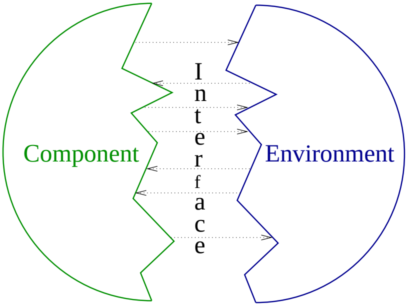
Heap-Abstraction for an Object-Oriented Calculus with Thread Classes

E. Ábrahám A. Grüner M. Steffen

Albert-Ludwigs-University Freiburg, Christian-Albrechts University Kiel

Starting point

- component = “program fragment” = “open program”
- environment = “context” = “observer”
- $\rightsquigarrow$ compositional semantics

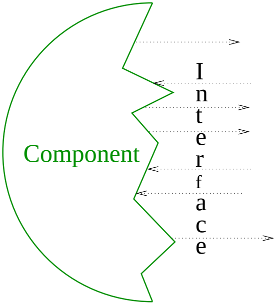


Starting point

- question:

> what's observable of an open class-based, object-oriented, multi-threaded program

- goal: fully-abstract semantics



Full abstraction

- natural definition of equivalence of program fragments
- basically: comparison between two semantics, resp. two implied notions of equality
- given a reference semantics, the 2nd one is
  - neither too abstract = sound
  - nor too concrete = complete
- Milner, Plotkin: λ-calculus
- Jeffrey, Rathke: concurrent ν-calculus

Notion of observation: Reference semantics

```
// component
public class P {
    public static void main(String[] arg) {
        O x = new O();
        x.m(42);
    }
}
```

Notion of observation: Reference semantics

```
// component
public class P {
    public static void main(String[] arg) {
        O x = new O();
        x.m(42);
    }
}
```

```
// external observer
class O {
    public void m(int x) {
      <some code>;
      System.out.println("success");
    }
}
```

Notion of observation: Reference semantics

- pretty simple observational notion: "may-testing":

  compose a component with an observer, let it run and see, whether the observer may be/is successful

- $P_1 \sqsubseteq_{may} P_2$: for all observers O: if $P_1 + O$ may be/is successful, then so may be/is $P_2 + O$.

Classes?

- open semantics (based on may testing/observational equivalence): in principle: straightforward and understood

⇒ corresponding semantics is "traces" as interface interactions (messages, method calls and returns)

what is the semantical import of classes?

- 3 issues:
  1. interface separates component and observer classes
  2. class = generators of object (via `new`)[1]

  ⇒ instantiation requests as interface interaction

[1] Classes in *Java* or $C^{\#}$ serve also as kind of types, and furthermore for inheritance. We ignore that mostly here.

Problems to tackle for an open f-a semantics

- "message passing"[2] framework ⇒ in first approx.: semantics = message interchange at the interface
- open = environment absent/arbitrary

[2] no direct access to instance variables

Problems to tackle for an open f-a semantics

- "message passing"[2] framework ⇒ in first approx.:
  semantics = message interchange at the interface
- open = environment absent/arbitrary

Labels:

$$
\begin{array}{lcll}
\gamma & ::= & n\langle call\ o.m(\vec{v})\rangle \mid n\langle return(v)\rangle & \\
 & \mid & \langle spawn\ n\ of\ c(\vec{v})\rangle \mid \nu(n{:}T).\gamma & \text{basic labels} \\
a & ::= & \gamma? \mid \gamma! & \text{receive and send labels}
\end{array}
$$

[2]no direct access to instance variables

Problems to tackle for an open f-a semantics

- "message passing"[2] framework $\Rightarrow$ in first approx.: semantics = message interchange at the interface
- open = environment absent/arbitrary

$\Rightarrow$ does this mean: environment behavior arbitrary?

[2] no direct access to instance variables

Problems to tackle for an open f-a semantics

- "message passing"[2] framework ⇒ in first approx.: semantics = message interchange at the interface
- open = environment absent/arbitrary

⇒ does this mean: environment behavior arbitrary?

- well, depends . . . does "arbitrary trace" mean $\in Label^*$?

[2] no direct access to instance variables

Problems to tackle for an open f-a semantics

- "message passing"[2] framework $\Rightarrow$ in first approx.: semantics = message interchange at the interface
- open = environment absent/arbitrary

$\Rightarrow$ does this mean: environment behavior arbitrary?

- well, depends ... does "arbitrary trace" mean $\in Label^*$?
- we know $P + O$ is a program of the language
  - well-formed
  - well-typed
  - class-structured

[2] no direct access to instance variables

Problems to tackle for an open f-a semantics

- "message passing"[2] framework ⇒ in first approx.: semantics = message interchange at the interface
- open = environment absent/arbitrary

⇒ does this mean: environment behavior arbitrary?

- well, depends ... does "arbitrary trace" mean $\in Label^*$?
- we know $P + O$ is a program of the language
  - well-formed
  - well-typed
  - class-structured

environment is arbitrary but realizable

[2] no direct access to instance variables

Open semantics

- operational description:
- assumption/commitment formulation
- $Ass \vdash C : Comm \xrightarrow{a} \acute{Ass} \vdash \acute{C} : \acute{Comm}$
- interface: 3 orthogonal abstractions:
  - static abstraction: type system
  - abstraction of the stack structure of thread(s)
  - dynamic abstraction of the heap topology

Open semantics

- operational description:
- assumption/commitment formulation
- $Ass \vdash C : Comm \xrightarrow{a} \acute{Ass} \vdash \acute{C} : \acute{Comm}$
- interface: 3 orthogonal abstractions:
  - static abstraction: type system
  - abstraction of the stack structure of thread(s)
  - dynamic abstraction of the heap topology

As illustration, let us have a look at incoming calls. Basically, an incoming call can always arrive. But:

Is each incoming call realizable?

1. Static abstraction: type system

```
// component
public class P{
    public void m(C x){
        ...
    }
}
```

E.g.: Method m of o:P must have one parameter of type C.

$\leadsto$ Traces

$$\ldots n\langle call\ o.m(o')\rangle? \ldots$$

with $o, o' : P$ are not realizable.

2. Abstraction of the stack structure

E.g.:

- A thread must start its execution on the side of its thread class.
- Calls and returns of a thread must occur pairwise in a nested fashion.
- Each call returns to its caller.

$\leadsto$ Traces

$$\begin{array}{l} \ldots \\ n\langle call\ o.m(\ldots)\rangle? \\ n\langle call\ o'.m(\ldots)\rangle? \\ \ldots \end{array}$$

are not realizable.

3. Dynamic abstraction of the heap topology

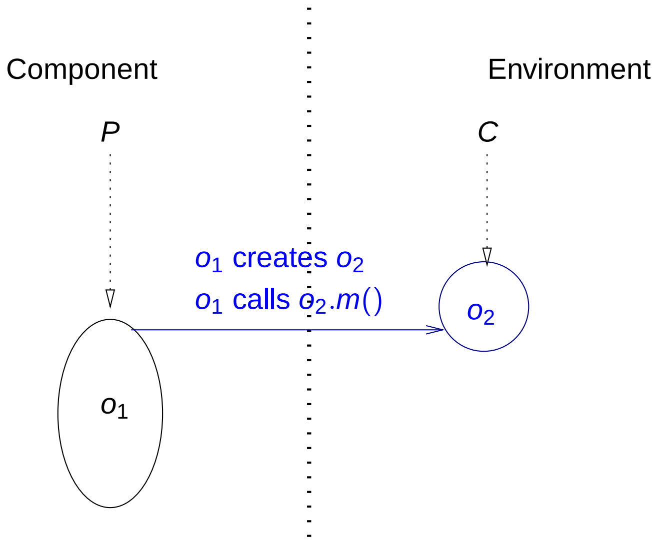
```
// component
public class P {
    ...
    public void m(){
        C x = new C();
        C y = x.m();
    }
}
```

Is a trace

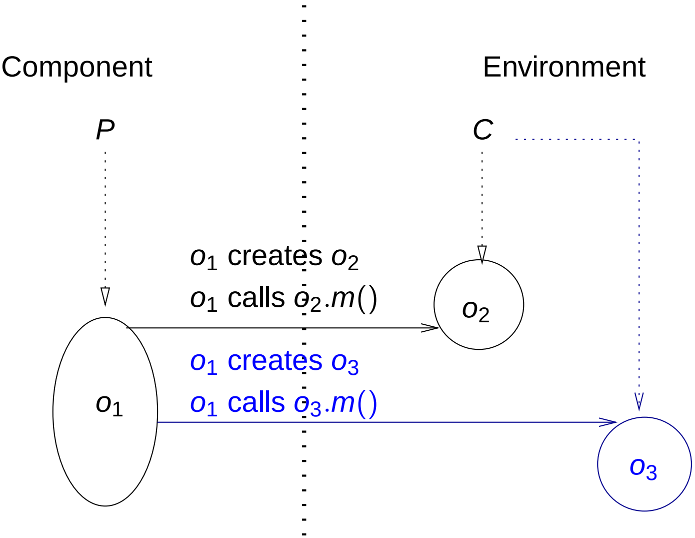
$$\begin{array}{l} \ldots \\ \nu(o_2 : C).n\langle call\ o_2.m()\rangle! \\ \nu(o_3 : C).n'\langle call\ o_3.m()\rangle! \\ n'\langle return(o_2)\rangle? \\ \ldots \end{array}$$

realizable?

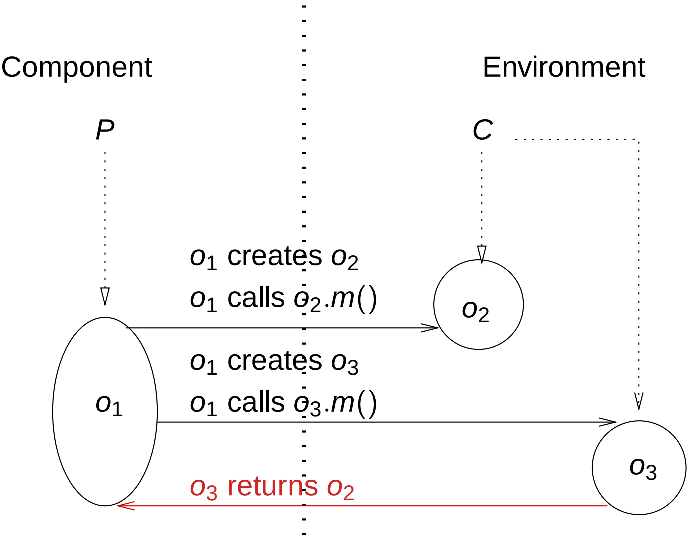
Dynamic heap abstraction example



Dynamic heap abstraction: example

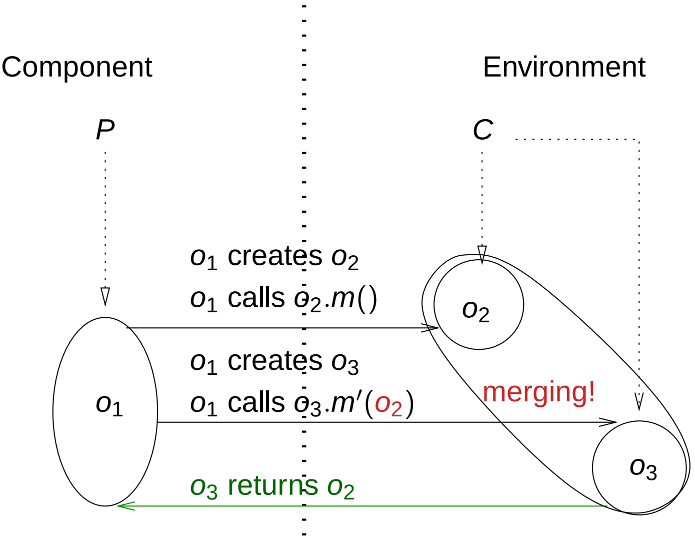


Dynamic heap abstraction: example

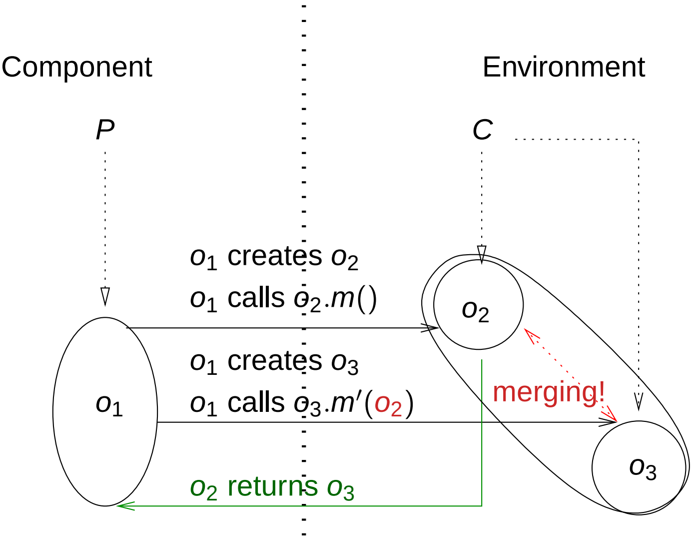


o_2 and o_3 cannot “know” each other!

Dynamic heap abstraction: example

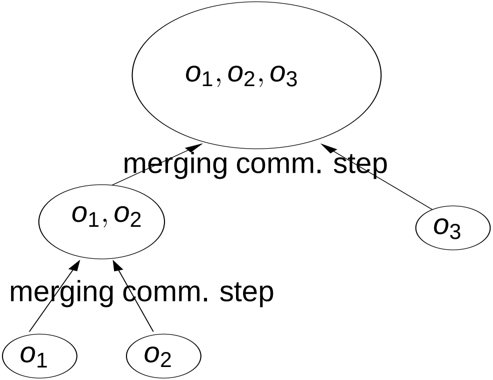


Dynamic heap abstraction: example



Dynamic aspects of cliques

- we have seen: cliques can merge
- assumption: names are never forgotten
  $\Rightarrow$ cliques never fall apart again
- clique evolution represents a tree:



Open semantics and heap abstraction

- exact interface behavior

$\Rightarrow$ abstraction of the heap topology necessary

- keep book about “who has been told what”:

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta$$

- assumption context: $E_\Delta \subseteq \Delta \times (\Delta + \Theta)$ = pairs of objects
- written $o_1 \hookrightarrow o_2$:
- worst case: equational theory implied by E_Δ (on Δ):

$$E_\Delta \vdash o_1 \leftrightharpoons o_2$$

(for $o_2 \in \Theta$: $E_\Delta \vdash o_1 \leftrightharpoons; \hookrightarrow o_2$)

Dynamic heap abstraction

- outgoing call
  - both caller and callee are known
  - $a = n\langle call\ o_{callee}.l(\vec{v})\rangle!$

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$$

  - update: $\acute{E}_\Delta = E_\Delta + \quad o_{callee} \hookrightarrow \vec{v}$

Dynamic heap abstraction

- outgoing call
  - both caller and callee are known
  - $a = n\langle call\ o_{callee}.l(\vec{v})\rangle!$

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$$

  - update: $\acute{E}_\Delta = E_\Delta + \quad o_{callee} \hookrightarrow \vec{v}$
- incoming call
  - only callee is known, caller is guessed
  - $a = n\langle call\ o_{callee}.l(\vec{v})\rangle?$

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$$

  - check:[3] $E_\Delta \vdash o_{caller} \hookrightarrow \vec{v}$

[3] actually, it's $\acute{E}_\Delta$ instead of E_Δ.

Simplified rule

$$\frac{\begin{array}{rl} & a = n\langle call\ o_r.l(\vec{v})\rangle? \\ \text{update contexts:} & \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + o_r \hookrightarrow \vec{v}, n \\ \text{check context:} & \acute{\Delta}; \acute{E}_\Delta \vdash o_s \leftrightharpoons\hookrightarrow \vec{v}, o_r : \acute{\Theta} \end{array}}{\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta} \textsc{CallI}$$

Where are we?

Open semantics in the presence of classes

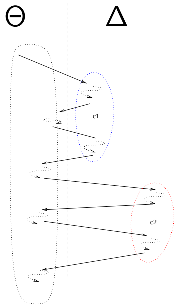
- static abstraction of type system
- abstraction of the stack structure
- abstraction of heap topology
- formalized in some “object calculus”

But we are still not ready...

Order of events

- separate observer cliques
- separate observer cliques cannot cooperate

⇒ order of interaction not globally observable[4]

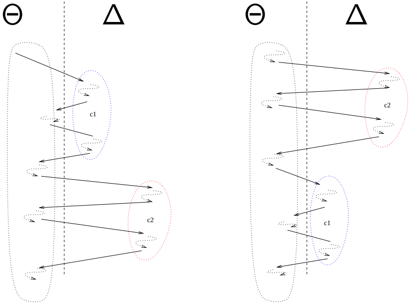


[4] Take care of merging

Order of events

- separate observer cliques
- separate observer cliques cannot cooperate

$\Rightarrow$ order of interaction not globally observable[4]

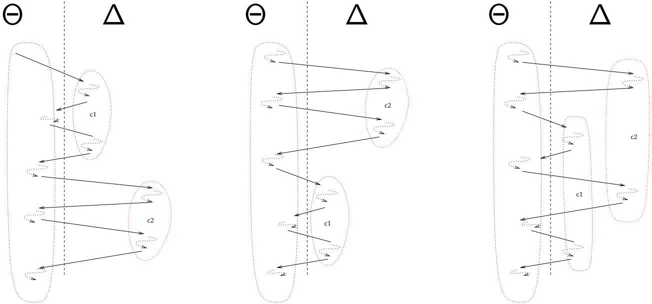


[4] Take care of merging

Order of events

- separate observer cliques
- separate observer cliques cannot cooperate

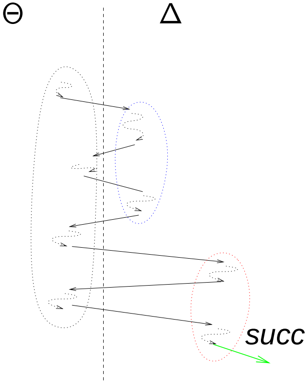
⇒ order of interaction not globally observable[4]



[4] Take care of merging

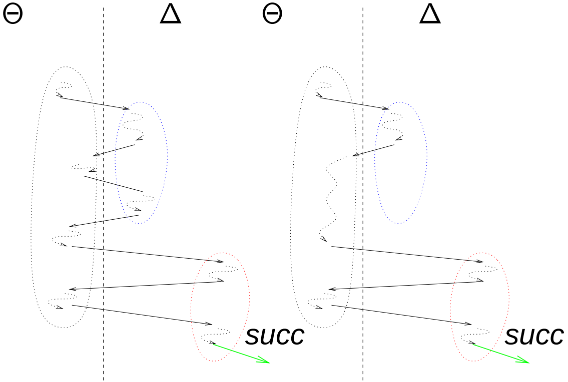
Order of events

- an observer reporting success, could additionally observe, that the interaction with the other clique is a prefix of the original, but not longer



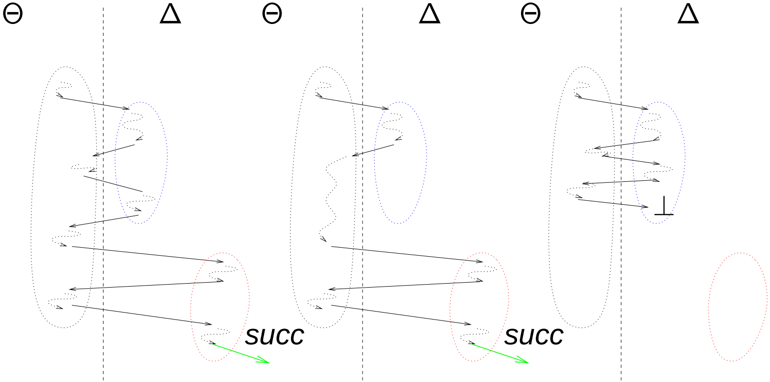
Order of events

- an observer reporting success, could additionally observe, that the interaction with the other clique is a prefix of the original, but not longer



Order of events

- an observer reporting success, could additionally observe, that the interaction with the other clique is a prefix of the original, but not longer



Trace semantics

Definition ($\sqsubseteq_{trace}$)

$\Xi_0 \vdash C_1 \sqsubseteq_{trace} C_2$, if the following holds. For all $\Xi_0 \vdash C_1 \stackrel{t}{\Longrightarrow}$ and all environment cliques $[o_t]$ after t, there exists $\Xi_0 \vdash C_2 \stackrel{s}{\Longrightarrow}$ such that

1. there exists an environment clique $[o_s]$ after s such that $\Xi_0 \vdash s \downarrow_{[o_s]} \asymp_\Delta t \downarrow_{[o_a]}$, and
2. $\Xi_0 \vdash t \precsim_\Delta s$.

- $\asymp_\Delta$: up-to swapping
- $\precsim_\Delta$: up-to swapping, replay, prefix

Completeness: line of argument

- goal: if $C_1 \sqsubseteq_{may} C_2$, then $C_1 \sqsubseteq_{trace} C_2$
- so, given a legal trace $s \in [\![C_1]\!]_{trace}$, do
  - construct a complementary context $\mathcal{C}_{\bar{s}}$
  - composition: program + context may do the observation

$$\mathcal{C}_{\bar{s}}[C_1] \longrightarrow^* success$$

  - observational equivalence: C_2 may do that, too:

$$\mathcal{C}_{\bar{s}}[C_2] \longrightarrow^* success$$

  - decomposition:[5] $s \in [\![C_2]\!]_{trace}$

⇒ problems for completeness (apart from technicalities)

1. definability ⇒ what are legal traces?
2. what can be observed/distinguished?

[5] That s is a trace of C_2 by decomposition is not a direct consequence. I ignore that here.

introduction

classes and observable behavior

consequences (closure conditions)

completeness

conclusion

Conclusions

- Fully abstract semantics for an
  - OO,
  - class-based,
  - multi-threaded (thread-classes)

  language.
- Abstractions:
  - type system
  - stack structure
  - heap topology
- Extensions:
  - monitors
  - subtyping (and subclassing), cloning, ...
  - (fully) compositional semantics

Results

- in the setting of = may-testing equivalence
  - exactly one kind of observation (e.g., “success”)
  - terminal i.e., not repeated observation

⇒ trace semantics gets weakened into a partial order semantics, relative to

  - dynamic cliques of connectivity of objects

- note: we don’t allow to observe (e.g.) divergence!
- note: if we allowed
  - different, repeated observations (for instance success-method + divergence), or
  - if we had a global shared variables (e.g., `stdout`)

  we are back in linear trace semantics

Results

Subject reduction: $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \stackrel{s}{\Longrightarrow} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$, then $\acute{\Delta} \vdash \acute{C} : \acute{\Theta}$. A fortiori: If $\Delta, \Sigma, \Theta \vdash n : T$, then $\acute{\Delta}, \acute{\Sigma}, \acute{\Theta} \vdash n : T$.

Soundness of connectivity abstraction: $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \stackrel{s}{\Longrightarrow} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$, then $\acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$.

No surprise $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \stackrel{a}{\to} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$, for incoming label a, then $\acute{\Delta}; \acute{E}_\Delta$ is a conservative extension of $\Delta; E_\Delta$. For outgoing steps, the situation is dual.

Soundness of legal trace system: If $\Delta_0; \vdash C : \Theta_0;$ and $\Delta_0; \vdash C : \Theta_0; \stackrel{t}{\Longrightarrow}$, then $\Delta_0 \vdash t : \mathit{trace}\, \Theta_0$.