 − α)(x ~> Cell) ⋆ α(x ~> Cell)
    (1 − α − β)(x ~> Cell) ⋆ α(x ~> Cell) ⋆ β(x ~> Cell)
    g(x, x);
    (1 − α − β)(x ~> Cell) ⋆ α(x ~> Cell) ⋆ β(x ~> Cell)
    (1 − α)(x ~> Cell) ⋆ α(x ~> Cell)
    x ~> Cell
    *x += 1;
}
```

Automatically carve subfractions when needed

After calls, repeatedly apply the CloseFracs rewriting rule:

$$(\alpha - \beta_1 - \ldots - \beta_n)H \star (\beta_i - \gamma_1 - \ldots - \gamma_m)H = (\alpha - \beta_1 - \ldots - \beta_{i-1} - \gamma_1 - \ldots - \gamma_m - \beta_{i+1} - \ldots - \beta_n)H$$

Write-only permissions

Dually to read-only permissions we may want to ensure a variable is never read until it is written to.

```
void f() {
  __pure();
  int* const k = malloc(sizeof(int));
  // malloc cannot return k ⇝ Cell:
  // reading in k is undefined behavior
  // instead malloc produces Uninit(k ⇝ Cell)
  reset(k);
  free(k);
}
```

```
// This function consumes k in write-only mode
int reset(int* k) {
  __consumes("Uninit(k ⇝ Cell)");
  __produces("k ⇝ Cell");
  // or just: __writes("k ⇝ Cell");
  *k = 0;
  return *k;
}
```

We add a new modality for uninitialized resources with the rules:

$$H \Rightarrow \mathrm{Uninit}(H) \qquad \{\mathrm{Uninit}(x \rightsquigarrow \mathrm{Cell})\}\ \mathit{set}(x, v)\ \{x \rightsquigarrow \mathrm{Cell}\}$$

Other application of Uninit: check never read after

Check a variable value is never read

```
int foo() {
  ...
  x ⇝ Cell or Uninit(x ⇝ Cell)
  *x = 1;
  x ⇝ Cell
  // If the rest of the code types with the weaker permission Uninit(x ⇝ Cell)
       then *x = 1 is dead code
  ...
}
```

Other application of Uninit: check never read after

Check a variable value is never read

```
int foo() {
  ...
  x ⤳ Cell or Uninit(x ⤳ Cell)
  *x = 1;
  x ⤳ Cell
  __ghost(forget_init, "x ⤳ Cell");
  Uninit(x ⤳ Cell)
  ...
}
```

In practice, adding a ghost and retypecheck can help to find semantic properties of the code.

Section 3

Typechecking loops

Loop contracts by example

$x \rightsquigarrow \mathsf{Cell}$

```
for (int i = 0; i < n; ++i) {
  __smodifies("x ~> Cell"); // sequentially
  (*x)++;
}
```

$\bigstar_{i \in 0..n}\ t[i] \rightsquigarrow \mathsf{Cell}$

```
for (int i = 0; i < n; ++i) {
  __xmodifies("&t[i] ~> Cell"); // in parallel (
      exclusive)
  t[i]++;
}
```

$\mathsf{Uninit}(\bigstar_{i \in 0..n}\ t[i] \rightsquigarrow \mathsf{Cell})$

```
for (int i = 0; i < n; ++i) {
  __xconsumes("Uninit(&t[i] ~> Cell)");
  __xproduces("&t[i] ~> Cell");
  // or just __xwrites("&t[i] ~> Cell");
  t[i] = 0;
}
```

$\left(\bigstar_{i \in 0..n}\ t[i] \rightsquigarrow \mathsf{Cell}\right) \star \alpha \left(\bigstar_{i \in 0..n}\ s[i] \rightsquigarrow \mathsf{Cell}\right)$

```
for (int i = 0; i < n; ++i) {
  __xmodifies("&t[i] ~> Cell");
  __xreads("&s[i] ~> Cell");
  t[i] += s[i];
}
```

$\left(\bigstar_{i \in 0..n}\ t[i] \rightsquigarrow \mathsf{Cell}\right) \star \alpha(y \rightsquigarrow \mathsf{Cell})$

```
for (int i = 0; i < n; ++i) {
  __xmodifies("&t[i] ~> Cell");
  __sreads("y ~> Cell");
  t[i] += *y;
}
```

No **`__smodifies`** $\Rightarrow$ parallelizable
$=$ safe concurrent execution of iterations

Loop contracts

Split the loop invariants in four parts

- Resources exclusive to one iteration that can be transformed by the loop (loop contract)
- Resources in a sequential invariant that are passed from one iteration to the next
- Read-only resources shared because they are split just before entering the loop
- Some pure variables abstractions that scope over all the rest of the loop contract

type	global	iteration contract		shared across iterations	
		pre-condition	post-condition	sequential inv.	read-only
pure	`__requires`	`__xrequires`	`__xensures`	`__invariant`	
linear		`__xconsumes`	`__xproduces`	`__smodifies`	`__sreads`

Loop annotations in practice

```
void matmul(float* C, float* A, float* B, int m, int n, int p) {
  __reads("A ⇝ Matrix2(m, p), B ⇝ Matrix2(p, n)");
  __modifies("C ⇝ Matrix2(m, n)");
  for (int i = 0; i < m; i++) {
    __xmodifies("for j in 0..n -> &C[i][j] ⇝ Cell");
    __sreads("A ⇝ Matrix2(m, p), B ⇝ Matrix2(p, n)");
    for (int j = 0; j < n; j++) {
      __xmodifies("&C[i][j] ⇝ Cell");
      __sreads("A ⇝ Matrix2(m, p), B ⇝ Matrix2(p, n)");
      float sum = 0.0f;
      for (int k = 0; k < p; k++) {
        __ghost(matrix2_ro_focus, "A, i, k");
        __ghost(matrix2_ro_focus, "B, k, j");
        sum += A[i][k] * B[k][j];
        __ghost(matrix2_ro_unfocus, "A");
        __ghost(matrix2_ro_unfocus, "B");
      }
      C[i][j] = sum;
    }
  }
}
```

Loop annotations in practice

```
void matmul(float* C, float* A, float* B, int m, int n, int p) {
  __reads("A ~> Matrix2(m, p), B ~> Matrix2(p, n)");
  __modifies("C ~> Matrix2(m, n)");
  for (int i = 0; i < m; i++) {
    __xmodifies("for j in 0..n -> &C[i][j] ~> Cell");

    for (int j = 0; j < n; j++) {
      __xmodifies("&C[i][j] ~> Cell");

      float sum = 0.0f;
      for (int k = 0; k < p; k++) {
        __ghost(matrix2_ro_focus, "A, i, k");
        __ghost(matrix2_ro_focus, "B, k, j");
        sum += A[i][k] * B[k][j];
        __ghost(matrix2_ro_unfocus, "A");
        __ghost(matrix2_ro_unfocus, "B");
      }
      C[i][j] = sum;
    }
  }
}
```

Loop annotations in practice

```
void matmul(float* C, float* A, float* B, int m, int n, int p) {
  __reads("A ~> Matrix2(m, p), B ~> Matrix2(p, n)");
  __modifies("C ~> Matrix2(m, n)");
  for (int i = 0; i < m; i++) {
    __xmodifies("for j in 0..n -> &C[i][j] ~> Cell");

    for (int j = 0; j < n; j++) {
      __xmodifies("&C[i][j] ~> Cell");

      float sum = 0.0f;
      for (int k = 0; k < p; k++) {


        sum += A[i][k] * B[k][j];


      }
      C[i][j] = sum;
    }
  }
}
```

Typing rule for loops (linear clauses only)

$$\left(\bigstar_{i \in 0..n} H(i)\right) \star R \star S(0) \star F$$

```
for(int i = 0; i < n; i += 1) {
    __xconsumes("H(i)");
    __xproduces("Q(i)");
    __sreads("R");
    __smodifies("S(i)");
```

$$[i : int, Pi : i \in 0..n] \star H(i) \star \frac{1}{n}R \star S(i)$$

```
    ...
```

$$Q(i) \star \frac{1}{n}R \star S(i+1)$$

```
}
```

$$\left(\bigstar_{i \in 0..n} Q(i)\right) \star R(i) \star S(n) \star F$$

Typing rule for loops

For

$$\Gamma_p = [\chi.\text{vars}] \star \left(\bigstar_{i \in r} \chi.\text{excl.pre} \right) \star \chi.\text{shrd.reads} \star \text{Subst}\{i := r.\text{first}\}(\chi.\text{shrd.inv})$$

$$\textbf{Some } (\sigma', \Gamma_f) = \Gamma_0 \ominus \text{Specialize}_{\Gamma_0}\{\sigma\}(\Gamma_p)$$

$$\Gamma'_p = [i : \text{int}, i \in r] \star [\chi.\text{vars}] \star \chi.\text{excl.pre} \star \frac{1}{r.\text{len}} \chi.\text{shrd.reads} \star \chi.\text{shrd.inv}$$

$$\{\Gamma'_p\}\ t_b\ \{\Gamma'_q\} \qquad \Gamma'_q \Rightarrow \chi.\text{excl.post} \star \frac{1}{r.\text{len}} \chi.\text{shrd.reads} \star \text{Subst}\{i := r.\text{next}(i)\}(\chi.\text{shrd.inv})$$

$$\Gamma_q = \left(\bigstar_{i \in r} \chi.\text{excl.post} \right) \star \chi.\text{shrd.reads} \star \text{Subst}\{i := r.\text{last}\}(\chi.\text{shrd.inv})$$

$$\frac{\Gamma_r = \text{CloseFracs}(\Gamma_f \star \text{Rename}\{\rho\}(\text{Subst}\{\sigma, \sigma'\}(\Gamma_q))) \qquad (\pi = \text{parallel}) \rightarrow \text{parallelizable}(\chi)}{\{\Gamma_0\}\ \textbf{for}^{\ \pi}(i \in r)_{\chi,\sigma,\rho}\ t_b\ \{\Gamma_r\}}$$

Section 4

Resource usage information

Annotations inserted by the typechecker

Output of the tool using virtual instructions that don't really exist in the AST:

```
void f(int* x, int* y) {
    __consumes("x ~> Cell");
    __modifies("y ~> Cell");

    __ctx_res("Hx: x ~> Cell, Hy: y ~> Cell");
    const int a = *x; // reads Hx and frame Hy
    __ctx_res("Hx: x ~> Cell, Hy: y ~> Cell");
    free(x); // consume permission Hx and frame Hy
    __ctx_res("Hy: y ~> Cell");
    *y += a; // consume Hy and produces Hy'
    __ctx_res("Hy': y ~> Cell");
    // consumes Hy' to instantiate the post-condition
}
```

Annotations inserted by the typechecker

```
void f(int* x, int* y) {
    __consumes("#18: x ⤳ Cell");
    __modifies("#17: y ⤳ Cell");

    ...
    __ctx_res("", "#18: x ⤳ Cell, #17: y ⤳ Cell");
    __framed_res("#17: y ⤳ Cell");
    __contract_inst("", "#15 := #18 : x ⤳ Cell");
    free(x);
    __produced_res("", "");
    __ctx_res("", "#17: y ⤳ Cell");
    ...
}
```

Usage map

$$\{\langle E \mid F \rangle\}\; t^{\Delta}\; \{\langle E' \mid F' \rangle\}$$

One entry in Δ for each resource manipulated by the term t.
Each entry binds the resource name to a usage kind:

Pure usage kind

- required: from E used in t
- ensured: produced by t put in E'

Linear usage kind

- uninit: e.g. with $x \leadsto$ Cell or Uninit($x \leadsto$ Cell) write in x before read
- full: e.g. with $x \leadsto$ Cell, reads in x before write
- splittedFrac: e.g. with αH reads using H or carve a sub-fraction of H
- joinedFrac: e.g. usage of $(\alpha - \beta)H$ when it is merged with βH
- produced: all resources from F' manipulated by t and not splittedFrac and joinedFrac

Usage map computation

$\Delta_1 ; \Delta_2$	$\varnothing$	required	ensured
$\varnothing$	$\varnothing$	required	ensured
required	required	required	$\bot$
ensured	ensured	ensured	$\bot$

$\Delta_1 ; \Delta_2$	$\varnothing$	full	uninit	splittedFrac	joinedFrac	produced
$\varnothing$	$\varnothing$	full	uninit	splittedFrac	joinedFrac	produced
full	full	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
uninit	uninit	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
splittedFrac	splittedFrac	full	full	splittedFrac	splittedFrac	$\bot$
joinedFrac	joinedFrac	full	uninit	splittedFrac	joinedFrac	$\bot$
produced	produced	$\varnothing$	$\varnothing$	produced	produced	$\bot$

Criterion for swap

let $a = ref(3)$;	ensured($a : ptr$), produced($Ha : a \leadsto$ Cell)
let $b = 2 * get(a)$;	ensured($b : int$), required(a), splittedFrac(Ha)
let $c = 3 * get(a)$;	ensured($c : int$), required(a), splittedFrac(Ha)
$set(a, c)$;	required(a), required(c), uninit(Ha), produced(Ha')
let $d = 2 + b$;	ensured($d : int$), required(b)

Two blocks of instructions commute if none of their used resources interfere:

	full / uninit / produced / ensured	splittedFrac / joinedFrac / required	$\varnothing$
full / uninit / produced / ensured	interference	interference	ok
splittedFrac / joinedFrac / required	interference	ok	ok
$\varnothing$	ok	ok	ok

Criterion for swap

let $a = ref(3)$;	ensured($a : ptr$), produced($Ha : a \rightsquigarrow$ Cell)
let $b = 2 * get(a)$;	ensured($b : int$), required(a), splittedFrac(Ha)
let $c = 3 * get(a)$; $set(a, c)$;	ensured($c : int$), required(a), full(Ha), produced(Ha')
let $d = 2 + b$;	ensured($d : int$), required(b)

Two blocks of instructions commute if none of their used resources interfere:

	full / uninit / produced / ensured	splittedFrac / joinedFrac / required	$\varnothing$
full / uninit / produced / ensured	interference	interference	ok
splittedFrac / joinedFrac / required	interference	ok	ok
$\varnothing$	ok	ok	ok

Criterion for swap

let $a = ref(3)$;	ensured($a : ptr$), produced($Ha : a \rightsquigarrow$ Cell)
let $b = 2 * get(a)$;	ensured($b : int$), required(a), splittedFrac(Ha)
let $c = 3 * get(a)$; $set(a, c)$;	ensured($c : int$), required(a), full(Ha), produced(Ha')
let $d = 2 + b$;	ensured($d : int$), required(b)

Two blocks of instructions commute if none of their used resources interfere:

	full / uninit / produced / ensured	splittedFrac / joinedFrac / required	∅
full / uninit / produced / ensured	interference	interference	ok
splittedFrac / joinedFrac / required	interference	ok	ok
∅	ok	ok	ok

Minimize a function contract by example

Original contract

```
void f(int* x, int* y, int* z, int*
    w) {
  __modifies("x ⇝ Cell");
  __modifies("y ⇝ Cell");
  __modifies("z ⇝ Cell");
  __modifies("w ⇝ Cell");
  *x = *y + *z;
  *z += *x;
}
```

Minimized contract

```
void f(int* x, int* y, int* z, int*
    w) {
  __writes("x ⇝ Cell");
  __reads("y ⇝ Cell");
  __modifies("z ⇝ Cell");

  *x = *y + *z;
  *z += *x;
}
```

Useful because better effect analysis ⇒ more optimization opportunities

Minimize a loop contract by example

Original contract

```
for (int i = 0; i < n; ++i) {
  __xmodifies("t[i] ~> Cell");
  __xmodifies("u[i] ~> Cell");
  __smodifies("p ~> Cell");
  t[i] = u[i] + *p;
}
```

Minimized contract

```
for (int i = 0; i < n; ++i) {
  __xwrites("t[i] ~> Cell");
  __xreads("u[i] ~> Cell");
  __sreads("p ~> Cell");
  t[i] = u[i] + *p;
}
```

Minimize

The operation Minimize(Γ, Γ', Δ), is defined in presence of a valid triple $\{\Gamma\}\ t^{\Delta}\ \{\Gamma'\}$.

The output of the operation is a quadruplet $(\hat{E}, \hat{F}, \hat{F}', \bar{F})$ that must satisfy:

- $\{\langle \Gamma.\text{pure}, \hat{E} \mid \hat{F} \rangle\}\ t\ \{\langle \Gamma'.\text{pure}, \hat{E} \mid \hat{F}' \rangle\}$
- $\Gamma \Leftrightarrow \langle \Gamma.\text{pure}, \hat{E} \mid \hat{F} \star \bar{F} \rangle$
- $\Gamma' \Leftrightarrow \langle \Gamma'.\text{pure}, \hat{E} \mid \hat{F}' \star \bar{F} \rangle$
- $\bar{F}$ is intuitively a “maximal” frame removed from both Γ and Γ' and unused by t

Minimize

$\Gamma(y)$	$\Gamma'(y)$	$\Delta(y)$	$\hat{E}$	$\hat{F}$	$\hat{F}'$	$\bar{F}$
H	H	$y \notin \Delta$	$\varnothing$	$\varnothing$	$\varnothing$	$y{:}H$
H	$\varnothing$	full	$\varnothing$	$y{:}H$	$\varnothing$	$\varnothing$
$\text{Uninit}(H)$	$\varnothing$	uninit	$\varnothing$	$y{:}\text{Uninit}(H)$	$\varnothing$	$\varnothing$
H	$\varnothing$	uninit	$\varnothing$	$y{:}\text{Uninit}(H)$	$\varnothing$	$\varnothing$
H	H	splittedFrac	α:frac	$y'{:}\alpha H$	$y'{:}\alpha H$	$y{:}(1-\alpha)H$
αH	αH	splittedFrac	β:frac	$y'{:}\beta H$	$y'{:}\beta H$	$y{:}(\alpha-\beta)H$
$(\alpha-\beta)H$	αH	splittedFrac	γ:frac	$y'{:}\gamma H$	$y'{:}\gamma H, y_\beta{:}\beta H$	$y{:}(\alpha-\beta-\gamma)H$
αH	$(\alpha-\beta)H$	splittedFrac	γ:frac	$y'{:}\gamma H$	$y'{:}(\gamma-\beta)H$	$y{:}(\alpha-\gamma)H$
$(\alpha-\beta_1-\beta_2)H$	$(\alpha-\beta_1-\beta_3)H$	splittedFrac	γ:frac	$y'{:}\gamma H$	$y'{:}(\gamma-\beta_3)H, y_2{:}\beta_2 H$	$y{:}(\alpha-\gamma)H$
$(\alpha-\beta)H$	αH	joinedFrac	$\varnothing$	$\varnothing$	$y'{:}\beta H$	$y{:}(\alpha-\beta)H$
$(\alpha-\beta_1-\beta_2-\beta_3)H$	$(\alpha-\beta_2)H$	joinedFrac	$\varnothing$	$\varnothing$	$y_1{:}\beta_1 H, y_3{:}\beta_3 H$	$y{:}(\alpha-\beta_1-\beta_2-\beta_3)H$

Sub-expression evaluation order

How to manage irrelevant argument evaluation order?
Approximation: Concurrent sub-expressions
Intuitively, distribute a disjoint set of resources to each sub-expression.

Correct example

```
void f(int* i) {
  __modifies("i ⇝ Cell");
  *i = (*i * *i) - *i;
  set(i, (get(i) * get(i)) - get(i))
  // Split i ⇝ Cell in 3 fractions
  // then typecheck set itself
  ...
}
```

Error example

```
void f(int* i) {
  __modifies("i ⇝ Cell");
  *i = get_and_incr(i) -
      get_and_incr(i);
  // Typing error: both get_and_incr
       consume full i ⇝ Cell
  ...
}
```

Declarative sub-expression typing rule

SubexprDeclarative

$$\dfrac{\begin{array}{c} x_i \text{ fresh} \qquad \Gamma_0 \Rightarrow \left(\bigstar_{i\in[0,n]} \hat{\Gamma}_i\right) \star \hat{\Gamma}_r \\ \forall i \in [0,n].\quad \{\hat{\Gamma}_i\}\ t_i^{\Delta_i}\ \{\hat{\Gamma}''_i\} \quad \wedge \quad \hat{\Gamma}'_i = \langle \hat{\Gamma}''_i.\text{pure} \Vdash \Delta_i.\text{ensured} \mid \hat{\Gamma}''_i.\text{linear}\rangle \\ \Gamma_c = \text{CloseFracs}\left(\hat{\Gamma}_r \star \bigstar_{i\in[0,n]} \text{Rename}\{\mathbf{res} := x_i\}(\hat{\Gamma}'_i)\right) \qquad \{\Gamma_c\}\ E[x_0, \dots, x_n]\ \{\Gamma_p\} \end{array}}{\{\Gamma_0\}\ E[t_0, \dots, t_n]\ \{\Gamma_p\}}$$

Declarative sub-expression typing rule

SubexprDeclarative

$$\dfrac{x_i \text{ fresh} \qquad \Gamma_0 \Rightarrow \left(\bigstar_{i\in[0,n]} \hat{\Gamma}_i\right) \star \hat{\Gamma}_r \\ \forall i \in [0,n].\quad \{\hat{\Gamma}_i\}\ t_i^{\Delta_i}\ \{\hat{\Gamma}''_i\} \quad \wedge \quad \hat{\Gamma}'_i = \langle \hat{\Gamma}''_i.\text{pure} \Vdash \Delta_i.\text{ensured} \mid \hat{\Gamma}''_i.\text{linear}\rangle \\ \Gamma_c = \text{CloseFracs}\left(\hat{\Gamma}_r \star \bigstar_{i\in[0,n]} \text{Rename}\{\textbf{res} := x_i\}(\hat{\Gamma}'_i)\right) \qquad \{\Gamma_c\}\ E[x_0, \ldots, x_n]\ \{\Gamma_p\}}{\{\Gamma_0\}\ E[t_0, \ldots, t_n]\ \{\Gamma_p\}}$$

Algorithmic sub-expression typing rule

SubexprAlgorithmic

$$\frac{x_i \text{ fresh} \qquad \forall i \in [0,n].\quad \{\Gamma_i\}\ t_i^{\Delta_i}\ \{\Gamma'_i\} \quad\wedge\quad (\hat{E}_i, \hat{F}_i, \hat{F}'_i, \bar{F}_i) = \text{Minimize}(\Gamma_i, \Gamma'_i, \Delta_i) \\ \forall i \in [0,n].\quad \Gamma_{i+1} = \langle\, \Gamma_i.\text{pure},\ \hat{E}_i \mid \bar{F}_i \,\rangle \quad\wedge\quad \hat{\Gamma}'_i = \langle\, \Gamma'_i.\text{pure} \Vdash \Delta_i.\text{ensured} \mid \hat{F}'_i \,\rangle \\ \Gamma_p = \text{CloseFracs}\left(\Gamma_{n+1} \star \mathop{\bigstar}_{i\in[0,n]} \text{Rename}\{\textbf{res} := x_i\}(\hat{\Gamma}'_i)\right) \qquad \{\Gamma_p\}\ E[x_0, \ldots, x_n]\ \{\Gamma_q\}}{\{\Gamma_0\}\ E[t_0, \ldots, t_n]\ \{\Gamma_q\}}$$

Demo on OptiTrust

We use this typechecker in OptiTrust: an interactive source-to-source optimization framework

Demo

Conclusion

We have build a practical Separation Logic typechecker with:

- Standard handling of linear and fractionnal permissions
- Automatic fraction splits and joins
- Extension with Uninit permissions
- Parallel loop contracts
- Usage summary
- Contract minimization
- Non-deterministic sub-expression evaluation order

Future work

- Improve expressivity: add models for cells and linear predicates
- More automation on folding/unfolding predicates
- Check and possibly improve performance of the typing algorithm

Resource entailment

Subtraction

$$\Gamma_0 \ominus \Gamma_r = \begin{cases} \textbf{Some } (\sigma, \Gamma_f) \\ \textbf{None} \end{cases}$$

- σ: map from each pure or linear resource in Γ_r to a matching resource from Γ_0
- Γ_f: resources from Γ_0 that remain after generating σ

Full entailment

$$\Gamma_0 \Rightarrow \Gamma_r \quad \approx \quad \textbf{Some } (\sigma, \Gamma_f) = \Gamma_0 \ominus \Gamma_r \quad \wedge \quad \Gamma_f = \varnothing$$

Do not try to split fraction when instantiating read-only resources