Why would anyone care about this title?

Rémi Bonnet Philipp Haller

May 4, 2009

1 Introduction

The aim of this work is to provide an extension of the work presented in [1]. In this work, a typing system was proposed that used capabilities sets in order to infer properties regarding the aliasing state of objects. Unfortunately, it required specific constructs, namely the expose and localize statements, that had to be explicitly written by the user.

We will present here an alternate set of typing rules, shown in table 2.6 on page 5, that doesn't require these specific constructs. We will prove in section 4 on page 9 that any program written in the previous style can be simplified into a program that typecheck to the same value with the new rules. Section 7 on page 15 will then present an algorithm that is able to find typing proofs with these rules. Then we will show (what?)

2 Formalization

As in [1], we will consider that we have to typecheck a piece of code following the grammar presented in 2 on the next page. We will highlight a few restrictions of this syntax :

- There is no possibility of assignement to local variables, only to object fields.
- "while" and "for" constructs have been left over. Such constructs can be emulated by recursion, that we fully support.

Typechecking statements are written as $\Gamma \ ; \ \Delta \vdash t : T \ ; \ \Delta'$ where Γ is the context, Δ is the set of capabilities provided to the instruction and Δ' are the capabilities left over.

2.1 Tracked types and guarded types

Stuff about types.

Table 1: Our simplified grammar

Expr	:=	Variable
	—	"let" Variable "=" Expr "in" Expr
	—	"if" Expr "then" Expr "else" Expr
	—	"{" Expr ";" Expr ";" ... ";" Expr "}"
	—	MethodIdentifier(Expr, Expr, ... Expr)
	—	Expr.FieldIdentifier
	—	Expr.FieldIdentifier "=" Expr
	—	"new" ClassIdentifier(Expr, Expr, ... Expr)

2.2 Capabilities

Stuff about capabilities.

We distinguish two kinds of capabilities :

- full capabilities, denoted ρ. An instruction with such a capability is able to perform any (valid) operation on the variable.
- localized capabilitites, denoted $\rho \rightarrow \delta$. An instruction with such a capability can treat any variable of type $\rho \blacktriangleright C$ as a variable of type $\delta \triangleright C$. As a special case, we define $\rho \rightarrow \rho$ as equal to ρ

We will sometimes also refer to the 'null' capability, that is always present and has no effect.

To ensure that we distinguish correctly distinct capabilities, we define labels for any new construct or method invocation. For one new construct at a specific code position p, there is a single guard $\rho_{new,p}$ that will be the capability associated to the new object. In the same way, we define for every method invocation at code position p an infinity of guards $\{\rho_{m,p,i}\}_i$ that will be used to map all possible fresh guards returned by the method body.

2.3 Method signatures and annotations

The (uniqueness) semantics of a method are provided by the typing result of the method body. Three specific things are of interest in such a type statement :

- The existence of a typing statement means that the method body doesn't violate any aliasing protection described in 2.1 on the preceding page.
- The difference between Δ and Δ' tells us which variables have been consumed by the method body.
- The type of T gives us the uniqueness status of the returned value.

We use annotations to the method header in order to get this method typing statement. Three annotations are possible :

- @unique (stuff)
- @transient (stuff)
- @exposed (stuff)

... and also possible constraints on guards (equality between parameters guard, freshness...)

From the typing statement of the method body, we will then get the method type. This method type is of the form $\forall\Delta^*.\ (\Delta, \overline{F}) \to (\Delta', H)$. The vector $\overline{F}$ contains the parameter types, which were described in Γ. Δ and Δ' corresponds to the same sets in the typing statement with a specific change : if the typing statement required a capability ρ that was the guard of no tracked parameters (only of variables with types $\rho \triangleright C$ – @exposed parameters), then the method type doesn't require this capability. We will explain this (seemingly arbitrary) special case below. Finally, the set Δ^* is the set of polymorphic guards : guards that can be instantiated to any value for method application.

The validity of this type provided by the user can be checked by the typing algorithm presented later. The typing algorithm may even find a more precise type (for example, reduce the number of variables that are consumed). Although, because of recursivity, we need the user to provide a valid type to start from.

We also accept the possibility of "built-in" methods, that are provided with their signatures directly by the compiler. However, such methods must satisfy the following properties (that hold for all method types built from a method body) :

Properties 2.1
- *If a method returns a tracked (resp. guarded) reference with a polymorphic guard, then this method also required a tracked (resp. guarded or tracked) reference of the same guard as parameter.*
- *If a method requires a capability ρ, then there is a tracked parameter with this guard.*

2.4 Polymorphism of method types

Now, the question is how a method with a given type can be applied to a specific set of typed expressions. We have mentionned before that a method type contains a set Δ^* of polymorphic guards.

(stuff...)

2.5 Merging of capabilities

At the end of a branching construct like the if one, we have to determine a set of capabilities that is in some sense the "intersection" of the output capabilities of the two branches.

Unfortunately, the simple intersection doesn't work. For example, if $\Delta_1 = \{\rho\}$ and $\Delta_2 = \{\rho \to \delta\}$, then the resulting capability set must be $\Delta = \{\rho \to \delta\}$,

as localized capabilities are intuitively restricted version of the full capabilities. We will formalize this more precisely in section 3.1.

(stuff...)

2.6 Typing rules

We will now present in table 2.6 on the next page our typing rules. We can distinguish three kind of rules.

- Rules prefixed by `C-` or `S-` are syntatic rules : they give the type of a syntaxic construct from the types of the subexpressions.
- The rule `T-LOC` is the counterpart of the `LOCALIZE` rule of [1]. It allows to start the localization of guard ρ into δ. As long as $\rho \rightarrow \delta$ will be present in the capabilities set, the rule `W-LOC` will allow to treat expressions of type $\rho \blacktriangleright C$ as expressions of type $\delta \triangleright C$
- Rules `W-LOC` and `W-EXP` are the exposition rules. A tracked reference can always be turned into a simply guarded reference, either to its own guard (with `W-EXP`) or to any guard δ such that the associated localization capability is present (with `W-LOC`)

A few remarks must be written regarding rule `S-INVK` :

- We have explained before that the guard of @exposed parameters are not required as input capabilities of the method type. The rule `S-INVK` explains why. If two different capabilities are required, then even if the method type is polymorphic with respect to guards, two different capabilities must be provided. @exposed parameters are less demanding : we require the capability to be present for every parameter provided (this is the $T_i = \rho_i\{\blacktriangleright, \triangleright\}C_i \implies \rho_i \in \Delta_{i+1}$ premisce of rule `S-INVK`), but we don't require them to be distinct. This is due to lemma 4.1 on page 9 that explains that for method with @exposed annotations, those accepting parameters with distinct guards also accept parameters with the same guard.
- TO REWRITE.

Finally, we must note that these rules are not completely syntax directed, so we will have to show some additional properties before being able to present the actual typechecking algorithm, in section 7.

3 Generic properties of the type system

3.1 Order on capabilities sets

Definition 3.1 *We will say that* (T_1, Δ_1) *dominates* (T_2, Δ_2) *and we write* $(T_2, \Delta_2) \preceq (T_1, \Delta_1)$ *iff:*

Table 2: Typing Rules

$$\frac{\Gamma\ ;\ \Delta \vdash t : \rho \blacktriangleright C\ ;\ \Delta' \otimes \rho}{\Gamma\ ;\ \Delta \vdash t : \delta \triangleright C\ ;\ \Delta'\{\rho \to \delta\}} \quad \text{(T-LOC)}$$

$$\frac{\Gamma\ ;\ \Delta \vdash t : \rho \blacktriangleright C\ ;\ \Delta' \qquad \rho \to \delta \in \Delta'}{\Gamma\ ;\ \Delta \vdash t : \delta \triangleright C\ ;\ \Delta'} \quad \text{(W-LOC)}$$

$$\frac{\Gamma\ ;\ \Delta \vdash t : \rho \blacktriangleright C\ ;\ \Delta'}{\Gamma\ ;\ \Delta \vdash x : \rho \triangleright C\ ;\ \Delta'} \quad \text{(W-EXP)}$$

$$\frac{\forall i.\Gamma\ ;\ \Delta_{i-1} \vdash t_i : T_i, \Delta_i}{\Gamma\ ;\ \Delta_0 \vdash t_1\ ;\ t_2\ ;\ t_3\ ...\ t_n : T_n, \Delta_n} \quad \text{(C-SEQ)}$$

$$\frac{\begin{array}{c}\Gamma\ ;\ \Delta_0 \vdash t_0 : \mathit{bool}, \Delta_1 \\ \Gamma\ ;\ \Delta_1 \vdash t_1 : T, \Delta_2 \\ \Gamma\ ;\ \Delta_1 \vdash t_2 : T, \Delta_3\end{array}}{\Gamma\ ;\ \Delta_0 \vdash \text{if } t_0 \text{ then } t_1 \text{ else } t_2 : T, \mathit{gcd}(\Delta_2, \Delta_3)} \quad \text{(C-IF)}$$

$$\frac{\begin{array}{c}\Gamma\ ;\ \Delta_0 \vdash t_1 : T_1, \Delta_1 \\ \Gamma \cup \{x : T_1\}\ ;\ \Delta_1 \vdash t_2 : T_2, \Delta_2\end{array}}{\Gamma\ ;\ \Delta_0 \vdash \text{let } x = t_1 \text{ in } t_2 : T_2, \Delta_2} \quad \text{(C-LET)}$$

$$\frac{x : T \in \Gamma}{\Gamma\ ;\ \Delta \vdash x : T\ ;\ \Delta} \quad \text{(S-VAR)}$$

$$\frac{\begin{array}{c}\Gamma\ ;\ \Delta \vdash t : \rho \triangleright C\ ;\ \Delta' \otimes \rho \\ \mathit{fields}(C) = \overline{l : D}\end{array}}{\Gamma\ ;\ \Delta \vdash t.l : \rho \triangleright D_i\ ;\ \Delta' \otimes \rho} \quad \text{(S-SEL)}$$

$$\frac{\begin{array}{c}\Gamma\ ;\ \Delta \vdash t_1 : \rho \triangleright C\ ;\ \Delta' \\ l : D \in \mathit{fields}(C) \\ \Gamma\ ;\ \Delta' \vdash t_2 : \rho \triangleright D\ ;\ \Delta'' \otimes \rho\end{array}}{\Gamma\ ;\ \Delta' \vdash t_1.l = t_2 : \rho \triangleright D\ ;\ \Delta'' \otimes \rho} \quad \text{(S-ASSIGN)}$$

$$\frac{\begin{array}{c}\forall i \in \{1..n\}.\Gamma\ ;\ \Delta_i \vdash t_i : \rho_i \blacktriangleright D_i; \Delta_{i+1} \otimes \rho_i \\ \mathit{fields}(C) = \overline{l : D} \qquad \rho = l_{new}\end{array}}{\Gamma\ ;\ \Delta_1 \vdash \text{new } C(\overline{t}) : \rho \blacktriangleright C\ ;\ \Delta_{n+1} \otimes \rho} \quad \text{(S-NEW)}$$

$$\frac{\begin{array}{c}\forall i \in \{1..n\}.\Gamma\ ;\ \Delta_i \vdash t_i : T_i\ ;\ \Delta_{i+1} \\ T_i = \rho_i\{\blacktriangleright, \triangleright\}C_i \implies \rho_i \in \Delta_{i+1} \\ \mathit{mtype}(m) = \forall \Delta^*.\ \left(\Delta', \overline{F}\right) \to (\Delta'', C) \\ \sigma = \mathit{unify}\left(\overline{T}, \overline{F}, \Delta^*\right) \\ \Delta_{n+1} = \sigma\Delta' \otimes \Delta_r \qquad \Delta''' = \sigma\Delta'' \otimes \Delta_r\end{array}}{\Gamma\ ;\ \Delta \vdash m\left(\overline{t}\right) \sigma C\ ;\ \Delta'''} \quad \text{(S-INVK)}$$

- *if* T_2 *is tracked, then* T_1 *is tracked with the same guard.*
- *if* $T_2 = \delta \triangleright C$*, then* $T_1 = \rho \blacktriangleright C$ *and* $\rho \to \delta \in \Delta_1 \vee \rho \in \Delta_1 \vee \delta = \rho$
- $\forall \rho \in \Delta_2.\ \rho \in \Delta_1$
- $\forall \rho \to \delta \in \Delta_2.\ \rho \to \delta \in \Delta_1 \vee \rho \in \Delta_1$

We will often compare only capabilities sets. To this aim, we also define $\Delta_2 \preceq \Delta_1 \Leftrightarrow (Unit, \Delta_2) \preceq \Delta_1$.

This relation is an order where greater elements are in some sense more precise typing results. The idea that greater elements are "better" typing result is provided by the following results :

Lemma 3.1 *Let* $\Gamma\ ;\ \Delta_1 \vdash t : T_1\ ;\ \Delta_1'$ *a valid typing statement. If there exists* $\Delta_2 \preceq \Delta_1$ *then there exists* $(T_2, \Delta_2') \preceq (T_1, \Delta_1')$ *with respect to* $(\Gamma, \emptyset)$ *such that* $\Gamma\ ;\ \Delta_2 \vdash t : T_2\ ;\ \Delta_2'$ *is valid.*

Lemma 3.2 *Let* $\{S_0, S_1, S_2\}$ *be three sequents of the form* $S_i := \Gamma_i; \Delta_i \vdash t_i : T_i, \Delta_i'$*. If :*

- S_1 *is an immediate premisce of* S_0
- $\Gamma_1 = \Gamma_2$ *and* $\Delta_1 = \Delta_2$
- $(T_1, \Delta_1') \preceq (T_2, \Delta_2')$

Then, there exists (T_f, Δ_f') *that dominates* (T_0, Δ_0') *such that* $\Gamma_0, \Delta_0 \vdash t_0 : T_f, \Delta_f$ *can be immediately proven from the same rule as* S_0 *and with* S_2 *as an immediate premisce.*

Lemma 3.3 *(weakening of typing result) Let* $S := \Gamma\ ;\ \Delta \vdash t : T_1\ ;\ \Delta_1$ *be a valid sequent. If* (T_1, Δ_1) *dominates* (T_2, Δ_2)*, then there exists* Δ_2' *dominating* Δ_2 *and* $\Gamma\ ;\ \Delta \vdash t : T_2\ ;\ \Delta_2'$ *is valid.*

With this order, we can prove that for any two pairs of type and capability set, there exists a greatest common dominated set. We will use this set as a way to merge if branches.

Definition 3.2 *We define* $gcd(\Delta_1, \Delta_2)$ *as :*

$$\begin{array}{rcl} \forall \rho.\ \rho \in gcd(\Delta_1, \Delta_2) & \Leftrightarrow & \rho \in \Delta_1 \wedge \rho \in \Delta_2 \\ \forall \rho, \delta.\ \rho \to \delta \in \mathrm{gcd}(\Delta_1, \Delta_2) & \Leftrightarrow & \left\{ \begin{array}{l} \rho \in \Delta_1 \vee \rho \to \delta \in \Delta_1 \\ \rho \in \Delta_2 \vee \rho \to \delta \in \Delta_2 \end{array} \right. \end{array}$$

It is easy to show that this definition corresponds to the greatest common dominated set.

3.2 Propagation of types and capabilities

Lemma 3.4 *(inheritance and unicity of tracked type) If t is a non-terminal term which fulfills $S := \Gamma\ ;\ \Delta_1 \vdash t : \rho \blacktriangleright C\ ;\ \Delta_2'$, then either ρ is a fresh guard with respect to (Γ, Δ) or there is a subterm of t that is shown to be of type $\rho \blacktriangleright C'$ in the proof of S.*

Moreover, ρ is unique : there is no $\Delta_2, \Delta_2', \rho_2 \neq \rho_1$ such that $\Gamma; \Delta_2 \vdash t : \rho_2 \blacktriangleright C'; \Delta_2'$, is valid.

We will here simply state a few (easy) useful properties for future reference.

Properties 3.5 *(propagation of localized capabilities) Let $S := \Gamma\ ;\ \Delta \vdash t : T\ ;\ \Delta'$ be a valid sequent. If $\rho \to \delta \in \Delta$, then :*

- $\rho \to \delta \in \Delta'$
- *For any sequent $S_2 := \Gamma_2\ ;\ \Delta_2 \vdash t : T\ ;\ \Delta_2'$ in the proof of S, $\rho \to \delta \in \Delta_2$*

Properties 3.6 *Let $\Gamma\ ;\ \Delta \vdash t : T\ ;\ \Delta'$ be valid. Then for any Δ_r with $\Delta_r \cap \Delta = \emptyset$:*

$$\Gamma\ ;\ \Delta \otimes \Delta_r \vdash t : T\ ;\ \Delta' \otimes \Delta_r \text{ valid}$$

3.3 Syntaxic equivalence

Lemma 3.7 *(encoding of sequences) For any $t1$, $t2$, $t3$, T, Gamma, Delta, Delta', we have :*

$$\Gamma\ ;\ \Delta \vdash t1\ ;\ (t2\ ;\ t3) : T\ ;\ \Delta' \text{ valid} \quad \Leftrightarrow \quad \Gamma\ ;\ \Delta \vdash t1\ ;\ t2\ ;\ t3 : T\ ;\ \Delta' \text{ valid}$$
$$\Gamma\ ;\ \Delta \vdash t1\ ;\ t2 : T\ ;\ \Delta' \text{ valid} \quad \Leftrightarrow \quad \Gamma\ ;\ \Delta \vdash \mathit{let}\ \_ = t1\ \mathit{in}\ t2 : T\ ;\ \Delta' \text{ valid}$$

This result feels very natural : it means that, as in lambda-calculus, we can encode any arbitrary sequence of instructions using only let constructs.

As we will see in next section, the erasure of EL programs leaves some strange let construction that we may want to suppress. Actually, it also feels natural that we should be able to replace a let construction assigning a variable to another one by a straight substitution. Although this result seems intutive, the typing proof for the two expressions can be quite different. Expecially, it is possible in the first form to use a single instance of `T-LOC` to change the type of the variable, while the second expression requires multiples instances of these rules. Thus we will prove here this result :

Theorem 1 *(let-suppression) If $\Gamma; \Delta \vdash letx = yint : T; \Delta'$ is valid in CT with $y \in Var$, then $\Gamma; \Delta \vdash [x := y]t : T'; \Delta''$ is valid in CT with $\Delta' \preceq \Delta''$.*

proof : First, let us consider the proof of $S_0 := \Gamma\ ;\ \Delta_0 \vdash letx = yint : T\ ;\ \Delta_2$. Immediate premisces must be the two sequents $S_1 := \Gamma\ ;\ \Delta_0 \vdash y : T_1\ ;\ \Delta_1$ and $S_2 := \Gamma, x : T_1\ ;\ \Delta_1 \vdash t : T\ ;\ \Delta_2$. We know that there exists such a proof of S_0 such that S_1 is proven from `S-VAR` with only rules `T-LOC`, `W-LOC` and `W-EXP`. As at most one possible application of these rules is possible, we have four possible cases:

- S_1 is directly inducted from `S-VAR`, which means $y : T_1 \in \Gamma$. Then if we consider the proof of S_2, at every place where the subterm x is typed, then it is given the type T_1 (from `S-VAR`). Thus, if we type $[x := y]t$ under (Γ, Δ), then the exact same proof can be used with the subterms y getting the type T_1 directly from `S-VAR`.

- S_1 is induced from `W-EXP`. Then we have $T_1 = \rho \triangleright C$ and $y : \rho \blacktriangleright C \in \Gamma$. Then, again, if we consider the proof of S_2, at every place where x is typed, then it is given the type $\rho \triangleright C$. Thus, if we type $[x := y]t$ under (Γ, Δ), the same proof applies, where the subterms y are getting the type T_1 from `S-EXP` with the premisce $y : \rho \blacktriangleright C$ being obtained from `S-VAR`

- S_1 is induced from `W-LOC`, which means we have $T_1 : \delta \triangleright C$ and $\rho \rightarrow \delta \in \Delta_1$. Then by the lemma 3.5 on the previous page, if we reuse the proof of S_2, the only work that is left to do is to prove the sequents $\Gamma_s \ ; \ \Delta_s \vdash y : T_1 \ ; \ \Delta'_s$. Thanks to lemma 3.5 on the preceding page, we know that for all these sequents, $\rho \rightarrow \delta \in \Delta_s$, which allows us to prove this sequent by application of `W-LOC`

- S_1 is induced from `T-LOC`. We can then notice that the typechecking statement $S_1 := \Gamma \ ; \ \Delta \vdash y : \delta \triangleright C \ ; \ \Delta \otimes \{\rho \rightarrow \delta\}$ is dominated by the typechecking statement $S_1^* := \Gamma \ ; \ \Delta_0 \vdash y : \rho \blacktriangleright C \ ; \ \Delta' \otimes \rho$ which is the direct application of `S-VAR`. That means there is a proof of a dominated version of S_0 where the type of y is decided from S_1^* (lemma 3.2), and thus that we can use our previous demonstration in the case of `S-VAR` to show that there exists (T', Δ'') dominating (T, Δ') such that $\Gamma \ ; \ \Delta \vdash [x := y]t : T' \ ; \ \Delta''$ holds. Thanks to lemma 3.3, we can even show that there exists Δ''' dominating Δ' such that $\Gamma \ ; \ \Delta \vdash [x := y]t : T \ ; \ \Delta'''$ holds, which concludes the demonstration.

Interestingly, this shows that the suppression of trivial let constructions doesn't give exactly the same typing result (but always a dominating one). This is due to the fact that x may not be present in t, and thus that we lose opportunities to weaken our capability sets. However, if we force x to be present in t, then we could show that we can get the exact same typing results.

We can get a similar (and easy-to-prove) result if the variable being defined appears only once :

Lemma 3.8 *(weak let-suppression) If x appears only once inside $t2$, then the following two type judgements are equivalent :*

- $\Gamma \ ; \ \Delta \vdash [x := t1]t2 : T \ ; \ \Delta'$
- $\Gamma \ ; \ \Delta \vdash letx = t1 int2 : T \ ; \ \Delta'$

We will state a similar result for methods themselves.

Theorem 2 *(method-substitution) Let m be a method with actuals $\{x_i : T_i\}_i$ and body body(m). Then, if $\{y_i\}$ are variables, we have the following two typing judgements being equivalent :*

1. $\Gamma\ ;\ \Delta \vdash m(y_1, y_2, ...y_n) : T\ ;\ \Delta'$
2. $\Gamma\ ;\ \Delta \vdash [\forall i.\ x_i := y_i]body(m) : T\ ;\ \Delta'$

4 Erasure of programs writen in EL style

In this section, we will attempt to show that any program that could typecheck under the old system still typechecks. For conciseness, we will name EL the type system of [1] and CT the one presented here.

Definition 4.1 *If t is a term writen in EL grammar, we will define the erasure of t, written $\underline{t}$ the term produced by removing all instances of expose and localize according to the following rules :*

- *expose $x = y$ in $t \rightarrow$ let $x = y$ in t*
- *localize(t1, t2) $\rightarrow$ let z = t1 in (t2 ; z)*

One of the major changes between EL and CT is that there is no real 'exposition' : with rule `W-EXP`, we use same capability to refer both to the tracked version of an object, and to its exposed version. So, before proving the real result, we will show this interesting property of the CT system :

Lemma 4.1 *Let ρ, δ be two guards and $S := \Gamma\ ;\ \Delta \otimes \rho \otimes \delta \vdash t : T\ ;\ \Delta' \otimes \rho \otimes \delta$ be valid under CT. Then, if all variables inside Γ with guard δ are not tracked, then $S' := [\delta := \rho]\Gamma\ ;\ \Delta \otimes \rho \vdash t : [\delta := \rho]T\ ;\ \Delta' \otimes \rho$ is also valid under CT.*

proof : By induction on the proof of S. The core of the proof is based on the fact that if a method requires only a guarded type, then it only asks the capabilities to be present and doesn't care about duplication (see properties 2.1 on page 3)

Theorem 3 *Let t be a term. If, under EL $S := \Gamma\ ;\ \Delta \vdash t : T\ ;\ \Delta'$ is valid, then $\underline{S} := \Gamma\ ;\ \Delta \vdash \underline{t} : T\ ;\ \Delta''$ is valid under CT with $\Delta' \subset \Delta''$.*

proof : We will work on induction on the proof of S to build a proof of $\underline{S}$

- case `ASSIGN`:

  Under EL, $\Gamma\ ;\ \Delta \vdash t1.l = t2 : \rho \triangleright D\ ;\ \Delta'' \otimes \rho$ is valid, and thus all its prerequisites also hold. By induction hypothesis, we have $\Gamma\ ;\ \Delta \vdash \underline{t1} : \rho \triangleright C\ ;\ \Delta' \otimes \Delta'_r$ and $\Gamma\ ;\ \Delta' \vdash \underline{t2} : \rho \triangleright D\ ;\ \Delta'' \otimes \Delta''_r$. By lemma 3.6, we have $\Gamma\ ;\ \Delta' \otimes \Delta'_r \vdash \underline{t2} : \rho \triangleright D\ ;\ \Delta'' \otimes \Delta'_r \otimes \Delta''_r$ valid, which means that $\Gamma\ ;\ \Delta \vdash t1.l = t2 : \rho \triangleright D\ ;\ \Delta' \otimes \Delta_r$ is valid.

  The exact same idea of demonstration works for `SEL`, `NEW` and `INVK`. The case of `VAR` being immediate, we will move to the two interesting cases, being `EXPOSE` and `LOCALIZE`

- case `EXPOSE`:

  We have that $\Gamma\ ;\ \Delta \vdash \text{expose } x = t_1 \text{ in } t_2 : T\ ;\ \Delta'' \otimes \rho$ is valid under EL. Thus, the following statement hold under EL:

  - $\Gamma\ ;\ \Delta \vdash t_1 : \rho \blacktriangleright C\ ;\ \Delta' \otimes \rho$
  - ρ' fresh
  - $\Gamma, x : \rho' \triangleright C\ ;\ \Delta' \otimes \rho, \rho' \vdash t_2 : T\ ;\ \Delta'' \otimes \rho, \rho'$

  By induction hypothesis, we get under CT :

  - $\Gamma\ ;\ \Delta \vdash \underline{t_1} : \rho \blacktriangleright C\ ;\ \Delta' \otimes \rho \otimes \Delta_{r1}$
  - $\Gamma, x : \rho' \triangleright C\ ;\ \Delta' \otimes \rho, \rho' \vdash \underline{t_2} : T\ ;\ \Delta'' \otimes \Delta_{r2} \otimes \rho, \rho'$

  And with lemma 4.1 on the preceding page, we now have :

  $$\Gamma, x : \rho \triangleright C\ ;\ \Delta' \otimes \rho \vdash \underline{t_2} : T\ ;\ \Delta'' \otimes \rho$$

  Which allows us to apply `C-LET` and with the help of lemma 3.6 we get :

  $$\Gamma\ ;\ \Delta \vdash \text{let } x = \underline{t_1} \text{ in } \underline{t_2} : T\ ;\ \Delta'' \otimes \Delta_{r1} \otimes \Delta_{r2} \otimes \rho$$

  That can be rewritten in :

  $$\Gamma\ ;\ \Delta \vdash \underline{\text{expose } x = t_1 \text{ in } t_2} : T\ ;\ \Delta'' \otimes \rho \otimes \Delta_r$$

  ... which ends this induction case.

- case `LOCALIZE`

  We have that $\Gamma\ ;\ \Delta \vdash \mathit{localize}(t_1, t_2) : \rho_2 \triangleright C\ ;\ \Delta''$ under EL, which means the following statement hold under EL :

  - $\Gamma\ ;\ \Delta \vdash t_1 : \rho_1 \blacktriangleright C\ ;\ \Delta' \otimes \rho_1$
  - $\Gamma\ ;\ \Delta' \vdash t_2 : \rho_2 \triangleright C'\ ;\ \Delta''$

  By induction hypothesis, their erasure counterparts are valid in CT :

  - $\Gamma\ ;\ \Delta \vdash \underline{t_1} : \rho_1 \blacktriangleright C\ ;\ \Delta' \otimes \rho_1 \otimes \Delta_{r1}$
  - $\Gamma\ ;\ \Delta' \vdash \underline{t_2} : \rho_2 \triangleright C'\ ;\ \Delta'' \otimes \Delta_{r2}$

  By `T-LOC` in CT, we then get :

  $$\Gamma\ ;\ \Delta \vdash \underline{t_1} : \rho_2 \triangleright C\ ;\ \Delta' \otimes \rho_1 \to \rho_2 \otimes \Delta_{r1}$$

  As y_2 is fresh, we can get through `C-SEQ` :

$$\Gamma, y_2 : \rho_2 \triangleright C \ ; \ \Delta' \vdash \{\underline{t2}; y_2\} : \rho_2 \triangleright C \ ; \ \Delta'' \otimes \rho_1 \rightarrow \rho_2 \otimes \Delta_{r2}$$

And by `C-LET` :

$$\Gamma \ ; \ \Delta \vdash \text{let } y_2 = \underline{t_1} \text{ in } \{t_2; y_2\} : \rho_2 \triangleright C \ ; \ \Delta'' \otimes \Delta_{r1} \otimes \Delta_{r2} \otimes \rho_1 \rightarrow \rho_2$$

... which concludes this case and achieves the demonstration.

5 Reducing CT to EL

In this section, we will show that the typing system CT provides the same guarantees as EL. To do this, we will have to look at specific executions of CT programs, leading us to introduce some rough program formalization.

5.1 From CT programs to terms

Definition 5.1 *We define a program $\mathcal{P}$ as a collection of class and method definitions : $\mathcal{P} = (\{C_i\}, \{m_i\})$.*

An execution $\mathcal{E}$ of the program with parameters $\{x_i\}$ is the reduction of the term $m_0(x_1, x_2, ...x_n)$ according to the usual reduction rules where non-builtin method calls are handled according to the theorem 2 on page 8.

After having defined the usual interpretation of all syntaxic constructs, we define the semantics *of a program as, for any possible execution, the final value returned by the program and the sequence of memory states that the program goes in, where memory is defined as the space for only objects (and not for any local variables).*

Definition 5.2 *For a given program $\mathcal{P}$, we define $\mathcal{P}^* = (\{C_i\}, \{m_{i,j,k}\})$ the program with an infinite number of methods where the $m_{i,j,k}$ are defined as copies of the original m_i where every call to a method m_p inside body$(m_{i,j,k})$ is substitued by a call to $m_{p,j+1,q}$ where q is a unique index inside m_i, j, k*

It is easy to see that $\mathcal{P}$ and $\mathcal{P}^*$ have exactly the same semantics. The interest of $\mathcal{P}^*$ is that during any execution of it, each physical term is only reduced once, leading to this definition:

Definition 5.3 *For an execution $\mathcal{E}$ of $\mathcal{P}$, we denote $\mathcal{P}_{|\mathcal{E}}$ the restriction of $\mathcal{P}$ to $\mathcal{E}$, defined by recursively eliminating all if branches and methods that were not visited during the execution.*

$\mathcal{P}_{|\mathcal{E}}$ presents some interesting properties : it has no more if constructs, has the same semantics and types as $\mathcal{P}$ for the execution $\mathcal{E}$, and if $\mathcal{E}$ terminates, then it is finite.

Interestingly enough, EL guarantees are only safety properties : they are true if and only if they are true for every finite prefix of an execution chain [2]. Thus, we will only look at such traces.

Definition 5.4 *We denote by $\mathcal{P}_{|\mathcal{E},p}$ the program $\mathcal{P}$ restricted to $\mathcal{E}$ and truncated to p the alteration of $\mathcal{P}_{|\mathcal{E}}$ where all method calls to $m_{i,j,k}$ with $j > p$ are replaced by calls to a $m_{error,mtype(m_{i,j,k})}$ methods that have the same type as $m_{i,j,k}$ but that have undefined behaviour.*

Now, by theorem 2 on page 8, we know that we can replace such a program by a single term, with the same semantics and the same type.

Thus, for any finite prefix of an execution chain of a program, we can exhibit a term, with no if constructs, that typechecks to the same value as the return value of m_0 and that exhibits the same behavior as the original program during the first p (with p arbitrary large) method calls.

Now, we have a term with only let constructs and atomic operations (field assignement, field selection, new object and m_{error}). Thanks to lemma 3.8 on page 8, if this term has p free variables $\{x_i\}_{1 \leq i \leq p}$, we can easily turn this term into the following normal form (administrative normal form, as described in [3]), which has same type and semantics :

$$
\begin{array}{ll}
 & \textbf{let } x_{p+1} = f_{p+1}(x_{k_{p+1,1}}, x_{k_{p+1,2}}, ..., x_{k_{1,q_{p+1}}}) \\
\textbf{in} & \textbf{let } x_{p+2} = f_{p+2}(x_{k_{P+2,1}}, x_{k_{p+2,2}}, ..., x_{k_{2,q_{p+2}}}) \\
 & \textbf{...} \\
\textbf{in} & \textbf{let } x_{n-1} = f_{p+2}(x_{k_{n-1,1}}, x_{k_{n-1,2}}, ..., x_{k_{2,q_{n-1}}}) \\
\textbf{in} & f_n(x_{k_{n,1}}, x_{k_{n,2}}, ..., x_{k_{n,q_n}})
\end{array}
$$

5.2 Enriching EL

Unfortunately, the standard EL syntax as described in [1] is not sufficient for CT terms to be translated to it. That's why we need to propose two extensions.

¡description of fixed-size tuples and multi-let¿

It is quite apparent that such an extension of EL doesn't violate any of the announced guarantees. We will skip here the demonstration, moving to the next extension:

Definition 5.5 *For every guard δ and class C, we define the method $cast_{\delta \triangleright C}$ of type $\forall\{\rho\}.(\emptyset, \{x : \rho \blacktriangleright C\}) \rightarrow (\emptyset, \delta \triangleright C)$*

We denote as ELC the EL type system that admits these methods as built-ins while giving them the semantics of identity.

Of course, a program that typechecks under ELC doesn't provide the guarantees of EL any more, but we can relate the two systems thanks to this lemma :

Lemma 5.1 *If y is a variable, the following two terms have the same type and semantics under EL(C):*

- $exposex = yint$
- $[x := cast_{\delta \triangleright C}(x)]t$

5.3 Adding back localize statements

Before having full-fledged EL(C) terms, we will define an intermediate form of program :

Definition 5.6 *Let CTWL be the system defined by :*

- *For the syntax, by the grammar of CT described in table 2 on page 2, with the additionnal localize construct.*
- *For the type system, by the rules of CT, with* `T-LOC` *and* `W-LOC` *replaced by the* `LOCALIZE` *of EL.*

We say that t' of CTWL is a standard localized form of t in CT if it can be obtained by the applications of the following rewriting rule :

$$\begin{array}{l} \ldots \\ let x_p = f_p(x_{k_1}, \ldots . x_{k_q}) in \\ \ldots \end{array} \rightarrow \begin{array}{ll} \ldots & \\ let x_p = f_p(x_{k_1}, \ldots x_{k_r}) in & \\ let x'_p = localize(x_p, x_q) & with\ q < p \\ \ldots & \end{array}$$

Lemma 5.2 *Let t of CT be a term in administrative normal form. There exists t', that can be*

5.4 Adding back expose statements

We still need a few additionnal notations before moving to the reduction proof:
¡definition of expose-prefix form¿
¡definition of CTWL¿
This allows us to state this result :

Lemma 5.3 *Let t be a term of CT in the form of 5.1 on the previous page with a $\Gamma\ ;\ \Delta \vdash t : T\ ;\ \Delta'$ typing judgement. Then, there exists t', a term of ELC in expose-prefix form such that :*

- *t' has the same type judgement and semantics than a term of EL.*
- *$\Gamma\ ;\ \Delta t' : T\ ;\ \Delta''$ with $\Delta'' = \{\rho \in \Delta'\}$ (Delta′ restrained to full capabilities)*

proof: (...)
And we can now formulate our final result:

Theorem 4 *Let $\mathcal{P}$ be a program of CT. If $\mathcal{P}$ typechecks, it fulfills the following properties : ¡EL safety guarantees¿*

Table 3: Weakened typing rules

$$\frac{\forall i.\ \Gamma \vdash t_i : T_i}{\Gamma \vdash t_1\ ;\ t_2\ ;\ t_3\ ...\ t_n : T_n} \quad \text{(CW-SEQ)}$$

$$\frac{\Gamma \vdash t_0 : \mathit{bool} \qquad \Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash \text{if } t_0 \text{ then } t_1 \text{ else } t_2 : T} \quad \text{(CW-IF)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \cup \{x : T_1\} \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \quad \text{(CW-LET)}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(SW-VAR)}$$

$$\frac{\Gamma \vdash t : C \qquad \mathit{fields}(C) = \overline{l : D}}{\Gamma \vdash t.l : D_i} \quad \text{(SW-SEL)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : [\rho \blacktriangleright] C \\ l : D \in \mathit{fields}(C) \\ \Gamma \vdash t_2 : [\rho \blacktriangleright] D\end{array}}{\Gamma \vdash t_1.l = t_2 : D} \quad \text{(SW-ASSIGN)}$$

$$\frac{\begin{array}{c}\forall i \in \{1..n\}.\Gamma \vdash t_i : \rho_i \blacktriangleright D_i \\ \mathit{fields}(C) = \overline{l : D}\end{array}}{\Gamma \vdash \text{new } C(\bar{t}) : \rho \blacktriangleright C} \quad \text{(SW-NEW)}$$

$$\frac{\begin{array}{c}\forall i \in \{1..n\}.\Gamma \vdash t_i : T_i \\ \mathit{mtype}(m) = (\Delta, \overline{F}) \to (\Delta', C) \\ \sigma = \mathit{unify}(\overline{T}, \mathit{erase}_{\triangleright}(\overline{F}))\end{array}}{\Gamma \vdash m\,(\bar{t}) : \mathit{erase}_{\triangleright}(\sigma C)} \quad \text{(SW-INVK)}$$

6 Availablity analysis and probable types

Definition 6.1 *We define the predicate* $MbA(t, \Gamma, \rho)$ *as :*

$$Mba(t, \Gamma, \rho) \Leftrightarrow \forall \Delta, \Delta', T.\ \Gamma\ ;\ \Delta \vdash t : T\ ;\ \Delta' \textit{ valid} \implies \rho \in \Delta \qquad (1)$$

We wish to have an algorithm able to compute MbA. To perform this analysis, we need to know if a subterm will be of a specific tracked type. Fortunately, a tracked type can only be inherited from a subterm and the guard of such a tracked type can not be affected by the present capabilities (see lemma 3.4 on page 7), so we can use a weakened set of rules that discard any information about capabilities and non-tracked guards in order to know which variables may be of a tracked type.

Such rules are presented in 6 on the previous page. They are fully syntax-directed, so writing a typing algorithm is immediate. We present now a few interesting properties of the types provided by this weakened system.

Definition 6.2 *We say that* T *is a probable type of* t *under* Γ *iff* $\Gamma \vdash t : T$ *under the rules presented in table 6 on the preceding page*

Properties 6.1

- *If* T *is a probable tracked type of* t *under* Γ*, then :*

$$\exists \Delta, \Delta'.\ \Gamma\ ;\ \Delta \vdash t : T\ ;\ \Delta' \textit{ valid or } !\exists \Delta, \Delta', T'.\ \Gamma\ ;\ \Delta \vdash t : T'\ ;\ \Delta' \textit{ valid}$$

- *If* T_1 *and* T_2 *are two probably types of* t*, then they are equal if we consider fresh guards are equivalent.*
- *If* t *has a type under the full rules with some input* (Γ, Δ)*, then it has a probable type with* Γ

Lemma 6.2 $MbA(t, \Gamma, \rho)$ *holds if and only if, for all subterms of* t *with a probable type of* $\rho \blacktriangleright C$*, this subterm is not used as a consumed argument of a method call or an argument of a new call.*

Given it is easy to get probable types of terms, we are now able to compute $MbA(t, \Gamma, \rho)$ for arbitrary parameters, which allows us to move to the presentation of the algorithm.

7 Typing Algorithm

This algorithm is close to the Hindley-Milner standard type inference in that it uses guard variables and builds a constraint set that has to be solved by unification, yielding a substitution from guard variables to guards.

In this section, when we write that σ satisfies a constraint set C, we mean that σ is an application from guard variables to guards that is a solution of C.

Given we use guard variables, we extend the definition of the greatest common dominated set thanks to the following definitions :

Definition 7.1 *When considering capability sets with guard variables, we define the capability* $\rho \to \bigcap_i \chi_i$ *that fulfills the following properties :*

- $\sigma(\rho \to \bigcap_i \chi_i) = \rho \to \delta$ *iff* $\forall i.\ \sigma(\chi_i) = \delta$
- $\sigma(\rho \to \bigcap_i \chi_i) = null$ *in other cases.*

Definition 7.2 *Let* Δ_1 *and* Δ_2 *be capability sets with guard variables. Then, we define :*

$$\Delta^* = gcd(\Delta, \Delta') \Leftrightarrow \begin{cases} \forall \sigma.\ \sigma(\Delta^*) \preceq \sigma(\Delta_1) \wedge \sigma(\Delta^*) \preceq \sigma(\Delta_2) \\ \forall \Delta'.\ \forall \sigma.\ \sigma(\Delta^*) \preceq \sigma(\Delta_1) \wedge \sigma(\Delta^*) \preceq \sigma(\Delta_2) \implies \Delta' \preceq \Delta^* \end{cases}$$

(proof well-defineness?)

7.1 Specifications

- Inputs : $\Gamma, \Delta, \Delta', t.\ \Delta'$ may only contain pure capabilities.
- Outputs : T, Δ_r and a constraint set C containing equations relating guards and guard variables (guard variables may appear in both T and Δ_r)

Properties

$$\forall \sigma.\ \begin{cases} \sigma \text{ satisfies } C \\ \Delta' \subset \Delta_r \end{cases} \implies \Gamma\ ;\ \sigma(\Delta) \vdash t : \sigma(T)\ ;\ \sigma(\Delta_r) \text{ valid} \qquad \text{(P1)}$$

$$\forall T^*, \Delta^*.\ \Gamma\ ;\ \Delta \vdash t : T^*\ ;\ \Delta^* \text{ valid} \implies \exists \sigma.\ \begin{cases} (T^*, \Delta^*) \preceq (\sigma(T), \sigma(\Delta_r)) \\ \sigma \text{ satisfies } C \end{cases} \qquad \text{(P2)}$$

7.2 Preliminary notes

Here i explain stuff about reducing to method calls and function calls with only variables.

7.3 Definition

Provided with t, Γ, Δ, Δ', we branch on the syntax of t:

- Variables : t := "x"

  We immediately return $(T, \Delta, \emptyset)$ where $x : T \in \Gamma$

- Let : t := "let x = t_1 in t_2"
  - We define $\Delta_1 = \Delta' \cup \{\rho \| MbA(t2, \Gamma \cup \{x : probable\_type(t1)\}, \rho\}$
  - We recursively typecheck t_1 with Γ, Δ and Δ_1 and get (T_1, Δ_2, C_1)

- We recursively typecheck t_2 with $\Gamma \cup x : T_1$, Δ_2 and Δ' and get (T_2, Δ_3, C_2)
- We return $(T_2, \Delta_3, C_1 \cup C_2)$

- If : t := "if (v) then t_1 else t_2"

1. We check v is of type bool. If not, we fail (we may return an unsolvable constraint set for example).
2. We recursively typecheck t_1 with Γ, Δ and Δ' and get (T_1, Δ_1, C_1)
3. We recursively typecheck t_2 with Γ, Δ and Δ' and get (T_2, Δ_2, C_2)
4. If T_1 and T_2 don't share the same class type, we fail.
5. If T_1 and T_2 are equal, return $(T_1, gcd(\Delta_1, \Delta_2), C_1 \cup C_2)$
6. If $T_1 = \rho_1 \blacktriangleright C$ and $T_2 = \rho_2 \blacktriangleright C$, with $\rho_1 \neq \rho_2$, three subcases arise:
   - If $\{\rho_1, \rho_2\} \subset \Delta'$, fail.
   - If $\rho_1 \in \Delta'$ and $\rho_2 \notin \Delta'$, check that ρ_2 is in Δ_2 (else fail) then return $(\rho_1 \triangleright C, gcd(\Delta_1, (\Delta_2 - \rho_2) \otimes \{\rho_2 \rightarrow \rho_1\}), C_1 \cup C_2)$
   - If $\rho_1 \notin \Delta'$ and $\rho_2 \notin \Delta'$, check that ρ_1 and ρ_2 are respectively in Δ_1 and Δ_2 (else fail), then take χ_k fresh guard variable and return $(\chi_k \triangleright C, gcd((\Delta_1 - \rho_1) \otimes \{\rho_1 \rightarrow \chi_k\}, (\Delta_2 - \rho_2) \otimes \{\rho_2 \rightarrow \chi_k\}), C_1 \cup C_2)$
7. If $T_1 = \rho \blacktriangleright C$ and $T_2 = \delta \triangleright C$, with $\rho \neq \delta$, two subcases arise:
   - If $\rho \in \Delta'$, check that ρ is in Δ_1 (else fail), then return $(\delta \triangleright C, gcd((\Delta_1 - \rho) \otimes \{\rho \rightarrow \delta\}, \Delta_2), C_1 \cup C_2)$
   - If $\rho \notin \Delta'$, fail.
8. If $T_1 = \rho \blacktriangleright C$ and $T_2 = \rho \triangleright C$, return $(\rho \triangleright C, gcd(\Delta_1, \Delta_2), C_1 \cup C_2)$
9. If $T_1 = \delta_1 \triangleright C$ and $T_2 = \delta_2 \triangleright C$ with $\delta_1 \neq \delta_2$, fail.

- Method Calls : m(x_1, x_2, ... x_n)

1. Let F_i be the type requested by the method for the i^{th} parameter. Now, for all parameters, let us compare the type T_i of x_i in Γ with F_i. Four cases arise with which we infer the $\{T'_i\}$ types and iterately build the set Δ^* of the remaining capabilities (at the beginning $(\Delta^*_0 = \Delta)$:
   - If F_i and T_i are of different class types, fail
   - If F_i and T_i are both guarded or both tracked : $T'_i = T_i$, $\Delta^*_i = \Delta^*_{i-1}$
   - If $F_i = \delta \triangleright C$, $T_i = \rho \blacktriangleright C$ and $\rho \in \Delta'$: $T'_i = \rho \triangleright C$ and $\Delta^*_i = \Delta^*_{i-1}$
   - If $F_i = \delta \triangleright C$, $T_i = \rho \blacktriangleright C$ and $\rho \notin \Delta'$: take χ_k fresh guard variable and set $T'_i = \chi_k \triangleright C$ and $\Delta^*_i = \Delta^*_{i-1} - \rho$ (if $\rho \notin \Delta^*_{i-1}$, fail)

2. Then we unify F_i and T'_i. For every i and j such that $guard^1(F_i) = guard(F_j)$, we add the constraint $guard(T'_i) = guard(T'_j)$ to the set C (starting empty)
3. Now, we consider the return type F_r of the method. If the guard α of F_r doesn't appear in any of the $\{F_i\}$, then we define T_r as $[\alpha := \gamma]F_r$ where γ is a fresh guard. If it appears in some F_p, then we set T_r as $[\alpha := guard(T'_p)]$.
4. We can now return (T_r, Δ^*_n, C)

7.4 Proof of correctness

By induction on the structure of t :

- The case of variables is immediate.
- Let : t := "let x = t_1 in t_2"

  1. Proof of `P1`

     Let σ be a substitution satisfying C. As $C = C_1 \cup C_2$, σ also satisfy C_1 and C_2.

     By induction hypothesis, we have:

     $$\Gamma \ ; \ \sigma(\Delta) \vdash t_1 : \sigma(T_1) \ ; \ \sigma(\Delta_2)$$
     $$\Gamma \ ; \ \sigma(\Delta_2) \vdash t_2 : \sigma(T_2) \ ; \ \sigma(\Delta_3)$$

     And by a direct application of `C-LET`, we get the conclusion.

- If : t := "if (v) then t_1 else t_2"

  1. Proof of `P1`

     The algorithm already checked that v fulfilled : $\Gamma \ ; \ \Delta \vdash v : bool \ ; \ \Delta$, so we can move to the two other premisces.

     We have the following two results by induction :

     $$\Gamma \ ; \ \sigma(\Delta) \vdash t_1 : \sigma(T_1) \ ; \ \sigma(\Delta_1)$$
     $$\Gamma \ ; \ \sigma(\Delta) \vdash t_2 : \sigma(T_2) \ ; \ \sigma(\Delta_2)$$

     Now, let us consider the different possibilities that were described in the algorithm :

     - All cases of failure are immediate given we are proving a safety property.
     - T_1 and T_2 are equal : we can directly apply `C-IF`

[1] guard being the application that associates ρ to the types $\rho \blacktriangleright C$ and $\rho \triangleright C$

- $T_1 = \rho_1 \blacktriangleright C$, $T_2 = \rho_2 \blacktriangleright C$, $\rho_1 \in \Delta'$ and $\rho_2 \notin \Delta'$: we can apply rule `T-LOC` to t_2, yielding :

$$\Gamma \ ; \ \Delta \vdash t_2 : \rho_1 \triangleright C \ ; \ (\Delta_2 - \rho_2) \otimes \{\rho_2 \rightarrow \rho_1\}$$

  and then rule `C-IF` applies.
- $T_1 = \rho_1 \blacktriangleright C$, $T_2 = \rho_2 \blacktriangleright C$, $\rho_1 \notin \Delta'$ and $\rho_2 \notin \Delta'$: we can apply rule `T-LOC` to both t_1 and t_2, yielding :

$$\Gamma \ ; \ \Delta \vdash t_1 : \sigma(\chi_k) \triangleright C \ ; \ (\Delta_2 - \rho_1) \otimes \{\rho_1 \rightarrow \chi_k\}$$
$$\Gamma \ ; \ \Delta \vdash t_2 : \sigma(\chi_k) \triangleright C \ ; \ (\Delta_2 - \rho_2) \otimes \{\rho_2 \rightarrow \chi_k\}$$

  and we can then apply again rule `C-IF`.
- $T_1 = \rho \blacktriangleright C$, $T_2 = \delta \triangleright C$ and $\rho \in \Delta$: we can again apply `T-LOC` to t_2, followed by `C-IF`
- $T_1 = \rho \blacktriangleright C$ and $T_2 = \rho \triangleright C$: we can apply `W-EXP` on t_1, then `C-IF`

- Method Calls : m(x_1, x_2, ... x_n)

  1. Proof of `P1`

References

[1] Ph. Haller, M. Odersky : Capabilities for external uniqueness.

[2] B. Alpern, F. B. Schneider : Recognizing safety and liveness.

[3] C. Flanagan, A. Sabry, B. F. Duba, M. Felleisen : The essence of compiling with continuations.