Effect Systems with Subtyping

Yan Mei Tang
Pierre Jouvelot

Centre de Recherche en Informatique
Ecole Des Mines de Paris
E-mail: {tang,jouvelot}@cri.ensmp.fr

Abstract

Effect systems extend classical type systems with effect information. Just as types describe the possible values of expressions, effects describe their possible evaluation behaviors. Effects, which appear in function types, introduce new constraints on the typability of expressions. To increase the flexibility and accuracy of effect systems, we present a new effect system based on subtyping. The subtype relation is induced by a subsumption relation on effects. This subtyping effect system avoids merging effect information together, thus collecting more precise effect information. We introduce a reconstruction algorithm which for any expression already typed with classical types, reconstructs its type and effect based on the subtype relation. The reconstruction algorithm is sound and complete w.r.t. the static semantics.

1 Introduction

Effect systems extend classical type systems with effect information. Just as types describe the possible values of expressions, effects describe their possible evaluation behaviors. Effect systems allow powerful static analysis to be performed in the presence of higher-order functions, imperative constructs and separate compilation. However, effects, which appear in function types, introduce new constraints on the typability of expressions, i.e., effect checking may force the rejection of programs which would have type-checked if no effects were present.

To increase the flexibility of previous effect systems, subflecting has been introduced [Gifford87, Talpin92, Tang92]. Subflecting allows expressions to admit larger effects, thus enabling type mismatches due to the introduction of effect information to be eliminated. But, subflecting alone forces a variable to have a unique type in different occurrences and thus merges effect information together; this often limits the accuracy of the effect analysis. Instead of relying on subflecting to eliminate undue type mismatches, we use the notion of effect-based subtyping to improve both the flexibility and accuracy of effect systems, while preserving type safety and reconstruction.

We present a new effect system based on subtyping where the subtype relation is induced by a subsumption rule on effects. This subtyping effect system avoids merging effect information together when forcing two types to be identical, thus it collects more precise effect information. We introduce a sound and complete reconstruction algorithm for this static semantics. Since type inequalities are only introduced by effect subsumption, it is a simple extension of classical type reconstruction algorithms with effect constraints whose solutions satisfy the subtype relation. To motivate this new notion, we show how to use effect-based subtyping within the effect system for control-flow analysis presented in [Tang92], thus improving the accuracy of this control-flow analysis technique.

In the sequel, we define a type and effect static semantics based on subtyping (Section 2), present the type and effect reconstruction algorithm and prove it sound and complete w.r.t. the static semantics (Section 3), discuss the related work (Section 4) before concluding (Section 5). The proofs are given in appendix.

2 Static Semantics with Subtyping

2.1 Language

A simple functional language is enough to present our ideas, although our analysis can be extended to additional language constructs, such as constants, imperative operations, separate compilation (in the vein of [Tang94-1, Tang94-2]). In particular, our analysis can also deal with polymorphism as presented in [Tang94-1]. The syntax of expressions is defined as follows:

\[ e ::= x \quad \text{value identifier} \]
\[ (\lambda x (e)) \quad \text{abstraction} \]
\[ (\text{rec} (f) e) \quad \text{recursive function} \]
\[ (e e') \quad \text{application} \]

where all lambda expressions, recursive or not, are explicitly given a name \( n \) (from the domain Id of identifiers) which is used to uniquely identify them. These names could be automatically assigned by the reconstruction process.

2.2 Domains

Classical types specify the data structure of expressions. A classical type \( \tau \) can either be \( \text{int} \), a type variable \( \alpha \), or a function type \( \tau' \rightarrow \tau \). A classical type environment \( \mathcal{T} \) is a finite map from identifiers to classical types.

\[ \tau \in \text{CTyp} = \text{int} | \alpha | \tau' \rightarrow \tau \quad \text{classical type} \]
\[ \mathcal{T} \in \text{CTEnv} = \text{Id} \mapsto \tau \quad \text{classical environment} \]

Effect systems extend classical types with effect information. An \( \text{type} \ t \) is either \( \text{int} \), a type variable \( \alpha \), or a function
type \( t' \xrightarrow{c} t \) with the latent effect \( c \), which abstracts the control-flow behavior of the function body. A type environment \( E \) is a finite map from identifiers to types:

\[
t \in \text{Type} = \text{int} \mid \alpha \mid t' \xrightarrow{c} t \quad \text{type}
\]

\[
E \in \text{TEnv} = \text{id} \rightarrow t \quad \text{type environment}
\]

Control-flow effects record the function names that are possibly called during the evaluation of expressions. An effect \( c \) can either be the constant \( \emptyset \), denoting the absence of any function call, or an effect variable \( \zeta \), a singleton \( \{ \alpha \} \) where \( \alpha \) is the name of a called function, or a union of a set of function names indicated by the infix union operator \( \cup \).

\[
c \in \text{Control} = \emptyset \mid \zeta \mid \{ \alpha \} \mid c \cup c'
\]

### 2.3 Subtype and Subeffect Relations

An effect, i.e. a set of function names, can be conservatively approximated by one of its supersets. The subtype relation is thus the usual set inclusion relation.

The subtype relation \( \leq \) is defined via effect inclusion between latent effects of function types that have the same structure. To properly define this notion, we introduce the \( \text{Struct} \) function which transforms types to classical types by erasing latent effects.

\[
\text{Struct}(\text{int}) = \text{int} \\
\text{Struct}(\alpha) = \alpha \\
\text{Struct}(t' \xrightarrow{c} t) = \text{Struct}(t') \rightarrow \text{Struct}(t)
\]

The type structure of \( t \) is \( \text{Struct}(t) \). Two types \( t \) and \( t' \) have the same structure if and only if \( \text{Struct}(t) = \text{Struct}(t') \).

The subtype relation \( t \leq t' \) is defined when \( t \) and \( t' \) have the same structure. Note that the subtype relation between function types is contravariant.

**Definition 1 (Subtype)**

\[
\begin{align*}
\alpha & \leq \alpha \\
\text{int} & \leq \text{int} \\
t_0 \leq c_0 t_0 & \leq t'_0 \xrightarrow{c_0} t_1 \quad \Leftrightarrow \quad t'_0 \leq t_0 \land t_0 \leq t_1 \land c_1 \geq c_0
\end{align*}
\]

The function \( \text{Eff} \) generates the set of effect inequalities corresponding to a given type inequality. An effect inequality is a pair \((c_i, c'_i)\) written \( c_i \geq c'_i \).

\[
\begin{align*}
\text{Eff}(\alpha \leq \alpha) & = \emptyset \\
\text{Eff}(\text{int} \leq \text{int}) & = \emptyset \\
\text{Eff}(t_0 \leq c_0 t_0 \leq t'_0 \xrightarrow{c_0} t_1) & = \{c_i \geq c_0 \} \cup \text{Eff}(t'_0 \leq t'_0) \\
& \cup \text{Eff}(t_0 \leq t_1)
\end{align*}
\]

### 2.4 Semantics

The static semantics defines the type and control-flow effect of expressions. It is specified by a set of inference rules [Plotkin81]. Given a type environment \( E \), the inference rules associate an expression \( e \) with its type \( t \) and control-flow information \( c \). We write:

\[
E \vdash e : t, c
\]

The crucial rules are the \((\text{abs})\) and \((\text{app})\) rules for lambda abstraction and application. In the abstraction case, the current function name is added to the functions called by the lambda body; the resulting set is the latent control-flow effect of the lambda expression. When such a function is applied, in the \((\text{app})\) rule, this latent control-flow information is used to determine the functions possibly called while evaluating the function body.

\[
\begin{align*}
(\text{var}) & : E[x \mapsto t] \vdash x : t, \emptyset \\
(\text{abs}) & : E \vdash (\lambda n (x) e) : t' \xrightarrow{[n]} t, \emptyset \\
(\text{rec}) & : E \vdash (\text{recn} (\{x\} e)) : t, \emptyset \\
(\text{app}) & : E \vdash e : t' \xrightarrow{c} \quad E \vdash e' : t', \text{subtype of } E \vdash e : t, c
\end{align*}
\]

The novelty here lies in the \((\text{sub})\) rule where we use subtyping to allow a larger type \( t' \) to be used in lieu of \( t' \). This increases the flexibility of the static semantics by relaxing the constraint on latent effects imposed by the context of an expression. We show that this new approach performs a better analysis than the one previously introduced in [Tang92] (see an example in Section 4) which used the less precise subeffecting rule:

\[
E \vdash e : t', c
\]

\[
E \vdash e : t, c
\]

### 3 Reconstruction with Subtyping

We present a new reconstruction algorithm that reconstructs types and effects of expressions based on the subtype relation. We describe the basic ideas, present the algorithm and state its correctness.

#### 3.1 Basic Idea

The reconstruction of types and effects based on subtyping is a type inequity solving problem. Since the subtype relation in our system is defined by the subsumption relation on effects, type inequities amount to sets of effect inequities when the structures of the types are known. Therefore, we define a type and effect reconstruction algorithm \( S \) which operates on expressions already typed with classical types. For any expression, the reconstruction algorithm \( S \) computes a set of type inequities beside its type and effect. Since classical types specify type structures, solving type inequities is reduced to solving the corresponding effect inequities. Thus reconstruction can be viewed as an effect constraint satisfaction problem. For every expression that has a type and a control-flow effect in the static semantics, its effect constraint set must have at least one solution, which satisfies the set of type inequities. The classical types of expressions are reconstructed by a simple type reconstruction algorithm [Milner78, Tofte87].
3.2 Algorithm $S$

Given a type environment $E$ and an expression $e$ assumed priorly decorated with its classical type (we use a straightforward expression annotation mechanism to express this information in the algorithm), the reconstruction algorithm $S$ computes a type $t$, an effect $e$ and an effect constraint set $\kappa$. We note:

\[ S(E, e) = (t, e, \kappa) \]

The effect constraint set is partly built by application of $Eff$ to type inequalities and partially during the reconstruction of lambda and $\text{rec}$ expressions. The function $New$ transforms a classical type $\tau$ to a type $t$ by adding fresh latent effect variables to $\tau$. Its proper definition is:

\[
\begin{align*}
New(\text{int}) &= \text{int} \\
New(e) &= e \\
New(\tau' \rightarrow \tau) &= New(\tau') \triangleleft New(\tau) \text{ for fresh } \zeta
\end{align*}
\]

The inference algorithm $S$ is defined as follows:

\[
S(E, x) \Rightarrow \begin{cases} 
S(E(x)) \ni \langle t, \emptyset, \text{Eff}(t' \leq t) \rangle 
\end{cases}
\]

\[
S(E, (\lambda x. (x : \tau) e)) \Rightarrow \begin{cases} 
\langle t', \zeta, c, \kappa \rangle = S(E[x \mapsto t'], e) 
\ni \langle t' \triangleleft t, \emptyset, \kappa \cup \{\zeta \cup \{n\} \cup c\} \rangle 
\end{cases}
\]

\[
S(E, (\text{rec}n \ (t : \tau' \rightarrow \tau \ x : \tau') e)) \Rightarrow \begin{cases} 
\langle t', \zeta, c, \kappa \rangle = S(E[t \mapsto t', \zeta], e) 
\ni \langle t' \triangleleft t, \emptyset, \kappa \cup \text{Eff}(t'' \leq t) \cup \{\zeta \cup \{n\} \cup c\} \rangle 
\end{cases}
\]

\[
S(E, (e \ e')) \Rightarrow \begin{cases} 
\langle t'' \triangleleft t', \zeta, c, \kappa \rangle = S(E, e) 
\rangle (t', c', \kappa') = S(E, e') 
\ni \langle (t, c \cup c' \cup c''), \kappa \cup \kappa' \cup \text{Eff}(t' \leq t'') \rangle 
\end{cases}
\]

Subeffecting can be easily reduced by subtyping by noticing that the related reconstruction operator [Tang94-I] is similar to $S$, except that $\leq$ is replaced by the more restrictive $=$, implemented by unification.

3.3 Properties of $S$

The reconstruction algorithm $S$ has the following properties, easily proved by induction:

**Lemma 1 (Properties of $S$)** For any $E$, $e$, if $S(E, e) = (t, e, \kappa)$, then:

- $t$ only includes fresh effect variables.
- All environment extensions within $S$ refer to types with only fresh effect variables.

The previous lemma implies that the constraint set computed by the reconstruction algorithm $S$ has the following normal form property:

**Lemma 2 (Normal Constraints)** If $S(E, e) = (t, e, \kappa)$, then $\kappa$ has the normal form $\{\zeta_i \geq c_i \mid i = 1..s\}$.

**Proof** See the appendix.

3.4 Constraint Satisfaction

An expression $e$ with its type environment $E$ is type and effect safe if and only if the constraint set $\kappa$ computed by $S(E, e)$ admits at least one solution. A constraint set that is in normal form always has solutions, among which we are interested in the minimal one. The substitutions satisfying $\kappa$ are called effect models.

**Definition 2 (Effect Model)** A substitution $\mu$ is an effect model of a constraint set $\kappa$, noted as $\mu \models \kappa$, if and only if $\forall \zeta \geq c \in \kappa, \mu \zeta \geq \mu c$.

The following lemma shows how to satisfy a type inequality by solving its corresponding effect constraint.

**Lemma 3 (Solution of Type Inequality)** If $t$ and $t'$ have the same structure and satisfy Lemma 1, then

\[
\mu \models \text{Eff}(t' \leq t) \Rightarrow \mu t' \leq \mu t
\]

**Proof** By induction of the structure of types.

**Theorem 1 (Satisfaction)** Every normal form constraint set $\kappa = \{\zeta_i \geq c_i \mid i = 1..s\}$ admits at least one model.

**Proof** $\{\zeta_i \mapsto c_i' \mid i = 1..n\}$ is an effect model of $\kappa$, where $c_i' := \{c \cup \{c_i\} \cup \zeta\}$, where $\|$ is the set difference operator.

A constraint set may admit more than one effect model, among which we are interested in the minimal one. We define a function $Min$ to characterize the minimal effect model of a constraint set $\kappa$. Note that the solution is independent of the order of inequalities in $\kappa$ because of the algebraic properties of $\cup$: the function $Min$ recursively computes an effect model by applying each solved inequation to the residual constraints.

\[
Min(\emptyset) = Id \\
Min(\{\zeta \geq c \cup \kappa'\}) = let \mu = Min(\kappa') in \{\zeta \mapsto \mu c \cup \zeta\}; \mu
\]

The constraint set of the reconstruction algorithm always admits a unique minimal model with respect to the subsumption relation $\geq$ on effects.

**Theorem 2 (Minimality)** Any constraint set admits a unique minimal effect model.

**Proof** By induction on the constraint set.

3.5 Correctness

Since the reconstruction algorithm $S$ is defined by induction on the structure of expressions, which are of finite height, it always terminates.

The reconstruction algorithm is sound and complete with respect to the static semantics. The soundness theorem states that the application of any effect model of the reconstructed type constraint set to the reconstructed type and effect satisfies the static semantics.

**Theorem 3 (Soundness)** Given an expression $e$ and its type environment $E$, if $S(E, e) = (t, e, \kappa)$, then for any effect model $\mu$ of $\kappa$, one has:

\[
\mu E \vdash e : pt, \mu c
\]
Proof See the appendix.

The completeness theorem states that the reconstructed type \( \tau \) and the control-flow effect \( \xi \) are minimal with respect to any type \( \tau \) and control-flow effect \( \xi \) derivable from the static semantics, modulo some substitution \( \theta \) that satisfies the computed constraint set \( \kappa \). The substitution \( \theta \) ranges over the free variables of \( \xi \).

Theorem 4 (Completeness) If \( \theta, \xi \vdash e : \tau, \xi \) then \( S(\xi, e) = (\tau, \xi, \kappa) \) and there exists an effect model \( \mu \) of \( \kappa \), such that:

\[ \theta, \xi = \mu, \xi \text{ and } \mu \leq \tau \text{ and } \mu \kappa \geq \mu \kappa \]

Proof See the appendix.

4 Related Work

Subtyping (see e.g. [Cardelli88]) adds flexibility to type systems by allowing type coercions to be performed if necessary in the presence of type mismatches. It is often used to capture aspects of object-oriented programming [Wand87, Stansifer88]. Subtyping in effect systems has been previously introduced in explicitly typed languages [Gifford87, Consel94]. There, a subsumption rule similar to the one presented above was used, but since only type checking was performed, its treatment was simpler than ours. This paper shows that type and effect reconstruction may be performed in an implicitly typed language.

Previous implicit effect systems [Dornic91, Talpin92, Tang92] have introduced subeffecting via the subeffecting rule (see Section 2), to increase the flexibility of the static semantics. Subeffecting allows expressions of the same classical types to also have the same effect-including types by allowing such effects to be replaced by larger ones if need be. Subtyping eliminates this information loss by allowing these expressions to simply obey the subtype relation. We show below on an example how subtyping can thus be more precise than subeffecting:

\[ ((\lambda_1 f) f) \]
\[ (f (\lambda_1 (a, a) ) )_1 \]
\[ (f (\lambda_2 (b, b) ) )_1 \]

There, the function \( f \) is bound when performing the call \( \lambda_2 f \) and is applied at \( \lambda_1 \) and \( \lambda_2 \) with arguments \( \lambda_1 (a, a) \) and \( \lambda_2 (b, b) \) respectively. We give, in the following table, the types of \( f \) at these three occurrences \( (t_f, t_{fa}, t_{fa}) \), and the types of the arguments \( \lambda_1 \) and \( \lambda_2 \) \( (t_n, n) \) and \( (t_{n2}, n_2) \) For clarity, we use \( i \) to indicates the type int.

<table>
<thead>
<tr>
<th>Subeffecting</th>
<th>Subtyping</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_f )</td>
<td>( i ) ( \lambda_1 ) ( \lambda_2 ) ( i ) ( i ) ( i ) ( i ) ( i ) ( i ) ( i ) ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) |</td>
</tr>
<tr>
<td>( t_{fa} )</td>
<td>( i ) ( \lambda_1 ) ( \lambda_2 ) ( i ) ( i ) ( i ) ( i ) ( i ) ( i ) ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) |</td>
</tr>
<tr>
<td>( t_n )</td>
<td>( i ) ( \lambda_1 ) ( \lambda_2 ) ( i ) ( i ) ( i ) ( i ) ( i ) ( i ) ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) | ( i ) |</td>
</tr>
<tr>
<td>( t_{fa} = t_f )</td>
<td>( t_f ) | ( t_f ) | ( t_f ) | ( t_f ) | ( t_f ) | ( t_f ) | ( t_f ) | ( t_f ) | ( t_f ) | ( t_f ) |</td>
</tr>
<tr>
<td>( t_{fa} = t_f )</td>
<td>( t_f ) | ( t_f ) | ( t_f ) | ( t_f ) | ( t_f ) | ( t_f ) | ( t_f ) | ( t_f ) | ( t_f ) | ( t_f ) |</td>
</tr>
</tbody>
</table>

Notice that, when using subeffecting, all occurrences of \( f \) are forced to have the same type while, when using subtyping, they only have to obey a subtype relation, leading to more precise local control-flow information.

5 Conclusion

We presented a new effect system based on subtyping where expressions with the same structure obey a subtype relation defined by a subsumption relation on effects. This subtype effect system avoids merging effect information together, thus collects more precise effect information than effect systems with subeffecting. We designed a sound and complete reconstruction algorithm that reconstructs the types and effects of expressions in the presence of subtyping, and show that it outperforms previous systems. A natural extension of this paper is the possibility of combining subtyping and subeffecting in a single framework. This has been proved valuable in [?].

Acknowledgements

We thank Jean-Pierre Talpin for his numerous comments on this paper.

References


Appendix

Proof of Lemma 2

Lemma 2 (Formal Effect Constraints) If $S(\xi, e) = (t, c, \kappa)$, then $\kappa$ is of the following form:

$$\{\zeta_i \supset c_i \mid i = 1..s\}$$

Proof By induction of the structure of expressions.

- Case of $x$
  The hypothesis is
  $$S(\xi, x) = (t, \emptyset, \text{Eff}(t' \leq t))$$
  By the definition of $S$
  (1) $t' = \xi(x)$
  (2) $t = \text{New}(\text{Struct}(t'))$
  From (1), by Lemma 1
  (3) $t'$ only includes fresh effect variables
  From (2)(3), by the definition of $\text{Eff}$
  $\text{Eff}(t' \leq t)$ satisfies the lemma

- Case of $(\lambda x \, e)$
  The hypothesis is
  $$S(\xi, (\lambda x \, \tau) \, e) = (t', t, \kappa \cup \{\zeta \supset \{c\} \cup c\})$$
  By the definition of $S$
  (1) $\xi \text{ new}$
  (2) $(t, t', \kappa) = S(\xi[x \mapsto t'], e)$
  From (2), by induction
  (3) $\kappa$ satisfies the lemma
  From (1)(3)
  $\kappa \cup \{\zeta \supset \{c\} \cup c\}$ satisfies the lemma

- Case of $(\text{recn} (f \, x) \, e)$
  The hypothesis is
  $$S(\xi, (\text{recn} (f : \tau' \rightarrow \tau \, x : \tau) \, e)) = (t', t, \kappa \cup \text{Eff}(t'' \leq t) \cup \{\zeta \supset \{c\} \cup c\})$$
  By the definition of $S$
  (1) $t' \xrightarrow{\zeta} t = \text{New}(\tau' \rightarrow \tau)$
  (2) $(t'', c, \kappa) = S(\xi[f \mapsto t' \xrightarrow{\zeta} t[x \mapsto t'], e)$
  From (1), by the definition of $\text{New}$
  (3) $t = \text{New}(\tau)$
  (4) $\zeta \text{ new}$
  From (2), by Lemma 1
  (5) $t''$ only includes fresh effect variables
  From (2), by induction
  (6) $\kappa$ satisfies the lemma
  From (3)(5), by the definition of $\text{Eff}$
  (7) $\text{Eff}(t'' \leq t)$ satisfies the lemma
  From (6)(7)(4)
  $\kappa \cup \text{Eff}(t'' \leq t) \cup \{\zeta \supset \{c\} \cup c\}$ satisfies the lemma

Proof of Theorem 3

Theorem 3 (Soundness) Given an expression $e$ and its type environment $\xi$, if $S(\xi, e) = (t, c, \kappa)$, then for any effect model $\mu$ of $\kappa$, one has:

$$\mu \xi \vdash e : \mu t, \mu c$$

Proof By induction on the structure of expressions
• Case of (var)
The hypotheses are
(1) $S(\mathcal{E}, x) = \{ t, \emptyset, E\mathcal{E}(t' \leq t) \}$
(2) $\mu \models E\mathcal{E}(t' \leq t)$
From (1), by the definition of $S$
(3) $t' = \mathcal{E}(x)$, i.e. $\mu\mathcal{E}(x) = \mu t'$
From (3), by (var) rule in the static semantics
(4) $\mu\mathcal{E} \vdash x : \mu t', \emptyset$
From (2), by Lemma 3
(5) $\mu t' \leq \mu t$
From (4)(5), by the (sub) rule in the static semantics
$\mu\mathcal{E} \vdash x : \mu t, \emptyset$

• Case of (abs)
The hypotheses are
(1) $S(\mathcal{E}, (\lambda n (x : t) e)) = \langle t', \emptyset, \emptyset \rangle, \zeta \in \{ n \cup c \}$
(2) $\mu \models n \cup \{ n \cup c \}$
where $t' = \text{New}(\tau)$ and $\zeta$ new
From (1), by the definition of $S$
(3) $\langle t, c, \kappa \rangle = S(\mathcal{E}, [x \mapsto t'], e)$
From (2), by the definition of effect models
(4) $\mu \models \kappa$
(5) $\mu = \{ n \cup c \}$
From (3)(4), by induction
(6) $\mu(\mathcal{E}[x \mapsto t']) \vdash e : \mu t, \mu c$
From (6), by (abs) in the static semantics
(7) $\mu\mathcal{E} \vdash (\lambda n (x : t) e) : \mu(t' \leq \mu t'), \emptyset$
From (5), by the definition of subtype relation
(8) $\mu(t' \leq t)$
From (7)(8), by the (sub) rule in the static semantics
$\mu\mathcal{E} \vdash (\lambda n (x : t) e) : \mu(t' \leq t), \emptyset$

• Case of (app)
The hypotheses are
(1) $S(\mathcal{E}, (e e')) = \langle t, c \cup c' \cup c'' \cup \kappa \cup \kappa' \cup E\mathcal{E}(t' \leq t') \rangle$
(2) $\mu \models \kappa \cup \kappa' \cup E\mathcal{E}(t' \leq t')$
From (1), by the definition of $S$
(3) $S(\mathcal{E}, e) = \langle t', c', \kappa' \rangle$
(4) $S(\mathcal{E}, e') = \langle t', c', \kappa' \rangle$
From (2), by the definition of effect models
(5) $\mu \models \kappa'$
(6) $\mu \models \kappa''$
(7) $\mu \models E\mathcal{E}(t' \leq t'' \wedge t')$
From (3)(5) and (4)(6), by induction
(8) $\mu\mathcal{E} \vdash e : \mu(t' \leq t'), \mu c$
(9) $\mu\mathcal{E} \vdash e' : \mu t', \mu c'$
From (7), by Lemma 3
(10) $\mu t' \leq \mu t''$
From (9)(10), by the (sub) rule in the static semantics
(11) $\mu\mathcal{E} \vdash e' : \mu t', \mu c'$
From (8)(11), by (app) in the static semantics
$\mu\mathcal{E} \vdash (e e') : \mu(\kappa \cup c' \cup c'')$ 

□

Proof of Theorem 4
Theorem 4 (Completeness) If $\emptyset, \mathcal{E} \vdash e : t, c_1$, then $S(\mathcal{E}, e) = \langle t, c, \kappa \rangle$ and there exists a effect model $\mu$ of $\kappa$, such that:
\[ \theta_1 \mathcal{E} = \mu \mathcal{E} \text{ and } \mu t \leq t_1 \text{ and } c_1 \supseteq \mu c \]

**Proof** By induction on the structure of expressions

- **Case of (var)**
  The hypothesis is 
  \[ \theta_1 \mathcal{E} \vdash x : t_1, \emptyset \]
  By the (var) and (sub) rules in the static semantics
  \[ \begin{align*}
  (1) & \quad t'_1 = \mathcal{E}(x) \\
  (2) & \quad \theta_1 t'_1 \leq t_1 
  \end{align*} \]
  From (1), by the definition of \( \mathcal{S} \)
  \[ \mathcal{S}(\mathcal{E}, x) = \{ t, \emptyset, \text{Eff}(t'_1 \leq t) \} \]
  where \( t = \text{New(Struct}(t'_1)) \)
  Since \( t \) only includes fresh effect variables, we can defined \( \theta \) such that:
  \[ (3) \quad \theta t = t_1 \]
  We define the effect model \( \mu \), such that :
  \[ \mu v = \begin{cases} 
  \theta v & v \in \text{fe}(t) \\
  \emptyset & \text{otherwise} 
  \end{cases} \]
  Note that since \( t \) only includes fresh effect variables, \( \mu \) is well defined.
  From (2)(3), by the definition of \( \mu \)
  \[ \begin{align*}
  (4) & \quad \mu t'_1 = \theta t'_1 \leq t_1 \\
  (5) & \quad \mu t = \theta t = t_1 
  \end{align*} \]
  From (4)(5), by Lemma 3
  \[ \mu \models \text{Eff}(t'_1 \leq t) \]
  By the definition of \( \mu \)
  \[ \theta_1 \mathcal{E} = \mu \mathcal{E} \]
  From (5)
  \[ \mu t \leq t_1 \]

- **Case of (abs)**
  The hypothesis is 
  \[ \theta_1 \mathcal{E} \vdash (\lambda x \mathcal{E}) e : t'_2 \supseteq t_2, \emptyset \]
  By the (abs) and (sub) rules in the static semantics
  \[ \begin{align*}
  (1) & \quad \theta_1 \mathcal{E} \vdash (\lambda x \mathcal{E}) e : t'_1 \supseteq t_1, \emptyset \\
  (2) & \quad t'_1 \supseteq t_1 \leq t'_2 \supseteq t_2 
  \end{align*} \]
  From (1), by (abs) rule in the static semantics
  \[ \begin{align*}
  (3) & \quad \theta_1 (\mathcal{E}[x \mapsto t]) \vdash e : t_1, c_1 \]
  If \( x \) has classical type \( \tau \), let \( t' = \text{New}(\tau) \).
  Then there exists a substitution \( \theta' \), such that:
  \[ (4) \quad t'_1 = \theta t' \]
  We define a substitution \( \theta'_1 \), such that :
  \[ \theta'_1 v = \begin{cases} 
  \theta v & v \in \text{fe}(t') \\
  \emptyset & \text{otherwise} 
  \end{cases} \]
  Note that since \( t' \) only includes fresh effect variables, \( \theta'_1 \) is well defined.
  From (4), by the definition of \( \theta'_1 \), (3) is equivalent to :
  \[ (5) \quad \theta'_1 (\mathcal{E}[x \mapsto t']) \vdash e : t_1, c_1 \]
  From (5), by induction
  \[ \begin{align*}
  (6) & \quad \mathcal{S}(\mathcal{E}[x \mapsto t'], e) = \{ t, c, \kappa \} \\
  \end{align*} \]
  where exists \( \mu \), such that :
  \[ \begin{align*}
  (7) & \quad \mu \models \kappa \\
  (8) & \quad \theta'_1 (\mathcal{E}[x \mapsto t']) = \mu (\mathcal{E}[x \mapsto t']) \\
  (9) & \quad \mu t \leq t_1 \\
  (10) & \quad c_1 \supseteq \mu c 
  \end{align*} \]
  From (8)(4), by the definition of \( \theta'_1 \)
  \[ (11) \quad \theta_1, \mathcal{E} = \mu \mathcal{E}, \text{ except on } x \text{ which doesn’t appear in } \mathcal{E} \]
  (alpha-renaming)
  \[ (12) \quad t'_1 = \mu t' \]
  From (6), since \( t' = \text{New}(\tau) \), by the definition of \( \mathcal{S} \)
  \[ \begin{align*}
  (13) & \quad \mathcal{S}(\mathcal{E}, \lambda x : \tau e) = \{ t', c, \kappa \cap \{ n \cup c \} \} \\
  \end{align*} \]
  where \( \zeta \) new
  We define an effect substitution \( \mu' \) on \( \text{fe}(\mathcal{E}, t', c, \kappa) \)
  \[ \mu' e = \begin{cases} 
  \mu v & v \in \text{fe}(\mathcal{E}, t', c, \kappa) \\
  c_2 & v = \zeta 
  \end{cases} \]
  Note that since \( \zeta \) is fresh, \( \mu' \) is well defined.
  From (7), by the definition of \( \mu' \)
  \[ \begin{align*}
  (14) & \quad \mu' \models \kappa \\
  \end{align*} \]
  By the definition of \( \mu' \)
  \[ \begin{align*}
  (15) & \quad \mu' \zeta = c_2 \\
  (16) & \quad \mu' (n \cup c) = \{ n \} \cup \mu c 
  \end{align*} \]
  From (10)(16), by the definition of \( \mu' \)
  \[ \begin{align*}
  (17) & \quad \{ n \} \cup c_1 \supseteq \mu' (n \cup c) 
  \end{align*} \]
  From (2), by the definition of the subtype relation
  \[ \begin{align*}
  (18) & \quad t_1 \leq t_2 \\
  (19) & \quad t'_2 \leq t_1 \\
  (20) & \quad c_2 \supseteq \{ n \} \cup c_1 
  \end{align*} \]
  From (20)(15)(17), by the definition of effect models
  \[ \begin{align*}
  (21) & \quad \zeta \models \{ c, \zeta \} \supseteq \{ n \} \cup c \}
  \end{align*} \]
  From (14)(21), by the definition of effect models
  \[ \begin{align*}
  (22) & \quad \mu' \models \kappa \cup \{ c, \zeta \} \supseteq \{ n \} \cup c 
  \end{align*} \]
  From (11)(12), by the definition of \( \mu' \)
  \[ \theta_1 \mathcal{E} = \mu' \mathcal{E} \]
(22) \( t'_1 = \mu \xi' \)

From (9), by the definition of \( \mu' \)

(23) \( \mu' t \leq t \)

From (22)-(23)(15), by the definition of subtype relation

(24) \( \mu'(t' \leq t) \leq t'_1 \leq t \)

From (18)(10), by the definition of subtype relation

(25) \( t'_2 \leq t'_1 \)

From (24)(25)

(26) \( \mu'(t' \leq t) \leq t'_2 \)

• Case of (rec)

The hypothesis is

\[ \theta; \mathcal{E} \vdash (\text{recn} (f \ x) \ e) : t'_1 \leq t_2, \emptyset \]

By the (rec) and (sub) rules in the static semantics

(1) \( \theta; \mathcal{E} \vdash (\text{recn} (f \ x) \ e) : t'_1 \equiv t_1, \emptyset \)

(2) \( t'_1 \equiv t_1 \leq t'_2 \equiv t_2 \)

From (1), by the (rec) rule in the static semantics

(3) \( \theta; \mathcal{E} \vdash (f \ x) \ e : t'_1 \equiv t_1, c_1 \)

If \( f \) and \( x \) have classical types \( \tau' \rightarrow \tau \) and \( \tau \) respectively, let \( (t' \leq t) = \text{New}(\tau' \rightarrow \tau) \).

Then, there exists a substitution \( \theta \), such that:

(4) \( t'_1 \equiv t \equiv \theta(t' \leq t) \)

We define a substitution on effect variables \( \theta'_1 \), such that:

\[ \theta'_1 v = \begin{cases} v & \text{if } v \in \text{fv}(t' \leq t) \\ \theta_1 v \quad & \text{otherwise} \end{cases} \]

Note that since \( t' \leq t \) only includes fresh effect variables, \( \theta'_1 \) is well defined.

From (4), by the definition of \( \theta'_1 \), (3) is equivalent to:

(5) \( \theta'_1 (\mathcal{E}[f \mapsto t' \leq t][x \mapsto t'_1]) \vdash e : t_1, c_1 \)

From (5), by induction

(6) \( \mathcal{S}(\mathcal{E}[f \mapsto t' \leq t][x \mapsto t'_1], e) = (t'' \leq t, c, \kappa) \)

there exists \( \mu \), such that:

(7) \( \mu \vdash \kappa \)

(8) \( \theta'_1 (\mathcal{E}[f \mapsto t' \leq t][x \mapsto t'_1]) = \mu(\mathcal{E}[f \mapsto t' \leq t][x \mapsto t'_1]) \)

(9) \( \mu t'' \leq t_1 \)

(10) \( c_1 \equiv \mu c \)

From (6), since \( t' \leq t \) = \text{New}(\tau' \rightarrow \tau) \), by the definition of \( \mathcal{S} \)

(11) \( \mathcal{S}(\mathcal{E}, (\text{recn} (f \mapsto t' \rightarrow \tau x \mapsto \tau') e)) = \langle t'' \leq t, \emptyset \cup \text{Eff} (t'' \leq t) \cup \{ \zeta \geq \{ n \} \cup \emptyset} \)

From (8)(4), by the definition of \( \theta'_1 \), except on \( f \) and \( x \) which don’t occur in \( \mathcal{E} \) (alpha-renaming)

\[ \theta'_1 \mathcal{E} = \mu \mathcal{E} \]

(12) \( t'_1 \equiv t_1 \)

(13) \( t = \mu t \)

(14) \( \{ n \} \cup c_1 = \mu \zeta \)

From (9)(13), by Lemma 3

(15) \( \mu \vdash \text{Eff} (t'' \leq t) \)

From (14)(10), by the definition of effect models

(16) \( \mu \vdash \{ \zeta \geq \{ n \} \cup \emptyset \}

From (7)(15)(16), by the definition of effect models

(17) \( \mu \vdash \kappa \cup \text{Eff} (t'' \leq t) \cup \{ \zeta \geq \{ n \} \cup \emptyset \}

From (12)(2)

(18) \( \mu(t' \leq t) \leq t'_2 \equiv t_2 \)

• Case of (app)

The hypotheses is

\[ \theta; \mathcal{E} \vdash (e e') : t_2, c_1 \cup c_1' \cup c''_1 \]

By the (app) and (sub) rules in the static semantics

(1) \( \theta; \mathcal{E} \vdash (e e') : t_1, c_1 \cup c_1' \cup c''_1 \)

(2) \( t_1 \equiv t \)

From (1), by the (app) rule in the static semantics

(3) \( \theta; \mathcal{E} \vdash e : t'_1 \equiv t_1, c_1 \)

(4) \( \theta; \mathcal{E} \vdash e' : t'_1 \equiv t_1, c_1' \)

From (3), by induction

(5) \( \mathcal{S}(\mathcal{E}, e) = (t'' \leq t, c_1, \kappa) \)

there exists \( \mu \), such that:

(6) \( \mu \vdash \kappa \)

(7) \( \theta; \mathcal{E} = \mu \mathcal{E} \)

(8) \( \mu(t'' \leq t) \leq t''_1 \leq t_1 \)

(9) \( c_1 \equiv \mu c \)

From (4), by induction

(10) \( \mathcal{S}(\mathcal{E}, (e e') = (t'', c_1', \kappa) \)

3\( \mu' \), such that:

(11) \( \mu' \vdash \kappa' \)

(12) \( \theta; \mathcal{E} = \mu'e \)

(13) \( \mu' \leq t''_1 \)

(14) \( c'_1 \equiv \mu' c'_1 \)

From (5)(10), by the definition of \( \mathcal{S} \)

\[ \mathcal{S}(\mathcal{E}, (e e')) = (t, c \cup c' \cup c''_1, \kappa \cup \kappa' \cup \text{Eff} (t' \leq t'')) \]

We define a substitution \( \mu'' \) on \( \text{fe}(\mathcal{E}, t' \leq t, c, \kappa) \), and \( \text{fe}(\mathcal{E}, t', c', \kappa') \)
\[
\mu''v = \begin{cases} 
\mu v & v \in f_t(\mathcal{E}, t', c', \kappa') \\
\mu' v & v \in f_t(\mathcal{E}, t', c', \kappa) 
\end{cases}
\]

Note that if \( v \in f_t(\mathcal{E}, t', c', \kappa') \cap f_t(\mathcal{E}, t'' \rightarrow t, c, \kappa) \), then, by the definition of \( \mathcal{S} \), \( v \in f_t(\mathcal{E}) \) and thus, by (7)(12), \( \mu v = \mu' v \); thus \( \mu'' \) is well defined.

From (6)(11), by the definition of \( \mu'' \)
(16) \( \mu'' \models \kappa \)
(17) \( \mu'' \models \kappa' \)

From (8), by the definition of \( \mu'' \)
(18) \( \mu''(t' \rightarrow t) = \mu(t'' \rightarrow t) \leq t'' \rightarrow t' \)

From (18), by the definition of the subtype relation
(19) \( \mu'' \leq t \)
(20) \( t' \leq \mu'' \)
(21) \( c_i' \leq c_i'' \)

From (13), by the definition of \( \mu'' \)
(22) \( \mu'' t = \mu' t' \leq t' \)

From (20)(22), by Lemma 3
(23) \( \mu'' \models \text{Eff} (t' \leq t'') \)

From (16)(17)(23), by the definition of effect models
\( \mu'' \models \kappa \cup \kappa' \cup \text{Eff} (t' \leq t'') \)

From (19)(2)
\( \mu'' \leq t_2 \)

From (9)(14)(21), by the definition of \( \mu'' \)
\( c_i \cup c_i' \cup c_i'' \geq \mu''(c \cup c' \cup c'') \)  \( \square \)