Reducing copies in a Lustre-like language with arrays

Grégoire Bussone

PARKAS

November 21, 2024

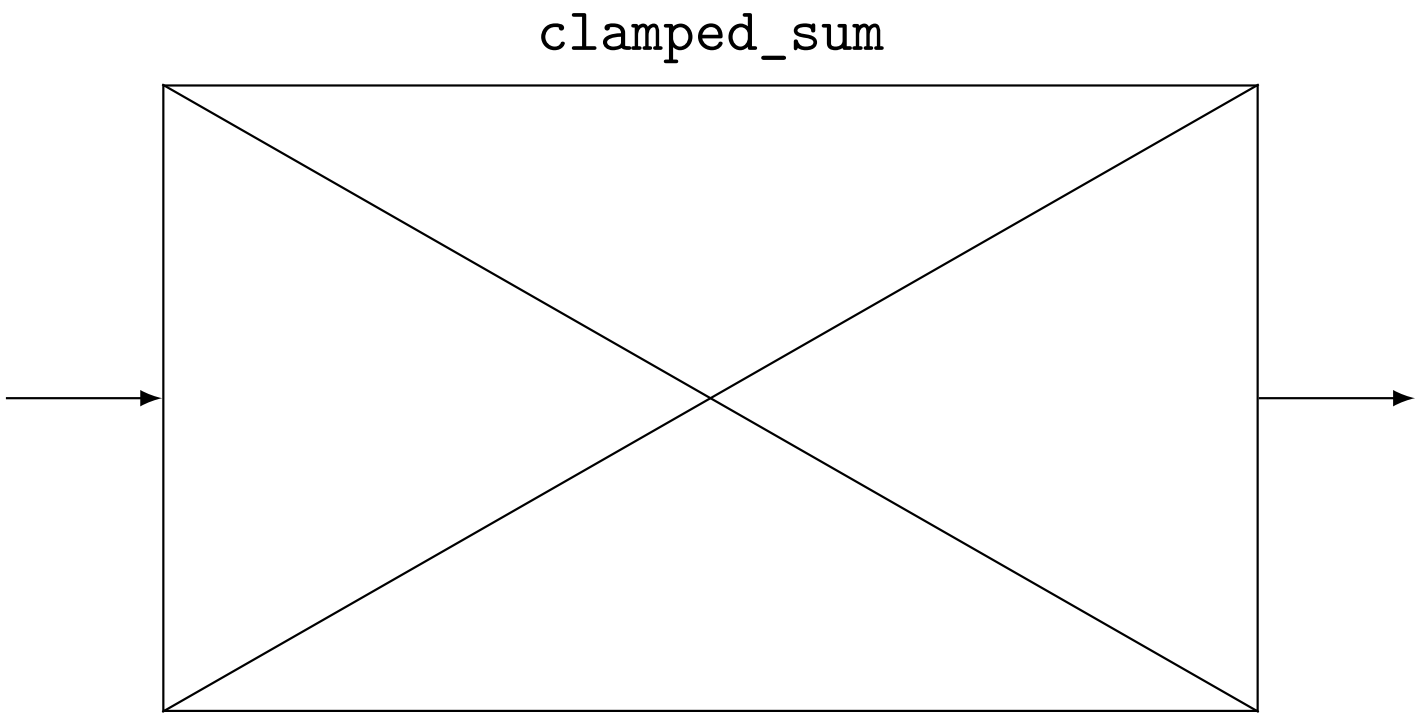
Table of contents

1. Lustre
2. Rust
3. Reducing copies
4. Language and compilation

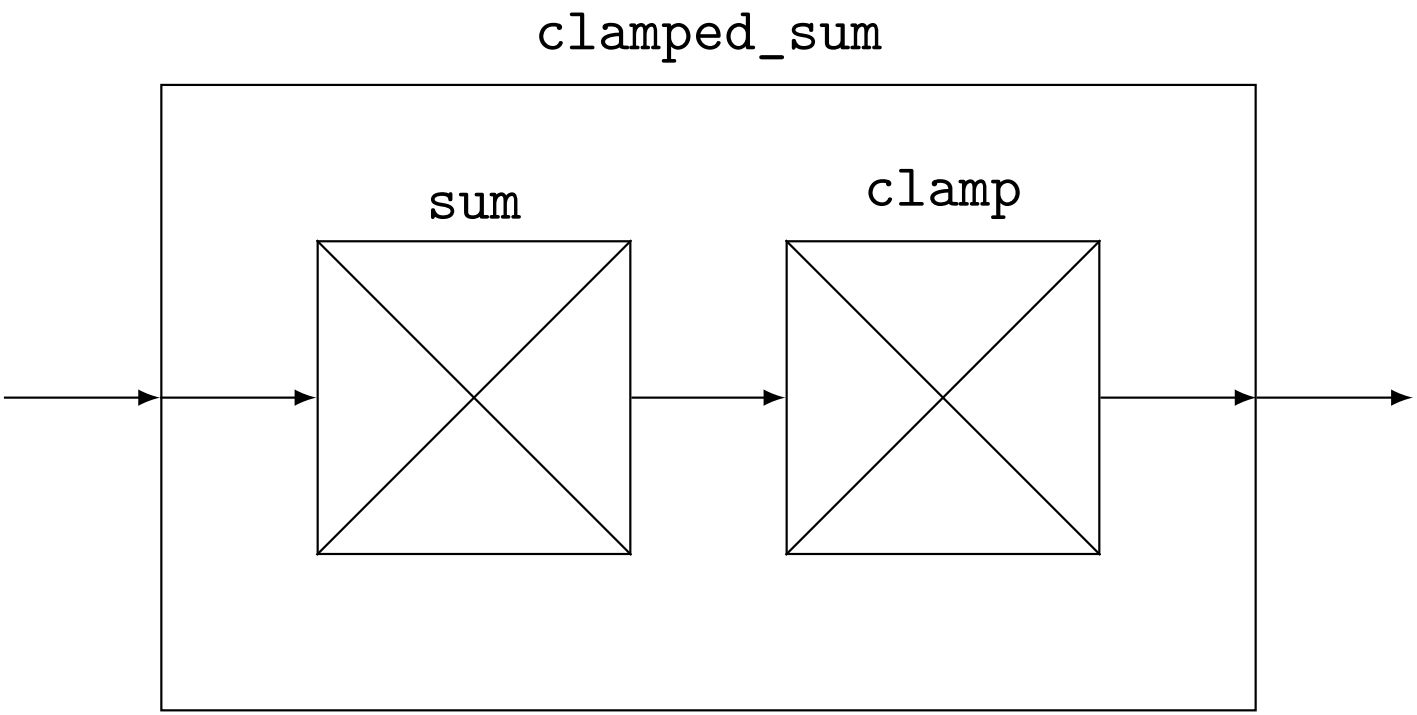
Table of contents

1. Lustre
2. Rust
3. Reducing copies
4. Language and compilation

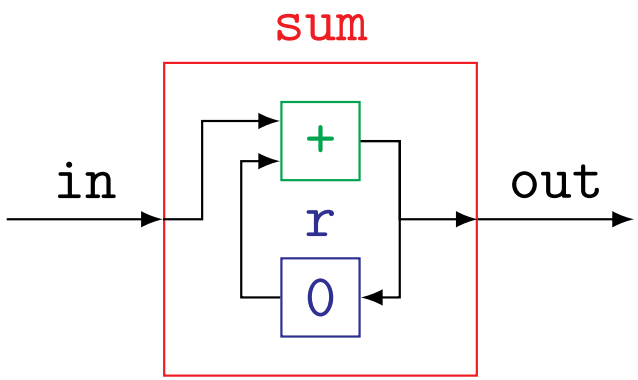
Lustre



Lustre



Lustre



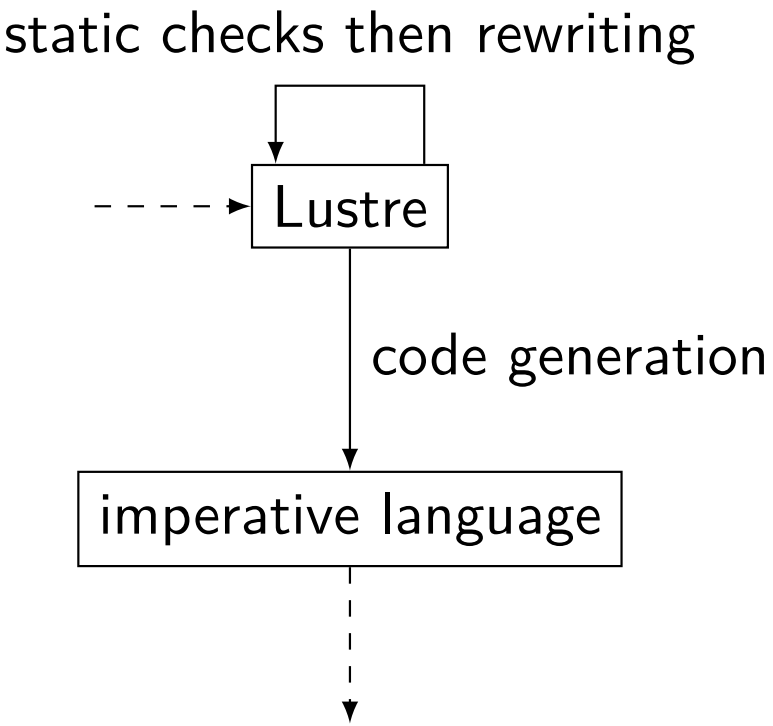

```
node sum(in: int) = (out: int)
  out = in + last r
  r init 0 = out
```

$$\forall n \in \mathbb{N}, out_n = \sum_{i=0}^{n} in_i$$

$$\forall n \in \mathbb{N}, out_n = in_n + r_{n-1}$$

$$r_{-1} = 0 \wedge \forall n \in \mathbb{N}, r_n = out_n$$

Compilation scheme



Constraints on code production

- worst case execution time for required responsiveness
- worst case memory consumption for hardware limitations

Table of contents

1. Lustre
2. Rust
3. Reducing copies
4. Language and compilation

Ownership

```
fn f(a: T);

fn g(x: T) {
  let y = x;
  f(y);
}
```

Ownership

```
fn f(a: T);

fn g(x: T) {
  let y = x;
  f(x);
}
```

Ownership

```
fn f(a: T);

fn g(x: T) {
  f(x);
  f(x);
}
```

Borrowing

```
fn f(a: &mut T, b: &T, c: &T);

fn g(x: T, y: &mut T, z: &T) {
  f(&mut x, ..., ...);
}
```

Borrowing

```
fn f(a: &mut T, b: &T, c: &T);

fn g(x: T, y: &mut T, z: &T) {
  f(&mut *y, ..., ...);
}
```

Borrowing

```
fn f(a: &mut T, b: &T, c: &T);

fn g(x: T, y: &mut T, z: &T) {
  f(&mut *z, ..., ...);
}
```

Borrowing

```
fn f(a: &mut T, b: &T, c: &T);

fn g(x: T, y: &mut T, z: &T) {
  f(..., &x, ...);
}
```

Borrowing

```
fn f(a: &mut T, b: &T, c: &T);

fn g(x: T, y: &mut T, z: &T) {
  f(..., &*y, ...);
}
```

Borrowing

```
fn f(a: &mut T, b: &T, c: &T);

fn g(x: T, y: &mut T, z: &T) {
  f(..., &*z, ...);
}
```

Borrowing

```
fn f(a: &mut T, b: &T, c: &T);

fn g(x: T, y: &mut T, z: &T) {
  f(&mut x, &x, ...);
}
```

Borrowing

```
fn f(a: &mut T, b: &T, c: &T);

fn g(x: T, y: &mut T, z: &T) {
  f(&mut *y, &*y, ...);
}
```

Borrowing

```
fn f(a: &mut T, b: &T, c: &T);

fn g(x: T, y: &mut T, z: &T) {
  f(&mut x, &*z, &*z);
}
```

Borrowing

```
fn f(a: &mut T, b: &T, c: &T);

fn g(x: T, y: &mut T, z: &T) {
  f(&mut x, &*y, &*y);
}
```

Borrowing

```
fn f(a: &mut T, b: &T, c: &T);

fn g(x: T, y: &mut T, z: &T) {
  f(&mut *y, &x, &x);
}
```

Safe Rust and Unsafe Rust

- Safe Rust statically ensures the absence of Undefined Behaviors but can't:
  - dereference raw pointers
  - call `unsafe` functions
  - access fields of `unions`
  - and more...
- Unsafe Rust can, but at the cost of fewer guarantees

Parametric polymorphism and monomorphization

```
fn choose<T>(b: bool, if_true: T, if_false: T)
  -> T {
  if !b { if_false } else { if_true }
}

... choose(b, 1, 2) ...
```

Parametric polymorphism and monomorphization

```
fn choose<T>(b: bool, if_true: T, if_false: T)
  -> T {
  if !b { if_false } else { if_true }
}

... choose::<int>(b, 1, 2) ...
```

Parametric polymorphism and monomorphization

```
fn choose<T>(b: bool, if_true: T, if_false: T)
  -> T {
  if !b { if_false } else { if_true }
}

fn choose_int(b: bool, if_true: int, if_false: int)
  -> int {
  if !b { if_false } else { if_true }
}

... choose_int(b, 1, 2) ...
```

Traits

```
trait Not {
  type Output;
  fn not(self) -> Self::Output;
}

impl Not for &bool {
  type Output = bool;

  fn not(self) -> Self::Output {
    ! *self
  }
}
```

Trait polymorphism

```
fn choose<B: Not<Output = bool>, T>
  (b: B, if_true: T, if_false: T)
  -> T {
  if b.not() { if_false } else { if_true }
}
```

Trait polymorphism

```
fn choose<B: Not<Output = bool>, T>
  (b: B, if_true: T, if_false: T)
  -> T {
  if <B as Not>::not(b) { if_false } else { if_true }
}
```

Trait polymorphism

```
fn choose<T>
   (b: impl Not<Output = bool>, if_true: T, if_false: T)
   -> T {
   if b.not() { if_false } else { if_true }
}
```

Table of contents

1. Lustre
2. Rust
3. Reducing copies
4. Language and compilation

Problem: copies introduced during scheduling

```
node fby_42(x: int) = (y: int)
  i init 0 = ((last i) + 1) % 42
  buf init 0^42 = (last buf)[i] <- x
  y = (last buf)[i]
```

Problem: copies introduced during scheduling

```
node fby_42(x: int) = (y: int)
  i init 0 = ((last i) + 1) % 42
  buf init 0^42 = (last buf)[i] <- x
  y = (last buf)[i]

struct fby_42_state {
  int i;
  int buf[42];
}
```

Problem: copies introduced during scheduling

```
node fby_42(x: int) = (y: int)
  i init 0 = ((last i) + 1) % 42
  buf init 0^42 = (last buf)[i] <- x
  y = (last buf)[i]

void fby_42_reset(struct fby_42_state *self) {
  self->i = 0;
  for (int i = 0; i < 42; i++) {
    self->buf[i] = 0;
  }
}
```

Problem: copies introduced during scheduling

```
node fby_42(x: int) = (y: int)
  i init 0 = ((last i) + 1) % 42
  buf init 0^42 = (last buf)[i] <- x
  y = (last buf)[i]

int fby_42_step(struct fby_42_state *self, int x) {
  int buf_tmp[42]; int y;
  self->i = (self->i + 1) % 42;
  memcpy(self->buf, buf_tmp, 42 * sizeof(int));
  buf_tmp[self->i] = x;
  y = self->buf[self->i];
  memcpy(buf_tmp, self->buf, 42 * sizeof(int));
  return y;
}
```

Problem: copies introduced during scheduling

```
node fby_42(x: int) = (y: int)
  i init 0 = ((last i) + 1) % 42
  y = (last buf)[i]
  buf init 0^42 = (last buf)[i] <- x

int fby_42_step(struct fby_42_state *self, int x) {
  int y;
  self->i = (self->i + 1) % 42;
  y = self->buf[self->i];
  self->buf[self->i] = x;
  return y;
}
```

Table of contents

1. Lustre
2. Rust
3. Reducing copies
4. Language and compilation

Philosophy

- idea: as early as possible, in the synchronous language, we want to talk about expression addresses in the produced imperative code, to take them into account during scheduling
- language primitives should be as copy-free as possible, making most copies explicit
- we design a highly annotated synchronous language, which would be an intermediate language in a full project
- erase locations to get semantics, erase values to get imperative code

```
struct State { ... }
impl State {
  fn reset(&mut self) { ... }
  fn step(&mut self, ...) -> ... { ... }
}
```

Tensors

```
trait Tensor {
  type Repr;
  type Idx;
  type Scal;
  fn get(ptr: NonNull<Self::Repr>, idx: Self::Idx)
    -> NonNull<Self::Scal>;
}
```

Tensors

```
impl<T> Tensor for Atom<T> {
  type Repr = T;
  type Idx = ();
  type Scal = T;

  fn get(ptr: NonNull<Self::Repr>, idx: Self::Idx)
    -> NonNull<Self::Scal> {
    ptr
  }
}
```

Tensors

```
impl<T: Tensor, const N: usize> Tensor for Array<T, N> {
  type Repr = [T::Repr; N];
  type Idx = (usize, T::Idx);
  type Scal = T::Scal;

  fn get(ptr: NonNull<Self::Repr>, idx: Self::Idx)
    -> NonNull<Self::Scal> {
    T::get(ptr.cast::<T::Repr>().add(idx.0), idx.1)
  }
}
```

Views

```
trait View: Copy {
  type Tensor: Tensor;
  fn get(self, idx: <Self::Tensor as Tensor>::Idx)
    -> NonNull<<Self::Tensor as Tensor>::Scal>;
}
```

Views

```
struct Pointer<T: Tensor>(NonNull<T::Repr>);

impl<T: Tensor> View for Pointer<T> {
  type Tensor = T;

  fn get(self, idx: <Self::Tensor as Tensor>::Idx)
    -> NonNull<<Self::Tensor as Tensor>::Scal> {
    T::get(self.0, idx)
  }
}
```

Views

```
struct Projection<V: View>(V, usize);

impl<T: Tensor, const N: usize,
     V: View<Tensor = Array<T, N>>>
  View for Projection<V> {
  type Tensor = T;

  fn get(self, idx: <Self::Tensor as Tensor>::Idx)
    -> NonNull<<Self::Tensor as Tensor>::Scal> {
    self.0.get((self.1, idx))
  }
}

Projection(v, i) = v[i < N]
```

Views

```
struct CutLeft<V: View, const L: usize>(V);

impl<T: Tensor, const N: usize,
     V: View<Tensor = Array<T, N>>, const L: usize>
  View for CutLeft<V, L> {
  type Tensor = Array<T, L>;

  fn get(self, idx: <Self::Tensor as Tensor>::Idx)
    -> NonNull<<Self::Tensor as Tensor>::Scal> {
    self.0.get(idx)
  }
}
```

(with $\mathtt{L} < \mathtt{N}$)

$\mathtt{CutLeft::<\_, L>(v)} = \lambda i < \mathtt{L}.\ \mathtt{v}[i < \mathtt{N}]$

Views

```
struct CutRight
  <V: View, const L: usize, const R: usize>(V);

impl<T: Tensor, const N: usize,
     V: View<Tensor = Array<T, N>>,
     const L: usize, const R: usize>
  View for CutLeft<V, L, R> {
  type Tensor = Array<T, R>;

  fn get(self, idx: <Self::Tensor as Tensor>::Idx)
    -> NonNull<<Self::Tensor as Tensor>::Scal> {
    self.0.get((idx.0 + L, idx.1))
  }
}
```

(with $\mathtt{L} + \mathtt{R} = \mathtt{N}$)

$\mathtt{CutRight::<\_, L, R>(v)} = \lambda i < \mathtt{R}.\ \mathtt{v}[i + \mathtt{L} < \mathtt{N}]$

Views

```
struct Repeat<V: View, const N: usize>(V);

impl<V: View, const N: usize> View for Repeat<V, N> {
  type Tensor = Array<V::Tensor, N>;

  fn get(self, idx: <Self::Tensor as Tensor>::Idx)
    -> NonNull<<Self::Tensor as Tensor>::Scal> {
    self.0.get(idx.1)
  }
}
```

`Repeat::<_, N>(v)` $= \lambda i < \mathtt{N}.\ \mathtt{v}$

Views

```
struct Single<V: View>(V);

impl<V: View> View for Single<V> {
  type Tensor = Array<V::Tensor, 1>;

  fn get(self, idx: <Self::Tensor as Tensor>::Idx)
    -> NonNull<<Self::Tensor as Tensor>::Scal> {
    self.0.get(idx.1)
  }
}
```

$\texttt{Single(v)} = \lambda i < 1.\ \texttt{v}$

Views

```
struct Extract<V: View>(V);

impl<T: Tensor, V: View<Tensor = Array<T, 1>>>
  View for Extract<V> {
  type Tensor = T;

  fn get(self, idx: <Self::Tensor as Tensor>::Idx)
    -> NonNull<<Self::Tensor as Tensor>::Scal> {
    self.0.get((0, idx))
  }
}
```

$\texttt{Extract(v)} = \texttt{v}[0 < 1]$

Views

```
struct Reverse<V: View>(V);

impl<T: Tensor, const N: usize,
     V: View<Tensor = Array<T, N>>>
  View for Reverse<V> {
  type Tensor = Array<T, N>;

  fn get(self, idx: <Self::Tensor as Tensor>::Idx)
    -> NonNull<<Self::Tensor as Tensor>::Scal> {
    self.0.get((N - 1 - idx.0, idx.1))
  }
}
```

$\texttt{Reverse(v)} = \lambda i < \texttt{N}.\ \texttt{v}[\texttt{N} - 1 - i < \texttt{N}]$

Views

```
struct Transpose<V: View>(V);

impl<T: Tensor, const N: usize, const M: usize,
     V: View<Tensor = Array<Array<T, M>, N>>>
  View for Transpose<V> {
  type Tensor = Array<Array<T, N>, M>;

  fn get(self, idx: <Self::Tensor as Tensor>::Idx)
    -> NonNull<<Self::Tensor as Tensor>::Scal> {
    self.0.get((idx.1.0, (idx.0, idx.1.1)))
  }
}
```

$\texttt{Transpose(v)} = \lambda i < \texttt{M}.\ \lambda j < \texttt{N}.\ \texttt{v}\,[j < \texttt{N}]\,[i < \texttt{M}]$

Views

```
struct Split<V: View, const L: usize, const C: usize>(V);

impl<T: Tensor, const N: usize,
     V: View<Tensor = Array<T, N>>,
     const L: usize, const C: usize>
  View for Split<V, L, C> {
  type Tensor = Array<Array<T, C>, L>;

  fn get(self, idx: <Self::Tensor as Tensor>::Idx)
    -> NonNull<<Self::Tensor as Tensor>::Scal> {
    self.0.get((idx.0 * C + idx.1.0, idx.1.1))
  }
}
```

(with $\mathtt{L} \times \mathtt{C} = \mathtt{N}$)

$\mathtt{Split::<\_, L, C>(v)} = \lambda i < \mathtt{L}.\ \lambda j < \mathtt{C}.\ \mathtt{v}[i \times \mathtt{C} + j < \mathtt{N}]$

Views

```
struct Window<V: View, const W: usize, const S: usize>(V);

impl<T: Tensor, const N: usize,
     V: View<Tensor = Array<T, N>>,
     const W: usize, const S: usize>
  View for Window<V, W, S> {
  type Tensor = Array<Array<T, W>, S>;

  fn get(self, idx: <Self::Tensor as Tensor>::Idx)
    -> NonNull<<Self::Tensor as Tensor>::Scal> {
    self.0.get((idx.0 + idx.1.0, idx.1.1))
  }
}
```

(with $\mathtt{W} + \mathtt{S} - 1 = \mathtt{N}$)

$\mathtt{Window::<\_, W, S>(v)} = \lambda i < \mathtt{S}.\ \lambda j < \mathtt{W}.\ \mathtt{v}[i + j < \mathtt{N}]$

Views

```
struct Sample<V: View, const S: usize, const SS: usize>(V);

impl<T: Tensor, const N: usize,
     V: View<Tensor = Array<T, N>>,
     const S: usize, const SS: usize>
  View for Sample<V, S, SS> {
  type Tensor = Array<T, SS>;

  fn get(self, idx: <Self::Tensor as Tensor>::Idx)
    -> NonNull<<Self::Tensor as Tensor>::Scal> {
    self.0.get((idx.0 * S, idx.1))
  }
}
```

(with $\mathtt{S} \times (\mathtt{SS} - 1) + 1 = \mathtt{N}$)

$\mathtt{Sample::<\_, S, SS>(v)} = \lambda i < \mathtt{SS}.\ \mathtt{v}[i \times S < \mathtt{N}]$

Calling conventions

```
node(&l_in: int, x @ l_in, ...) = ...

fn step(&mut self,
        l_in: impl View<Tensor = Atom<int>>,
        ...)
  -> ...;
```

Calling conventions

```
node(&l_in: int, &l_out: int, x @ l_in) = (y @ l_out, ...)

fn step(&mut self,
        l_in: impl View<Tensor = Atom<int>>,
        l_out: impl View<Tensor = Atom<int>>)
  -> ...;
```

Calling conventions

```
node(&l_in: int, &l_out: int, x @ l_in) = (y @ l_out)

fn step(&mut self,
        l_in: impl View<Tensor = Atom<int>>,
        l_out: impl View<Tensor = Atom<int>>);
```

Calling conventions

```
node(&l_in: int, x @ l_in) = (&l_out: int, y @ l_out)

fn step(&mut self,
        l_in: impl View<Tensor = Atom<int>>)
  -> impl View<Tensor = Atom<int>>;
```

Calling conventions

```
node(&l_in: int, &mut l_out: int, x @ l_in) = (y @ l_out)

fn step(&mut self,
        l_in: impl View<Tensor = Atom<int>>,
        l_out: impl View<Tensor = Atom<int>>);
```

Calling conventions

```
node(&mut l_in: int, &mut l_out: int, x @ l_in) =
  (y @ l_out)

fn step(&mut self,
        l_in: impl View<Tensor = Atom<int>>,
        l_out: impl View<Tensor = Atom<int>>);
```

Memories

```
memories (fby_0_n init 0: int @ l_mem)

struct State {
  l_mem: MaybeUninit<<Atom<int> as Tensor>::Repr>,
  ...
}

fn reset(&mut self) { self.l_mem.write(0); ... }

fn step(&mut self, ...) {
  let l_mem: Pointer<Atom<int>> =
    Pointer(
      NonNull::new_unchecked(
        self.l_mem.as_mut_ptr()));
  ...
}
```

Stack

```
stack (&mut l_loc: int)

fn step(&mut self, ...) {
  let l_loc: Pointer<Atom<int>> =
    Pointer(
      NonNull::new_unchecked(
        MaybeUninit::uninit().as_mut_ptr()));
  ...
}
```

Variables locales

```
locals (a @ l_in) (b @ l_out) (c @ l_mem) (d @ l_loc)
```

Clocks and views

```
node ex(c: bool, in) = (out)
  tmp = f(in when c)
  out = g(merge(c, tmp, in when not c))
```

c	T	F	T	T	F	...
in	in_0	in_1	in_2	in_3	in_4	...
in when c	in_0		in_2	in_3		...
tmp	tmp_0		tmp_2	tmp_3		...
in when not c		in_1			in_4	...
merge(c, tmp, in when not c)	tmp_0	in_1	tmp_2	tmp_3	in_4	...
out	out_0	out_1	out_2	out_3	out_4	...

Clocks and views

```
node ex(c: bool, in) = (out)
  tmp = f(in when c)
  out = g(merge(c, tmp, in when not c))
```

c	T	F	T	T	F	...
in	in_0	in_1	in_2	in_3	in_4	...
in when c	in_0	?	in_2	in_3	?	...
tmp	tmp_0	?	tmp_2	tmp_3	?	...
in when not c	?	in_1	?	?	in_4	...
merge(c, tmp, in when not c)	tmp_0	in_1	tmp_2	tmp_3	in_4	...
out	out_0	out_1	out_2	out_3	out_4	...

Clocks and views

$$\frac{\texttt{c: } C_1 \texttt{ + } \cdots \texttt{ + } C_n \qquad \texttt{x @ l}}{\texttt{x when c = } C_i \texttt{ @ l on c = } C_i}$$

$$\frac{\texttt{c: } C_1 \texttt{ + } \cdots \texttt{ + } C_n \qquad \forall i,\ \texttt{e}_i \texttt{ @ l on c = } C_i}{\texttt{merge(c, } C_1 \texttt{ => e}_1\texttt{, } \cdots\texttt{, } C_n \texttt{ => e}_n\texttt{) @ l}}$$

Arrays and views

$$\frac{\texttt{l: [T; M + N]} \qquad \texttt{x @ l}}{\texttt{fst(x as [\_; M] ++ [\_; N]) @ fst(l as [\_; M] ++ [\_; N])}}$$

$$\frac{\texttt{l: [T; M + N]} \qquad \texttt{x @ l}}{\texttt{snd(x as [\_; M] ++ [\_; N]) @ snd(l as [\_; M] ++ [\_; N])}}$$

$$\frac{\begin{array}{l}\texttt{x @ fst(l as [\_; M] ++ [\_; N])} \\ \texttt{y @ snd(l as [\_; M] ++ [\_; N])}\end{array}}{\texttt{x ++ y @ l}}$$

Arrays and views

$$\frac{\texttt{l: [T; N]} \qquad \texttt{x @ l}}{\texttt{x[i] @ l[i]}}$$

Arrays and views

$$\frac{\texttt{l: [T; N]} \qquad \texttt{x @ l}}{\texttt{x[i] @ l[i]}} \qquad\qquad \frac{\texttt{l: [T; N]} \qquad \texttt{x @ l}}{\texttt{x[\textbackslash i] @ l[\textbackslash i]}}$$

$$\frac{\texttt{x @ l[i]} \qquad \texttt{y @ l[\textbackslash i]}}{\texttt{update(y, x) @ l}}$$

$$\texttt{[arr with i <- x]} = \texttt{update(arr[\textbackslash i], x)}$$

Arrays and views

$$\frac{\texttt{x @ l}}{\texttt{[x] * N @ [l] * N}}$$

Interference of views

$$\frac{\mathtt{l}_1 \neq \mathtt{l}_2}{\mathtt{l}_1 \parallel \mathtt{l}_2} \qquad \frac{i \neq j}{\mathtt{v\ on\ c} = \mathtt{C}_i \parallel \mathtt{v\ on\ c} = \mathtt{C}_j}$$

$$\frac{}{\mathtt{fst(v\ as\ [\_;\ M]\ ++\ [\_;\ N])} \parallel \mathtt{snd(v\ as\ [\_;\ M]\ ++\ [\_;\ N])}}$$

$$\frac{}{\mathtt{v[i]} \parallel \mathtt{v[\backslash i]}}$$

$$\frac{\mathtt{v}_1 \parallel \mathtt{v}_2}{\mathtt{v}_2 \parallel \mathtt{v}_1} \qquad \frac{\mathtt{v}_1 \parallel \mathtt{v}_2}{\mathtt{v}_1\ \mathtt{on\ c} = \mathtt{C} \parallel \mathtt{v}_2} \qquad \frac{\mathtt{v}_1 \parallel \mathtt{v}_2}{[\mathtt{v}_1]\ \mathtt{*\ N} \parallel \mathtt{v}_2} \qquad \ldots$$

Node call and reset

```
b = (f on c1 = C1 reset every c2 = C2, c3 = C3)
      (N; idx, e_idx; &l, a, &mut l' on a = true)
```

$\forall s, \forall i_1\ i_2,$
$\forall(\&l_1\text{: bool})(x\ @\ l_1)(\&\text{mut}\ l_2\text{: [int; s] on } x = \text{true}),$
$\exists(y\ @\ l_2[i_1])$

```
struct State {
  instance_f_42: State_f<N>,
  ...
}
if let C1 = l_c1.get(()).read() {
  self.instance_f_42
    .step(idx, l_e_idx.get(()).read() as usize, l, l');
}
```

Node call and reset

```
b = (f on c1 = C1 reset every c2 = C2, c3 = C3)
      (N; idx, e_idx; &l, a, &mut l' on a = true)
```

$\forall s, \forall i_1\ i_2,$

$\forall(\&l_1\text{: bool})(x\ @\ l_1)(\&\text{mut}\ l_2\text{: [int; s] on } x = \text{true}),$

$\exists(y\ @\ l_2[i_1])$

```
struct State {
  instance_f_42: State_f<N>,
  ...
}
if let C2 = l_c2.get(()).read() {
  self.instance_f_42.reset();
}
```

Node call and reset

```
b = (f on c1 = C1 reset every c2 = C2, c3 = C3)
      (N; idx, e_idx; &l, a, &mut l' on a = true)
```

$\forall s, \forall i_1\ i_2,$
$\forall(\&l_1\text{: bool})(x\ @\ l_1)(\&\text{mut}\ l_2\text{: [int; s] on } x = \text{true}),$
$\exists(y\ @\ l_2[i_1])$

```
struct State {
  instance_f_42: State_f<N>,
  ...
}
if let C3 = l_c3.get(()).read() {
  self.instance_f_42.reset();
}
```

Node call and reset

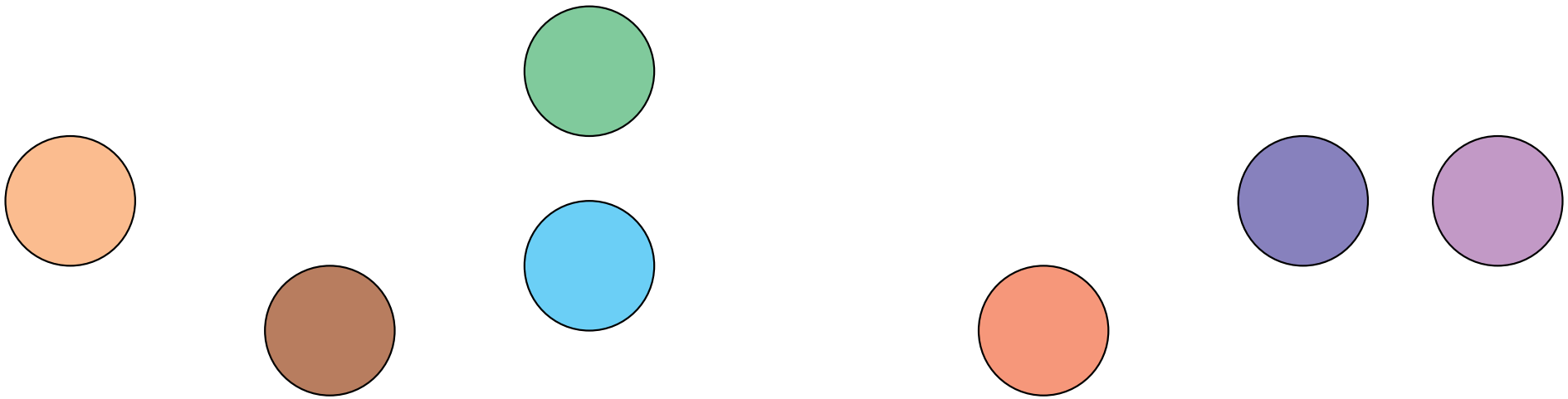
```
b = (f on c1 = C1 reset every c2 = C2, c3 = C3)
      (N; idx, e_idx; &l, a, &mut l' on a = true)
```

$$\forall s, \forall i_1\ i_2,$$
$$\forall(\&l_1\text{: bool})(x\ @\ l_1)(\&\text{mut } l_2\text{: [int; s] on } x = \text{true}),$$
$$\exists(y\ @\ l_2[i_1])$$

```
struct State {
  instance_f_42: State_f_N,
  ...
}
if let C3 = l_c3.get(()).read() {
  self.instance_f_42.reset();
}
```

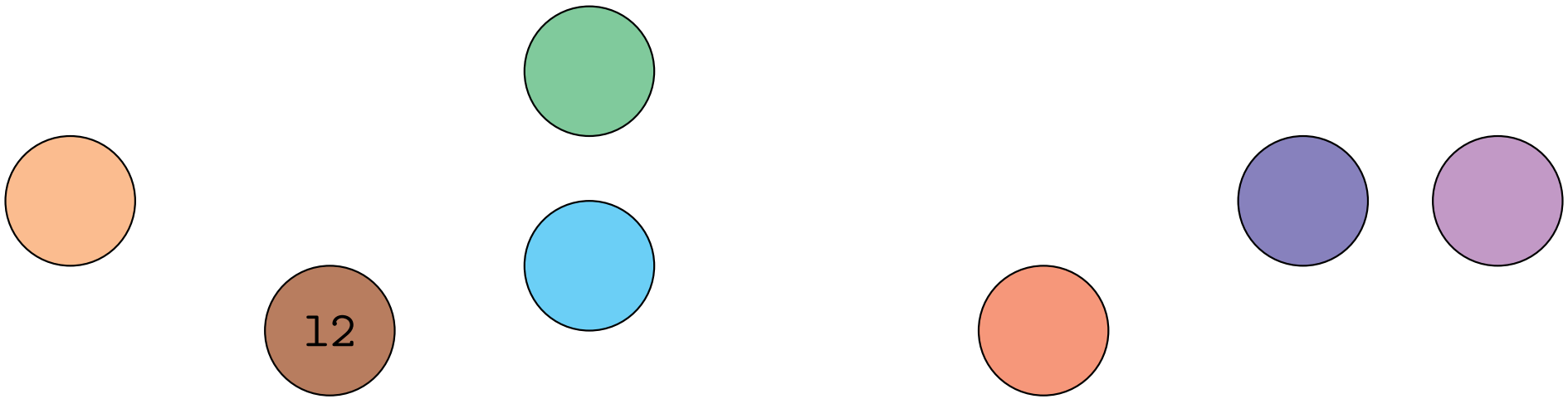
Scheduling

```
node(&mut l1: int, x @ l1) = (z @ l1)
  stack (&mut l2: bool)
  locals (c1 @ l2) (c2 @ l2) (y @ l1 on c1)
  c1 = i(&mut l2)
  c2 = h(&mut l2)
  y = (f on c1)(&mut l1 on c1, x when c1)
  z = (g reset every c1)
        (&l2, &mut l1, c2, merge(c1, y, x whenot c1))
```



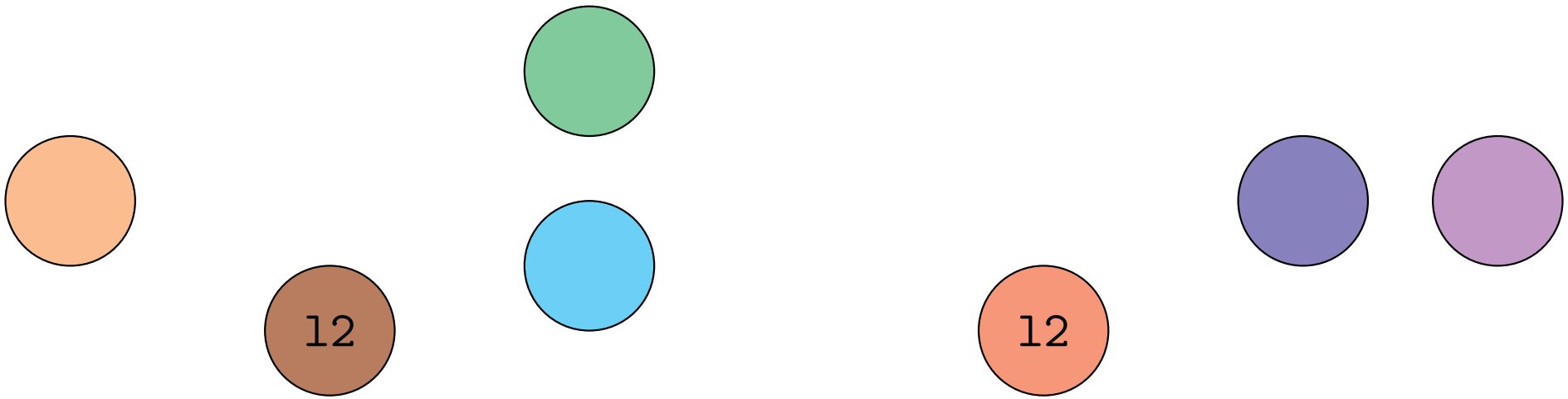
Scheduling

```
node(&mut l1: int, x @ l1) = (z @ l1)
  stack (&mut l2: bool)
  locals (c1 @ l2) (c2 @ l2) (y @ l1 on c1)
  c1 = i(&mut l2)
  c2 = h(&mut l2)
  y = (f on c1)(&mut l1 on c1, x when c1)
  z = (g reset every c1)
        (&l2, &mut l1, c2, merge(c1, y, x whenot c1))
```



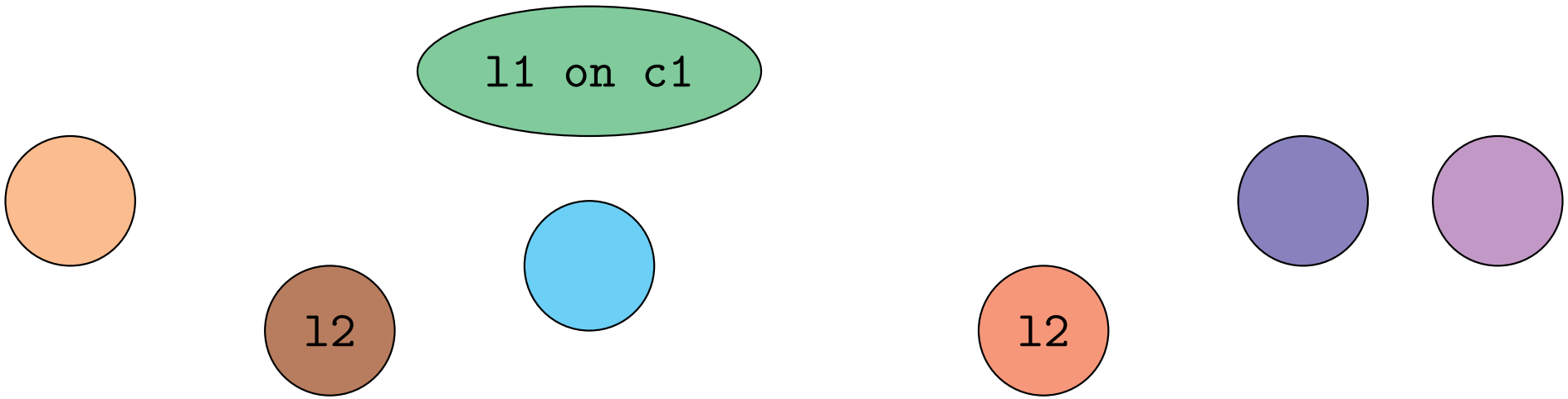
Scheduling

```
node(&mut l1: int, x @ l1) = (z @ l1)
  stack (&mut l2: bool)
  locals (c1 @ l2) (c2 @ l2) (y @ l1 on c1)
  c1 = i(&mut l2)
  c2 = h(&mut l2)
  y = (f on c1)(&mut l1 on c1, x when c1)
  z = (g reset every c1)
        (&l2, &mut l1, c2, merge(c1, y, x whenot c1))
```



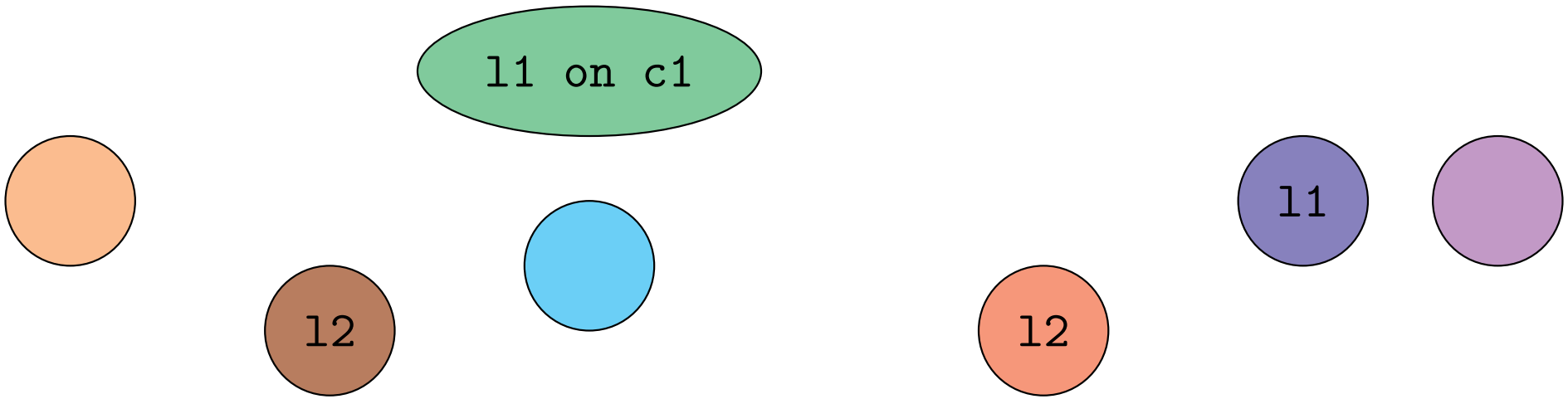
Scheduling

```
node(&mut l1: int, x @ l1) = (z @ l1)
  stack (&mut l2: bool)
  locals (c1 @ l2) (c2 @ l2) (y @ l1 on c1)
  c1 = i(&mut l2)
  c2 = h(&mut l2)
  y = (f on c1)(&mut l1 on c1, x when c1)
  z = (g reset every c1)
        (&l2, &mut l1, c2, merge(c1, y, x whenot c1))
```



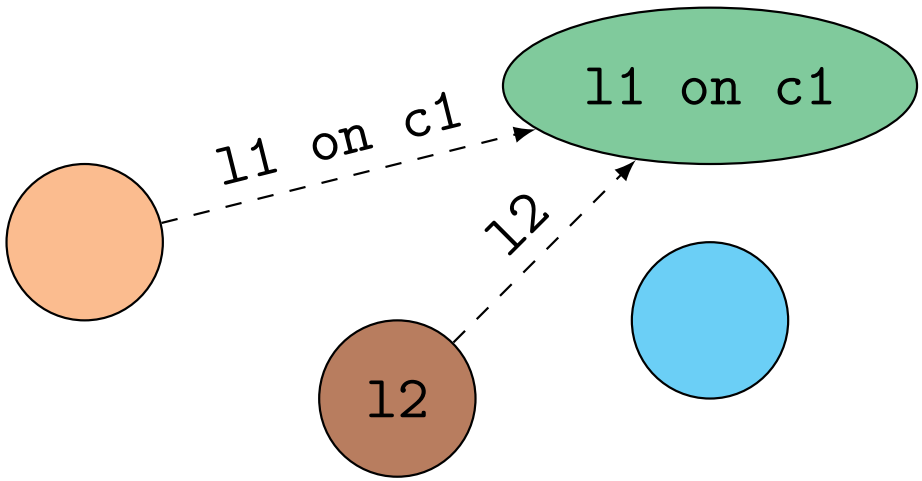
Scheduling

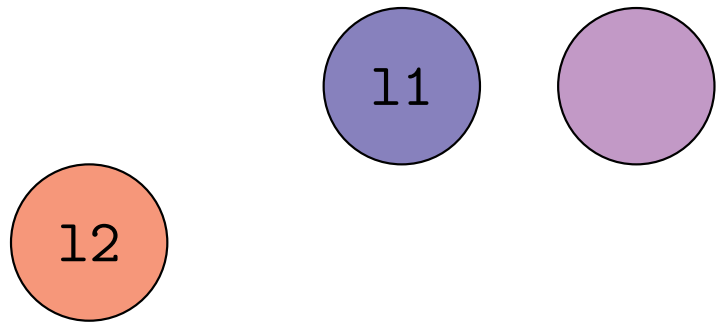
```
node(&mut l1: int, x @ l1) = (z @ l1)
  stack (&mut l2: bool)
  locals (c1 @ l2) (c2 @ l2) (y @ l1 on c1)
  c1 = i(&mut l2)
  c2 = h(&mut l2)
  y = (f on c1)(&mut l1 on c1, x when c1)
  z = (g reset every c1)
        (&l2, &mut l1, c2, merge(c1, y, x whenot c1))
```



Scheduling

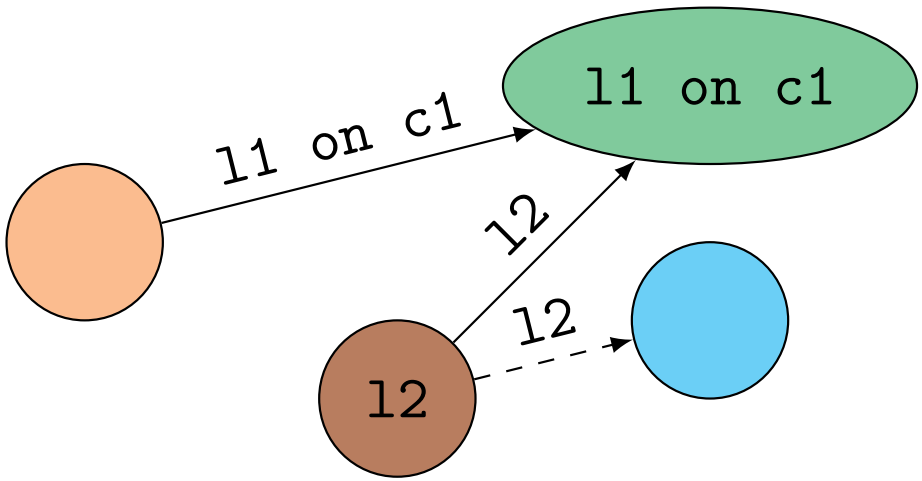
```
node(&mut l1: int, x @ l1) = (z @ l1)
  stack (&mut l2: bool)
  locals (c1 @ l2) (c2 @ l2) (y @ l1 on c1)
  c1 = i(&mut l2)
  c2 = h(&mut l2)
  y = (f on c1)(&mut l1 on c1, x when c1)
  z = (g reset every c1)
        (&l2, &mut l1, c2, merge(c1, y, x whenot c1))
```

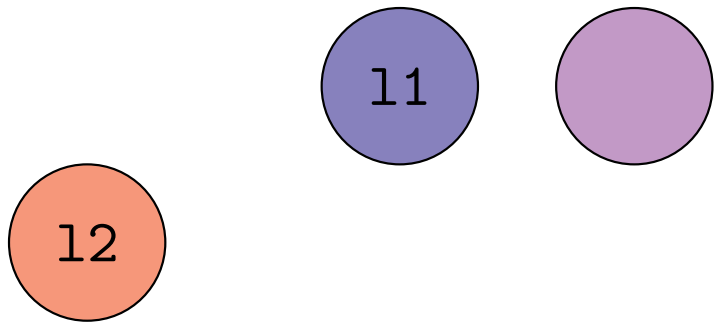




Scheduling

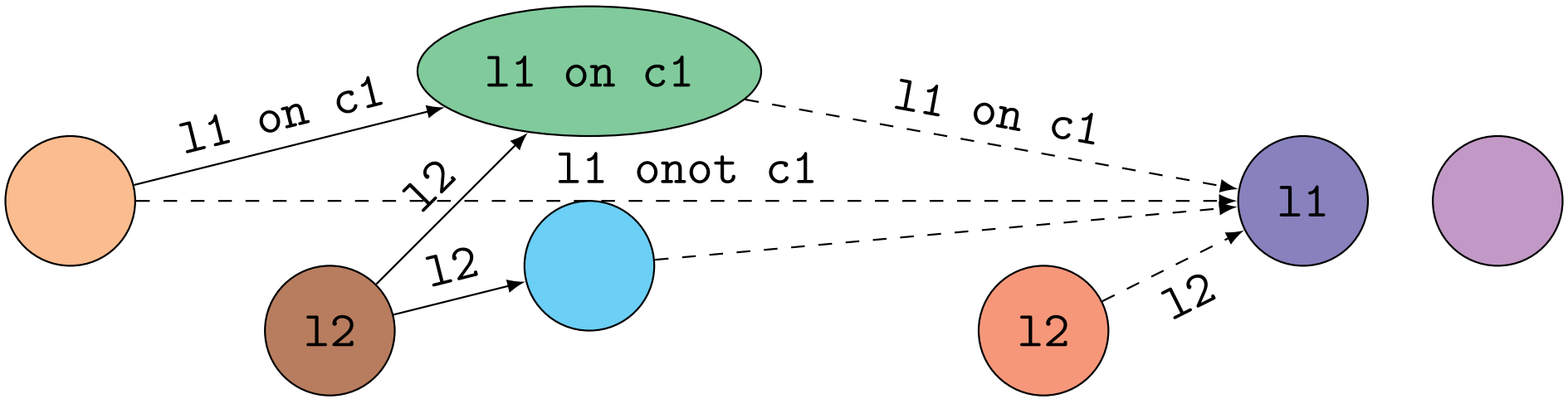
```
node(&mut l1: int, x @ l1) = (z @ l1)
  stack (&mut l2: bool)
  locals (c1 @ l2) (c2 @ l2) (y @ l1 on c1)
  c1 = i(&mut l2)
  c2 = h(&mut l2)
  y = (f on c1)(&mut l1 on c1, x when c1)
  z = (g reset every c1)
        (&l2, &mut l1, c2, merge(c1, y, x whenot c1))
```





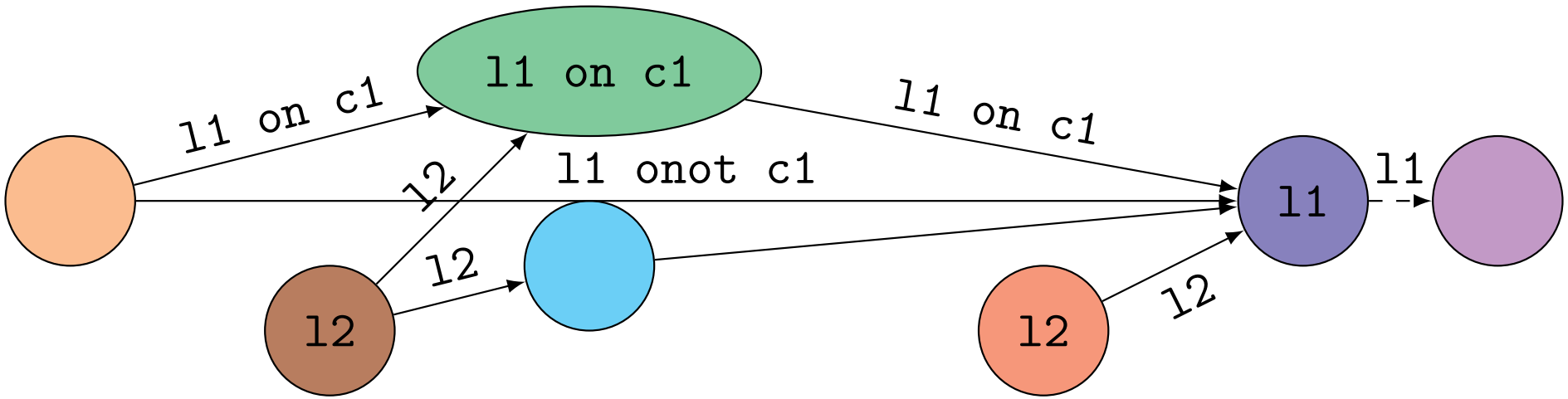
Scheduling

```
node(&mut l1: int, x @ l1) = (z @ l1)
  stack (&mut l2: bool)
  locals (c1 @ l2) (c2 @ l2) (y @ l1 on c1)
  c1 = i(&mut l2)
  c2 = h(&mut l2)
  y = (f on c1)(&mut l1 on c1, x when c1)
  z = (g reset every c1)
        (&l2, &mut l1, c2, merge(c1, y, x whenot c1))
```



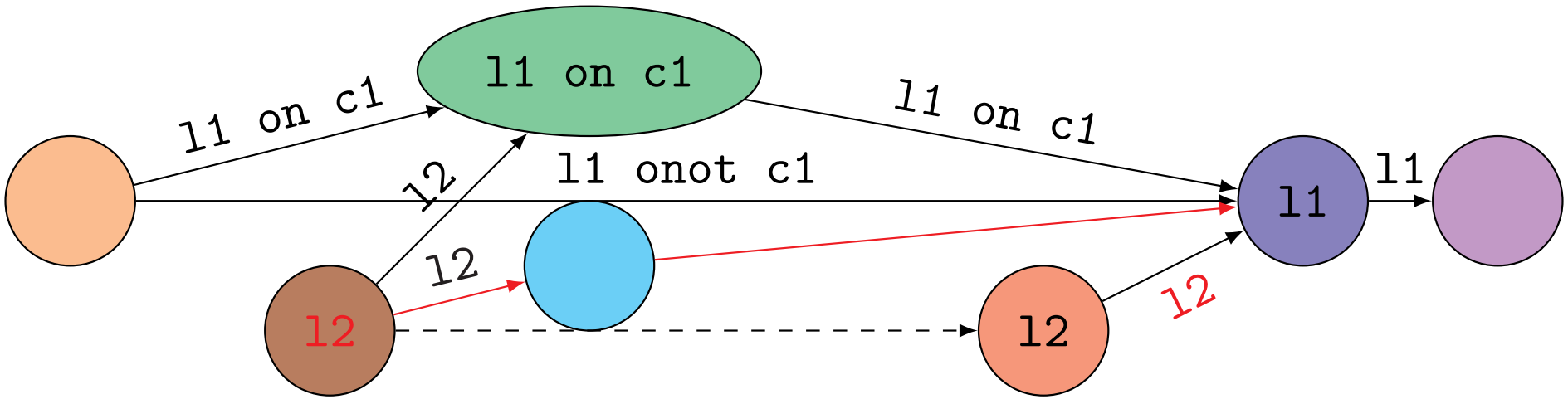
Scheduling

```
node(&mut l1: int, x @ l1) = (z @ l1)
  stack (&mut l2: bool)
  locals (c1 @ l2) (c2 @ l2) (y @ l1 on c1)
  c1 = i(&mut l2)
  c2 = h(&mut l2)
  y = (f on c1)(&mut l1 on c1, x when c1)
  z = (g reset every c1)
        (&l2, &mut l1, c2, merge(c1, y, x whenot c1))
```



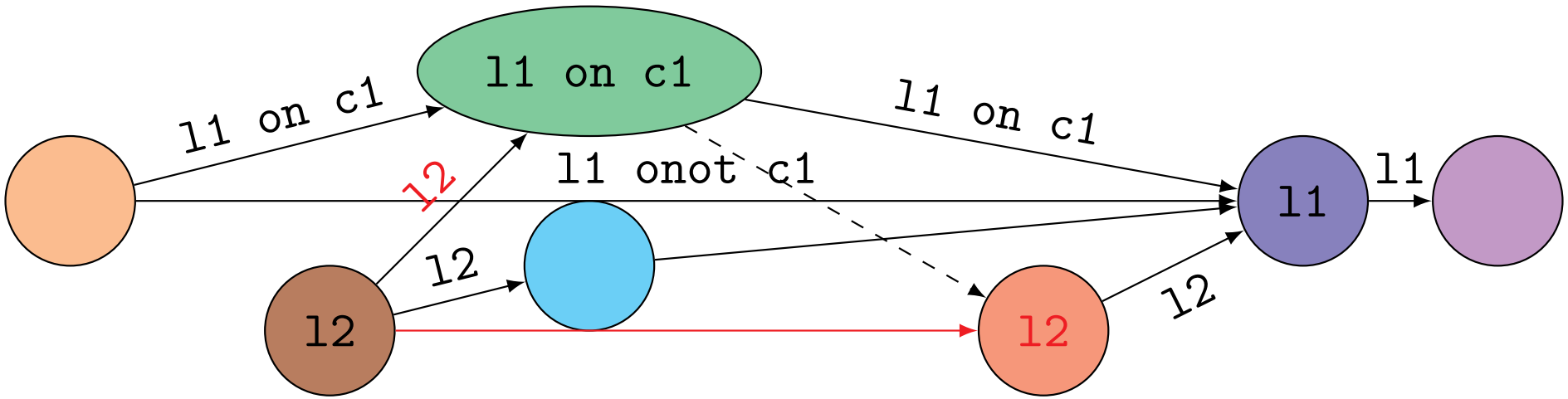
Scheduling

```
node(&mut l1: int, x @ l1) = (z @ l1)
  stack (&mut l2: bool)
  locals (c1 @ l2) (c2 @ l2) (y @ l1 on c1)
  c1 = i(&mut l2)
  c2 = h(&mut l2)
  y = (f on c1)(&mut l1 on c1, x when c1)
  z = (g reset every c1)
        (&l2, &mut l1, c2, merge(c1, y, x whenot c1))
```



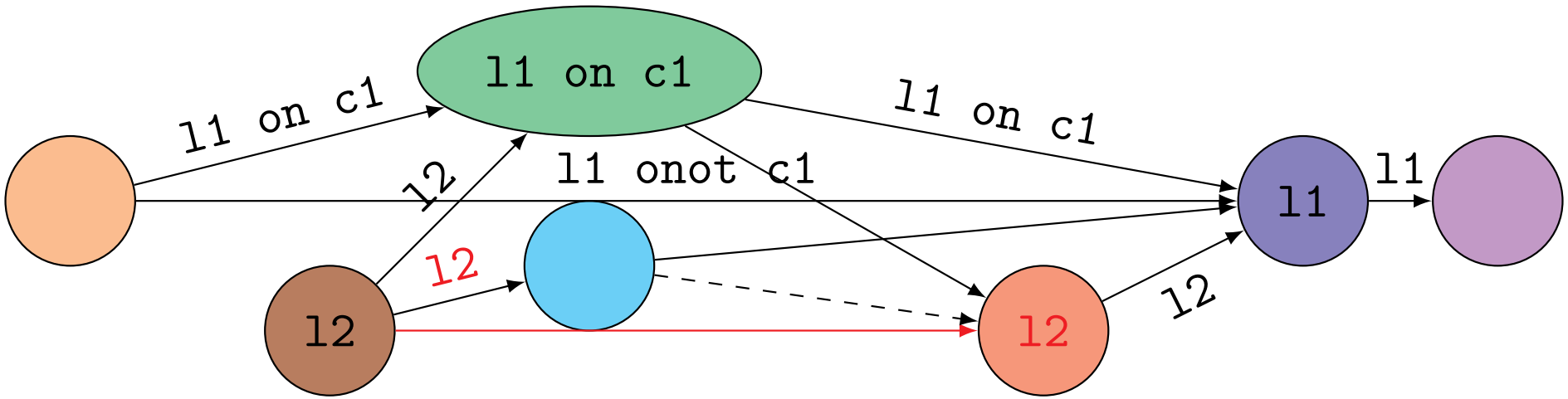
Scheduling

```
node(&mut l1: int, x @ l1) = (z @ l1)
  stack (&mut l2: bool)
  locals (c1 @ l2) (c2 @ l2) (y @ l1 on c1)
  c1 = i(&mut l2)
  c2 = h(&mut l2)
  y = (f on c1)(&mut l1 on c1, x when c1)
  z = (g reset every c1)
        (&l2, &mut l1, c2, merge(c1, y, x whenot c1))
```



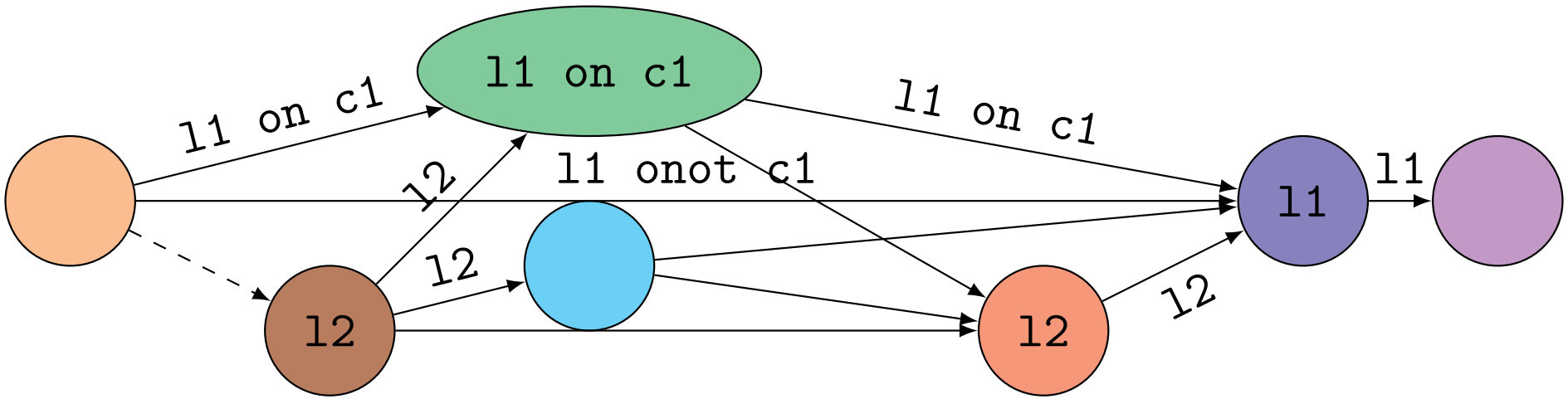
Scheduling

```
node(&mut l1: int, x @ l1) = (z @ l1)
  stack (&mut l2: bool)
  locals (c1 @ l2) (c2 @ l2) (y @ l1 on c1)
  c1 = i(&mut l2)
  c2 = h(&mut l2)
  y = (f on c1)(&mut l1 on c1, x when c1)
  z = (g reset every c1)
        (&l2, &mut l1, c2, merge(c1, y, x whenot c1))
```



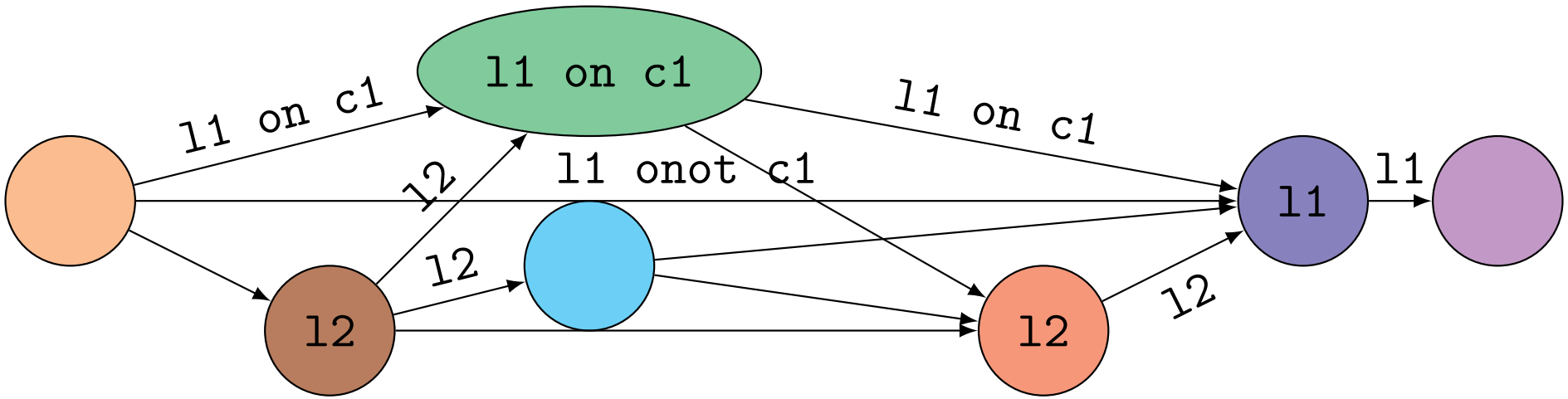
Scheduling

```
node(&mut l1: int, x @ l1) = (z @ l1)
  stack (&mut l2: bool)
  locals (c1 @ l2) (c2 @ l2) (y @ l1 on c1)
  c1 = i(&mut l2)
  c2 = h(&mut l2)
  y = (f on c1)(&mut l1 on c1, x when c1)
  z = (g reset every c1)
        (&l2, &mut l1, c2, merge(c1, y, x whenot c1))
```



Scheduling

```
node(&mut l1: int, x @ l1) = (z @ l1)
  stack (&mut l2: bool)
  locals (c1 @ l2) (c2 @ l2) (y @ l1 on c1)
  c1 = i(&mut l2)
  c2 = h(&mut l2)
  y = (f on c1)(&mut l1 on c1, x when c1)
  z = (g reset every c1)
        (&l2, &mut l1, c2, merge(c1, y, x whenot c1))
```



Thank you for your attention!
Any questions?