Time credits and time receipts in Iris

Glen Mével, Jacques-Henri Jourdan, François Pottier

Inria
CNRS, LRI, Univ. Paris Sud, Université Paris-Saclay

April 8, 2019
Prague

This talk

- recent works: **time credits**
  aim: prove an upper bound on the running time of a program
- this talk: **time receipts**
  aim: **assume** an upper bound on the running time of a program

These are dual notions.

This talk

- recent works: **time credits**
  aim: prove an upper bound on the running time of a program
- this talk: **time receipts**
  aim: **assume** an upper bound on the running time of a program

These are dual notions.

Example: a unique symbol generator

The function *genSym* returns fresh symbols:

$$\texttt{let}\ \mathit{lastSym} = \texttt{ref}\ 0$$

$$\begin{array}{l} \texttt{let}\ \mathit{genSym}() = \\ \quad \mathit{lastSym} := !\,\mathit{lastSym} + 1; \\ \quad !\,\mathit{lastSym} \end{array}$$

Example: a unique symbol generator

The function *genSym* returns fresh symbols:

$\texttt{let}\ \mathit{lastSym} = \texttt{ref}\ 0 \quad$ (* unsigned 64-bit integer *)

$\texttt{let}\ \mathit{genSym}() =$
$\quad \mathit{lastSym} := !\,\mathit{lastSym} + 1; \quad$ (* may overflow! *)
$\quad !\,\mathit{lastSym}$

Strictly speaking, this code is **not** correct.

Example: a unique symbol generator

The function *genSym* returns fresh symbols:

$$\texttt{let}\ \mathit{lastSym} = \texttt{ref}\ 0 \quad \text{(* unsigned 64-bit integer *)}$$

$$\begin{array}{l} \texttt{let}\ \mathit{genSym}() = \\ \quad \mathit{lastSym} := !\,\mathit{lastSym} + 1; \quad \text{(* may overflow! *)} \\ \quad !\,\mathit{lastSym} \end{array}$$

Strictly speaking, this code is **not** correct.

We still want to prove that this code is “correct” in some sense.

The Bounded Time Hypothesis [Clochard **et al.**, 2015]

Counting from 0 to 2^{64} takes **centuries** with a modern processor.

Therefore, this overflow won't happen in a lifetime.

How to express this informal argument in separation logic?

The Bounded Time Hypothesis [Clochard **et al.**, 2015]

Counting from 0 to 2^{64} takes **centuries** with a modern processor.

Therefore, this overflow won't happen in a lifetime.

How to express this informal argument in separation logic?

In this talk:

- We answer this question using **time receipts**.
- We prove that Iris, extended with time receipts, is **sound**.

A closer look at the problem

Specification of genSym

A specification (in separation logic):

$$P\ \emptyset \ast \forall S.\ \begin{pmatrix} \{P\ S\} \\ \quad genSym() \\ \{\lambda n.\ n \notin S \ast P(S \cup \{n\})\} \end{pmatrix}$$

for some proposition $P\ S$ which represents:

- the ownership of the generator;
- the fact that S is the set of all symbols returned so far.

Tentative proof of genSym

$\texttt{let}\ \mathit{lastSym} = \texttt{ref}\ 0$

$\texttt{let}\ \mathit{genSym}() =$

$\quad \mathit{lastSym} := !\,\mathit{lastSym} + 1;$

$\quad !\,\mathit{lastSym}$

Tentative proof of genSym

$\{\}$
`let` *lastSym* $=$ `ref` 0

$\{P\ \emptyset\}$

$\{P\ S\}$
`let` *genSym*() $=$

$lastSym := !\, lastSym + 1;$

$!\, lastSym$

$\{\lambda n.\ n \notin S\ *\ P(S \cup \{n\})\}$

Tentative proof of genSym

Invariant: $P\ S \triangleq \mathit{lastSym} \mapsto \max S$

$\{\}$

`let` $\mathit{lastSym} =$ `ref` 0

$\{P\ \emptyset\}$

$\{P\ S\}$

`let` $\mathit{genSym}() =$

$\quad \mathit{lastSym} := !\,\mathit{lastSym} + 1;$

$\quad !\,\mathit{lastSym}$

$\{\lambda n.\ n \notin S\ *\ P(S \cup \{n\})\}$

Tentative proof of genSym

Invariant: $P\ S \triangleq lastSym \mapsto \max S$

$\{\}$
$\texttt{let}\ lastSym = \texttt{ref}\ 0$
$\{lastSym \mapsto 0\}$
$\{P\ \emptyset\}$

$\{P\ S\}$
$\texttt{let}\ genSym() =$
$\quad \{lastSym \mapsto \max S\}$
$\quad lastSym := !\,lastSym + 1;$
$\quad \{lastSym \mapsto \lfloor \max S + 1 \rfloor_{2^{64}}\}$
$\quad \{\lfloor \max S + 1 \rfloor_{2^{64}} \notin S \ast lastSym \mapsto \lfloor \max S + 1 \rfloor_{2^{64}}\}$
$\quad !\,lastSym$
$\quad \{\lambda n.\ n \notin S \ast lastSym \mapsto n\}$
$\{\lambda n.\ n \notin S \ast P(S \cup \{n\})\}$

Tentative proof of genSym

Invariant: $P\ S \triangleq lastSym \mapsto \max S$

$\{\}$
$\texttt{let}\ lastSym = \texttt{ref}\ 0$
$\{lastSym \mapsto 0\}$
$\{P\ \emptyset\}$

$\{P\ S\}$
$\texttt{let}\ genSym() =$
$\quad \{lastSym \mapsto \max S\}$
$\quad lastSym := !\, lastSym + 1;$
$\quad \{lastSym \mapsto \lfloor \max S + 1 \rfloor_{2^{64}}\}$
Wrong $\quad \{\cancel{\lfloor \max S + 1 \rfloor_{2^{64}} \notin S} \ast lastSym \mapsto \lfloor \max S + 1 \rfloor_{2^{64}}\}$
$\quad !\, lastSym$
$\quad \{\lambda n.\ n \notin S \ast lastSym \mapsto n\}$
$\{\lambda n.\ n \notin S \ast P(S \cup \{n\})\}$

An unpleasant workaround: patch the specification

We may add a precondition to exclude any chance of overflow:

$$P\ \emptyset\ *\ \forall S.\ \begin{pmatrix} \{P\ S\ *\ |S| < 2^{64} - 1\} \\ \quad genSym() \\ \{\lambda n.\ n \notin S\ *\ P(S \cup \{n\})\} \end{pmatrix}$$

This pollutes user proofs with cumbersome proof obligations. . .
which may even be unprovable!

Time receipts in action

Time receipts in separation logic

To count execution steps, we introduce **time receipts**.

Each step produces **one** time receipt, **and only one**:

$$\{\mathrm{True}\} \quad x + y \quad \{\lambda z.\ z = \lfloor x + y \rfloor_{2^{64}} * ⧗ 1\}_{⧗}$$

Time receipts sum up:

$$\underbrace{⧗ 1 * \ldots * ⧗ 1}_{n} \equiv ⧗ n$$

But time receipts do **not** duplicate (separation logic):

$$⧗ 1 \not\twoheadrightarrow ⧗ 1 * ⧗ 1$$

Therefore, $⧗ n$ is a witness that (at least) n steps have been taken.

Proof of genSym using time receipts

Invariant: $P\ S \triangleq \mathit{lastSym} \mapsto \max S$

$\{\}$
$\mathtt{let}\ \mathit{lastSym} = \mathtt{ref}\ 0$
$\{\mathit{lastSym} \mapsto 0\}$
$\{P\ \emptyset\}$

$\{P\ S\}$
$\mathtt{let}\ \mathit{genSym}() =$
$\quad \{\mathit{lastSym} \mapsto \max S\}$
$\quad \mathit{lastSym} := !\,\mathit{lastSym} + 1;$
$\quad \{\mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}}\}$
$\quad \{\lfloor \max S + 1 \rfloor_{2^{64}} \notin S \ast \mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}}\}$
$\quad !\,\mathit{lastSym}$
$\quad \{\lambda n.\ n \notin S \ast \mathit{lastSym} \mapsto n\}$
$\{\lambda n.\ n \notin S \ast P(S \cup \{n\})\}$

Proof of genSym using time receipts

Invariant: $P\ S \triangleq \mathit{lastSym} \mapsto \max S \ * \ ⧗(\max S)$

We keep track of elapsed time.

$\{\}$
`let` $\mathit{lastSym} = $ `ref` 0
$\{\mathit{lastSym} \mapsto 0\}$
$\{P\ \emptyset\}$

$\{P\ S\}$
`let` $\mathit{genSym}() =$
$\quad\{\mathit{lastSym} \mapsto \max S\}$
$\quad\mathit{lastSym} := !\,\mathit{lastSym} + 1;$
$\quad\{\mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}}\}$
$\quad\{\lfloor \max S + 1 \rfloor_{2^{64}} \notin S \ * \ \mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}}\}$
$\quad!\,\mathit{lastSym}$
$\quad\{\lambda n.\ n \notin S \ * \ \mathit{lastSym} \mapsto n\}$
$\{\lambda n.\ n \notin S \ * \ P(S \cup \{n\})\}$

Proof of genSym using time receipts

Invariant: $P\ S \triangleq \mathit{lastSym} \mapsto \max S \ * \ \text{⧗}(\max S)$

$\{\}$
$\texttt{let}\ \mathit{lastSym} = \texttt{ref}\ 0$
$\{\mathit{lastSym} \mapsto 0 \ * \ \text{⧗}0\}$
$\{P\ \emptyset\}$

$\{P\ S\}$
$\texttt{let}\ \mathit{genSym}() =$
$\quad \{\mathit{lastSym} \mapsto \max S \ * \ \text{⧗}\max S\}$
$\quad \mathit{lastSym} := !\,\mathit{lastSym} + 1;$
$\quad \{\mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}} \ * \ \text{⧗}(\max S + 1)\}$
$\quad \{\lfloor \max S + 1 \rfloor_{2^{64}} \notin S \ * \ \mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}} \ * \ \text{⧗}(\max S + 1)$
$\quad !\,\mathit{lastSym}$
$\quad \{\lambda n.\ n \notin S \ * \ \mathit{lastSym} \mapsto n \ * \ \text{⧗}n\}$
$\{\lambda n.\ n \notin S \ * \ P(S \cup \{n\})\}$

Proof of genSym using time receipts

Invariant: $P\ S \triangleq \mathit{lastSym} \mapsto \max S \ * \ ⧗(\max S)$

$\{\}$
let $\mathit{lastSym}$ = ref 0
$\{\mathit{lastSym} \mapsto 0 \ * \ ⧗0\}$
$\{P\ \emptyset\}$

Initialization

We obtain 0 time receipts for free.

$\{P\ S\}$
let $\mathit{genSym}() =$
$\{\mathit{lastSym} \mapsto \max S \ * \ ⧗\max S\}$
$\mathit{lastSym} := !\,\mathit{lastSym} + 1;$
$\{\mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}} \ * \ ⧗(\max S + 1)\}$
$\{\lfloor \max S + 1 \rfloor_{2^{64}} \notin S \ * \ \mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}} \ * \ ⧗(\max S + 1)\}$
$!\,\mathit{lastSym}$
$\{\lambda n.\ n \notin S \ * \ \mathit{lastSym} \mapsto n \ * \ ⧗n\}$
$\{\lambda n.\ n \notin S \ * \ P(S \cup \{n\})\}$

Proof of genSym using time receipts

Invariant: $P\ S \triangleq \mathit{lastSym} \mapsto \max S \ * \ ⧗(\max S)$

$\{\}$
$\texttt{let}\ \mathit{lastSym} = \texttt{ref}\ 0$
$\{\mathit{lastSym} \mapsto 0 \ * \ ⧗0\}$
$\{P\ \emptyset\}$

$\{P\ S\}$
$\texttt{let}\ \mathit{genSym}() =$
$\quad \{\mathit{lastSym} \mapsto \max S \ * \ ⧗\max S\}$
$\quad \mathit{lastSym} := !\,\mathit{lastSym} + 1;$
$\quad \{\mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}} \ * \ ⧗(\max S + 1)\}$
$\quad \{\lfloor \max S + 1 \rfloor_{2^{64}} \notin S \ * \ \mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}} \ * \ ⧗(\max S + 1)$
$\quad !\,\mathit{lastSym}$
$\quad \{\lambda n.\ n \notin S \ * \ \mathit{lastSym} \mapsto n \ * \ ⧗n\}$
$\{\lambda n.\ n \notin S \ * \ P(S \cup \{n\})\}$

Tick

Addition produces one time receipt.

The Bounded Time Hypothesis with time receipts

Let N be an arbitrary integer.

We posit the **Bounded Time Hypothesis**:

$$\text{⧗}\, N \vdash \mathsf{False}$$

In other words, we assume that no execution lasts for N steps.

The larger N, the weaker this assumption.

Consequence:

$$\text{⧗}\, n \vdash n < N$$

Proof of genSym using time receipts and the BTH

Invariant: $P\ S \triangleq \mathit{lastSym} \mapsto \max S \ast ⧗(\max S)$

$\{\}$
`let` $\mathit{lastSym} =$ `ref` 0
$\{\mathit{lastSym} \mapsto 0 \ast ⧗0\}$
$\{P\ \emptyset\}$

$\{P\ S\}$
`let` $\mathit{genSym}() =$
$\quad\{\mathit{lastSym} \mapsto \max S \ast ⧗\max S\}$
$\quad\mathit{lastSym} := !\,\mathit{lastSym} + 1;$
$\quad\{\mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}} \ast ⧗(\max S + 1)\}$
$\quad\{\lfloor \max S + 1 \rfloor_{2^{64}} \notin S \ast \mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}} \ast ⧗(\max S + 1)\}$
$\quad!\,\mathit{lastSym}$
$\quad\{\lambda n.\ n \notin S \ast \mathit{lastSym} \mapsto n \ast ⧗n\}$
$\{\lambda n.\ n \notin S \ast P(S \cup \{n\})\}$

Proof of genSym using time receipts and the BTH

Invariant: $P\ S \triangleq \mathit{lastSym} \mapsto \max S \ast \text{⧗}(\max S)$

$\{\}$
$\texttt{let}\ \mathit{lastSym} = \texttt{ref}\ 0$
$\{\mathit{lastSym} \mapsto 0 \ast \text{⧗}0\}$
$\{P\ \emptyset\}$

Bounded Time

$\text{⧗}(\max S + 1)$ entails $\max S + 1 < N$.

$\{P\ S\}$
$\texttt{let}\ \mathit{genSym}() =$
$\{\mathit{lastSym} \mapsto \max S \ast \text{⧗}\max S\}$
$\mathit{lastSym} := !\,\mathit{lastSym} + 1;$
$\{\mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}} \ast \text{⧗}(\max S + 1)\}$
$\{\lfloor \max S + 1 \rfloor_{2^{64}} \notin S \ast \mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}} \ast \text{⧗}(\max S + 1)\}$
$!\,\mathit{lastSym}$
$\{\lambda n.\ n \notin S \ast \mathit{lastSym} \mapsto n \ast \text{⧗}n\}$
$\{\lambda n.\ n \notin S \ast P(S \cup \{n\})\}$

Proof of genSym using time receipts and the BTH

Invariant: $P\ S \triangleq \mathit{lastSym} \mapsto \max S \ast \text{⧗}(\max S)$

$\{\}$
$\texttt{let}\ \mathit{lastSym} = \texttt{ref}\ 0$
$\{\mathit{lastSym} \mapsto 0 \ast \text{⧗}\,0\}$
$\{P\ \emptyset\}$

Bounded Time

We further require $N \leq 2^{64}$.

$\{P\ S\}$
$\texttt{let}\ \mathit{genSym}() =$
$\{\mathit{lastSym} \mapsto \max S \ast \text{⧗}\max S\}$
$\mathit{lastSym} := !\,\mathit{lastSym} + 1;$
$\{\mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}} \ast \text{⧗}(\max S + 1)\}$
$\{\lfloor \max S + 1 \rfloor_{2^{64}} \notin S \ast \mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}} \ast \text{⧗}(\max S + 1)$
$!\,\mathit{lastSym}$
$\{\lambda n.\ n \notin S \ast \mathit{lastSym} \mapsto n \ast \text{⧗}\,n\}$
$\{\lambda n.\ n \notin S \ast P(S \cup \{n\})\}$

Proof of genSym using time receipts and the BTH

Invariant: $P\ S \triangleq \mathit{lastSym} \mapsto \max S \ast ⧗(\max S)$

$\{\}$
$\texttt{let}\ \mathit{lastSym} = \texttt{ref}\ 0$
$\{\mathit{lastSym} \mapsto 0 \ast ⧗0\}$
$\{P\ \emptyset\}$

No overflow

Then, $\max S + 1 < 2^{64}$.

$\{P\ S\}$
$\texttt{let}\ \mathit{genSym}() =$
$\{\mathit{lastSym} \mapsto \max S \ast ⧗\max S\}$
$\mathit{lastSym} := !\,\mathit{lastSym} + 1;$
$\{\mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}} \ast ⧗(\max S + 1)\}$
$\{\max S + 1 \quad \notin S \ast \mathit{lastSym} \mapsto \max S + 1 \quad \ast ⧗(\max S + 1)\}$
$!\,\mathit{lastSym}$
$\{\lambda n.\ n \notin S \ast \mathit{lastSym} \mapsto n \ast ⧗n\}$
$\{\lambda n.\ n \notin S \ast P(S \cup \{n\})\}$

Proof of genSym using time receipts and the BTH

Invariant: $P\ S \triangleq \mathit{lastSym} \mapsto \max S \ast ⧗(\max S)$

$\{\}$
`let` $\mathit{lastSym} =$ `ref` 0
$\{\mathit{lastSym} \mapsto 0 \ast ⧗ 0\}$
$\{P\ \emptyset\}$

$\{P\ S\}$
`let` $\mathit{genSym}() =$
$\{\mathit{lastSym} \mapsto \max S \ast ⧗ \max S\}$
$\mathit{lastSym} := !\,\mathit{lastSym} + 1;$
$\{\mathit{lastSym} \mapsto \lfloor \max S + 1 \rfloor_{2^{64}} \ast ⧗(\max S + 1)\}$
✓ $\{\max S + 1 \quad \notin S \ast \mathit{lastSym} \mapsto \max S + 1 \quad \ast ⧗(\max S + 1)$
$!\,\mathit{lastSym}$
$\{\lambda n.\ n \notin S \ast \mathit{lastSym} \mapsto n \ast ⧗ n\}$
$\{\lambda n.\ n \notin S \ast P(S \cup \{n\})\}$

Iris$^{⧗}$, a program logic with time receipts

Time receipts satisfy the **Bounded Time Hypothesis**:

$$⧗ N \vdash \text{False}$$

Each step produces one time receipt; for instance:

$$\{\text{True}\} \quad x + y \quad \{\lambda z.\ z = \lfloor x + y \rfloor_{2^{64}} \ast ⧗ 1\}_{⧗}$$

Iris$^{\text{⧗}}$, a program logic with time receipts

Time receipts satisfy the **Bounded Time Hypothesis**:

$$⧗ N \vdash \text{False}$$

Each step produces one time receipt; for instance:

$$\{\text{True}\} \quad x + y \quad \{\lambda z.\ z = \lfloor x + y \rfloor_{2^{64}} \ast ⧗ 1\}_{⧗}$$

We can obtain zero time receipts unconditionally:

$$\vdash \ ⧗ 0$$

Time receipts are additive:

$$⧗ m \ast ⧗ n \ \equiv \ ⧗(m + n)$$

Soundness of Iris with time receipts

Soundness of Iris$^{⧗}$

We want our program logic Iris$^{⧗}$ to satisfy this property:

Theorem (Soundness of Iris$^{⧗}$)

If the following Iris$^{⧗}$ triple holds:

$$\{\mathrm{True}\}\ e\ \{\_\}_{⧗}$$

then e cannot crash ***until N steps have been taken.***

We say that "e is $(N-1)$-safe".

Crashing means trying to step while in a stuck configuration; for example, dereferencing a non-pointer.

Proof sketch of the soundness theorem

We use Iris as a model of Iris$^{⧗}$.

$$\{P\}\, e\, \{\varphi\}_{⧗} \triangleq \{P\}\, \langle\!\langle e \rangle\!\rangle\, \{\varphi\}$$

The transformation $\langle\!\langle\cdot\rangle\!\rangle$ inserts *ticks* (see next slides).

The proof then works as follows:

$$\begin{array}{l} \{\mathrm{True}\}\, \langle\!\langle e \rangle\!\rangle\, \{\_\} \\ \quad \Big\Downarrow \; \textit{Soundness theorem of Iris [Jung } \textbf{\textit{et al.}}\textit{, 2015]} \\ \langle\!\langle e \rangle\!\rangle \text{ is safe} \\ \quad \Big\Downarrow \; \textit{Simulation lemma} \\ e \text{ is } (N-1)\text{-safe} \end{array}$$

The program transformation

We keep track of the number of steps using a global counter c, initialized with 0.

The transformation inserts one *tick* instruction per operation.

$$\langle\!\langle e_1 + e_2 \rangle\!\rangle \triangleq \mathit{tick}\ (\langle\!\langle e_1 \rangle\!\rangle + \langle\!\langle e_2 \rangle\!\rangle)$$

tick increments c. On its N^{th} execution, it does not return.

$$\begin{array}{l} \texttt{let}\ \mathit{tick}\ x = \\ \quad !\,c := !\,c + 1; \\ \quad \texttt{if}\ !\,c < N\ \texttt{then}\ x\ \texttt{else}\ \mathit{loop}\,() \end{array}$$

Idea: transform a program that runs for too long
into a program that never ends, hence is safe.

The simulation lemma

This program transformation does satisfy the desired lemma:

Lemma (Simulation)

If $\langle\langle e \rangle\rangle$ is safe (i.e. it cannot crash),
then e is $(N-1)$-safe (i.e. it cannot crash until N steps have been taken).

The model of time receipts

The transformation maintains the invariant $!\,c < N$.

$\mathbin{⧗} 1$ is modeled as an exclusive portion of the value of the counter c (Iris features used: authoritative monoidal resource, invariant).

In particular, $⧗\, n \vdash\, !\,c \geq n$. Hence, $⧗\, N \vdash \text{False}$.

All other axioms of time receipts are realised as well.

Conclusion

Conclusion

Contributions (new):

	Soundness	*Application*
Time credits	✓	Reconstruction of Okasaki and Danielsson's thunks (amortized analysis)
Time receipts (exclusive / persistent)	✓	Reconstruction of Clochard **et al.**'s overflow-free integers
Time credits and time receipts	✓	Proof of Union-Find: complexity, absence of overflow in ranks

Defined within Iris, machine-checked with Coq

Open question: Can we prove useful facts about **concurrent** code?

Thank you for your time.

⌛17

What about concurrency?

Iris is a concurrent separation logic; thus, our program logics already support concurrency: they measure the **work** (total number of operations in all threads).

$$\begin{array}{l} \texttt{let}\ \mathit{tick}\ x = \\ \quad \texttt{if}\,(\texttt{FAA}\ c\ 1 < N - 1)\,\texttt{then}\,x\,\texttt{else}\,\mathit{loop}\,() \end{array}$$

What about measuring the **span** (running time of the longest-living thread)?
A path to explore: a separate notion of time receipt for each thread, with a rule to clone time receipts of the calling thread when forking.

Compiling code analysed with time receipts

For time receipt proofs to be valid, we need to forbid optimizations!
Otherwise, programs may compute faster than expected.
For example:

```
for i from 1 to N do
  ()
done;
(* This point is beyond the scope of Iris⧗:
 * anything below may be unsafe,
 * but it shouldn't be reached in a lifetime... *)
crash ()
```

A compiler may optimize it to:

```
(* Too bad! *)
crash ()
```

A solution: insert actual *tick* operations and make them opaque.

Example application: Union-Find

We implement the Union-Find with ranks stored in machine words. While proving the correctness of the algorithm, we also prove

- its complexity (using time credits)
- and the absence of overflows for ranks (using time receipts).

Granted that $x, y \in D$ and $\log_2 \log_2 N < \mathit{word\_size} - 1$, we show the Iris$^{\$⧗}$ triple:

$$\begin{array}{l}\{\text{isUF}\ D\ R\ V\ *\ \$(44\alpha(|D|) + 152)\} \\ \quad \mathit{union}\ x\ y \\ \{\lambda z.\ \text{isUF}\ D\ R'\ V'\ *\ (z = R\ x \lor z = R\ y)\}_{\$⧗}\end{array}$$

Consequences:

- the (amortized) complexity is the inverse Ackermann function;
- if $N = 2^{64}$, then $\mathit{word\_size} \geq 8$ is enough to avoid overflows.

Example: a unique symbol generator (functional version)

Code:

$$
\begin{array}{l}
\texttt{let}\ \mathit{makeGenSym}() = \\
\quad \texttt{let}\ \mathit{lastSym} = \texttt{ref}\ 0\ \texttt{in} \qquad \text{(* unsigned 64-bit integer *)} \\
\quad \texttt{fun}\ () \rightarrow \\
\qquad \mathit{lastSym} := !\,\mathit{lastSym} + 1; \qquad \text{(* may overflow *)} \\
\qquad !\,\mathit{lastSym}
\end{array}
$$

Specification (in higher-order separation logic):

$$
\begin{array}{l}
\{\mathrm{True}\} \\
\quad \mathit{makeGenSym}() \\
\left\{
\begin{array}{l}
\lambda\ \mathit{genSym}.\ \exists P. \\
\quad P\ \emptyset\ *\ \forall S.\ \left(
\begin{array}{l}
\{P\ S\} \\
\quad \mathit{genSym}() \\
\{\lambda n.\ n \notin S\ *\ P(S \cup \{n\})\}
\end{array}
\right)
\end{array}
\right\}
\end{array}
$$

Alternative specification of makeGenSym

Specification (in Iris):

$$
\begin{array}{l}
\{\mathsf{True}\} \\
\quad \mathit{makeGenSym}() \\
\left\{ \begin{array}{l}
\lambda\ \mathit{genSym}.\ \exists\gamma. \\
\qquad \forall n.\ \left(\begin{array}{l} \{\mathsf{True}\} \\ \quad \mathit{genSym}() \\ \{\lambda m.\ \mathsf{OwnSym}_\gamma(m)\} \end{array} \right)
\end{array} \right\}
\end{array}
$$

- The ownership of the generator is shared through an invariant.
- $\mathsf{OwnSym}_\gamma(m)$ asserts uniqueness of symbol m:

$$\mathsf{OwnSym}_\gamma(m_1) \ast \mathsf{OwnSym}_\gamma(m_2) \ -\!\!\ast\ \ m_1 \neq m_2$$

A program logic with time credits

Each step consumes one time credit; for instance:

$$\{\$1\} \quad x + y \quad \{\lambda z.\ z = \lfloor x + y \rfloor_{2^{64}}\}_{⧗}$$

We can obtain zero time credits unconditionally:

$$\vdash \ \$0$$

Time credits are additive:

$$\$m \ast \$n \ \equiv \ \$(m + n)$$

A program logic with time credits — Adequacy

Our program logic Iris$^{\$}$ satisfies this property:

Theorem (Adequacy of Iris$^{\$}$)

If the following Iris$^{\text{⧗}}$ triple holds:

$$\{\$n\}\ e\ \{\varphi\}_{\$}$$

then:

- *e cannot crash;*
- *if e computes a value v, then $\varphi\ v$ holds;*
- ***e computes for at most n steps.***

Adequacy of Iris$^{⧗}$

Our program logic Iris$^{⧗}$ satisfies this property:

Theorem (Adequacy of Iris$^{⧗}$)

If the following Iris$^{⧗}$ triple holds:

$$\{\text{True}\}\ e\ \{\varphi\}_{⧗}$$

then:

- *e cannot crash* ***until N steps have been taken****;*
- *if e computes a value v* ***in less than N steps****, then φ v holds.*

A program logic with duplicable time receipts

Duplicable time receipts satisfy the **Bounded Time Hypothesis**:

$$\text{⧖} N \vdash \text{False}$$

Each step increments a duplicable time receipt; for instance:

$$\{\text{⧖} m\} \quad x + y \quad \{\lambda z.\ z = \lfloor x + y \rfloor_{2^{64}} \ast \text{⧖}(m+1)\}_{\text{⧗}}$$

We can obtain zero duplicable time receipts unconditionally:

$$\vdash \text{⧖} 0$$

Duplicable time receipts obey maximum:

$$\text{⧖} m \ast \text{⧖} n \equiv \text{⧖} \max(m, n)$$

Duplicable time receipts **are** duplicable:

$$\text{⧖} m \mathrel{-\!\!\ast} \text{⧖} m \ast \text{⧖} m$$

Relation between time receipts and duplicable time receipts:

$$\text{⧗} m \vdash \text{⧗} m \ast \text{⧖} m$$

Overflow-free integers (summable)

$$\mathrm{IsClock}(v, n) \triangleq 0 \leq n < 2^{64} \ * \ v = n \ * \ \boldsymbol{\Sigma}\, n$$

- non-duplicable
- supports addition (consumes its operands):

$$\{\mathrm{IsClock}(v_1, n_1) \ * \ \mathrm{IsClock}(v_2, n_2)\}$$
$$v_1 + v_2$$
$$\{\lambda w.\ \mathrm{IsClock}(w, n_1 + n_2)\}$$

no overflow!

Overflow-free integers (incrementable)

$$\text{IsSnapClock}(v, n) \triangleq 0 \leq n < 2^{64} * v = n * ⧖ n$$

- duplicable
- supports incrementation (does **not** consume its operand):

$$\{\text{IsSnapClock}(v, n)\}$$
$$v + 1$$
$$\{\lambda w.\ \text{IsSnapClock}(w, n + 1)\}$$

no overflow!

Hoare logic primer

- `prgm` is a program (source code).
- *Pre* and *Post* are logical formulas.

$$\{Pre\}\, \texttt{prgm}\, \{Post\}$$

Soundness:
"If Pre holds, then prgm won't crash."

(Partial) correctness:
"If Pre holds, then after prgm is run, Post will hold."

Total correctness:
"If Pre holds, then prgm terminates and, after prgm is run, Post will hold."

Separation logic primer

P is a resource.
$x \mapsto v$ is an exclusive resource, its ownership cannot be shared.

- Standard logic: $P \Rightarrow P \wedge P$
- Separation logic: $P \not\twoheadrightarrow P * P$ (resources are not duplicable)

$P * Q$ are disjoint resources.
$x \mapsto v * x \mapsto v'$ is absurd.

Affine sep. logic: $P * Q \twoheadrightarrow P$ (resources can be thrown away)

Iris primer

Iris is:

- an affine separation logic,
- higher-order,
- full-featured (impredicative invariants, monoidal resources. . .),
- very extensible,
- formalized in Coq.