Submitted for publication in *Information and Computation*, Academic Press, 1994.

The Type and Effect Discipline

Jean-Pierre Talpin[*]

And

Pierre Jouvelot

Centre de Recherche en Informatique
Ecole des Mines de Paris
35, rue Saint Honoré, F-77305 Fontainebleau.

E-Mail: `talpin@ensmp.fr` and `jouvelot@ensmp.fr`

November 10th, 1993

[*] Current Address: Jean-Pierre Talpin, European Computer-Industry Research Center (ECRC GmbH), Arabella Straße 17, D-81925 München. E-mail: `jp@ecrc.de`

Abstract

The *type and effect discipline* is a new framework for reconstructing the principal type and the minimal effect of expressions in implicitly typed polymorphic functional languages that support imperative constructs. The type and effect discipline outperforms other polymorphic type systems. Just as types abstract collections of concrete values, *effects* denote imperative operations on regions. *Regions* abstract sets of possibly aliased memory locations.

Effects are used to control type generalization in the presence of imperative constructs while regions delimit observable side-effects. The observable effects of an expression range over the regions that are free in its type environment and its type; effects related to local data structures can be discarded during type reconstruction. The type of an expression can be generalized with respect to the type variables that are not free in the type environment or in the observable effect.

Introducing the type and effect discipline, we define both a dynamic and a static semantics for an ML-like language and prove that they are consistently related. We present a reconstruction algorithm that computes the principal type and the minimal observable effect of expressions. We prove its correctness with respect to the static semantics.

Contents

1 Introduction

Type inference [Milner, 1978] is the process that automatically reconstructs the type of expressions in programming languages. Polymorphic type inference in functional languages becomes problematic in the presence of imperative constructs and much investigations have been devoted to this issue [Tofte, 1987, Leroy & Weis, 1991, Wright, 1992].

Effect systems [Gifford & al., 1987, Lucassen, 1987, Lucassen & Gifford, 1988] aim at a safe integration of imperative programming features in functional languages. Just as types describe what expressions compute, effects describe how expressions compute and both can be statically reconstructed [Jouvelot & Gifford, 1991, Talpin & Jouvelot, Sept. 1992].

We introduce the *type and effect discipline*, a new framework for reconstructing the principal type and the minimal effect of expressions in implicitly typed polymorphic functional languages that support imperative constructs. Just as types structurally abstract collections of concrete values, *regions* abstract sets of possibly aliased memory locations while *effects* denote imperative operations on regions. Effects control type generalization in the presence of imperative constructs while regions are used to report their only observable side-effects.

The observable effects of an expression range over the regions that are free in its type environment and its type. Effects related to local data structures can be discarded during type reconstruction. The type of an expression can be generalized with respect to the variables that are neither free in the type environment nor in the observable effect.

In this paper, section 2 presents the related work. Then, we describe the dynamic (section 3) and static (sections 4 and 5) semantics of the language. We state that the static and dynamic semantics are consistent (section 6), and that our reconstruction algorithm, presented in sections 7 and 8, is correct with respect to the static semantics (section 9). We give some examples (sections 10 and 11) that show that our approach surpasses previous techniques before suggesting some extensions (section 12) and concluding (section 13).

2 Related Work

Short of the ad-hoc techniques used in the first type inference systems, the imperative type discipline [Tofte, 1987] is the classical way to deal with the problem of type generalization for polymorphic functional languages in the presence of non referentially transparent constructs. Its extension, based on weak type variables, is used in the implementation of Standard ML [Appel & Mac Queen, 1990]. A different approach, suggested in [Leroy & Weis, 1991], consists in labeling the type of each function with the set of the types of the value identifiers that occur in its body, and then to track the *dangerous* type variables of references.

All those approaches build conservative approximations of value types that may be accessible from the global store by relating the typing of references to syntactic information. A more intuitive and integrated approach is to infer a more semantically meaningful information by recording and and keeping track of the types of values referenced in the store, as soon as the store is expanded and as long as its locations are used. The quest for such a type system has for long been the subject of many investigations [Damas, 1985, O'Toole, 1989, Wright, 1992]. Effect inference allows us to approximate the store by regions and types and, as such, can be used to decide when to perform type generalization.

The FX system [Lucassen & Gifford, 1988, Gifford & al., 1987] suggests a static semantics for polymorphic type and effect checking. In [Jouvelot & Gifford, 1991], the authors show that effect reconstruction can be seen as a constraint satisfaction problem. However, the exact matching of effects required by the static semantics, together with the use of explicit polymorphism, imply the non-existence of syntactic principal types; it also somewhat limits the kind of accepted programs. We present in [Talpin & Jouvelot, Sept. 1992] an algorithm that computes the maximal type and the minimal effect of expressions, using subsumption on effects to overcome this particular problem of effect matching.

In the *type and effect discipline*, we apply the technique of type, region and effect inference to the problem of typing references in ML-like languages in the presence of polymorphic `let` constructs. We determine the principal type and the minimal observable effects of expressions. We use effect information to perform type generalization. By using effect information together with an observation criterion, our type system is able to precisely delimit the scope of side-effecting operations, thus allowing type generalization to be performed in `let` expressions in a more efficient and uniform way than previous systems. It is shown with some simple examples (section 10) that our system improves over earlier type generalization policies for ML-like languages.

3 A Core Language and its Semantics

Reasoning on the complete definition of a functional language such as Standard ML or FX would have been complex and tedious. In order to simplify the presentation and to ease the formal reasoning, this section introduces a core language. It is an attempt to trade between integrating the principal features of functional and imperative programming, and being simple. This section introduces its syntax and its dynamic semantics together with a series of conventions and notations that are used in this paper.

3.1 Syntax

The expressions of the language, written e possibly with a prime or a subscript, are the elements of the term algebra *Exp* generated by the grammar described below. It uses enclosing parentheses in the reminiscence of Scheme [Rees & al., 1988].

e ::=	x |	value identifier
	(op e) |	operation
	(e e′) |	application
	(lambda (x) e) |	abstraction
	(let (x e) e′)	lexical value binding
op ::=	new | get | set	operations on references

Syntax

In this grammar, x and f range over a countable set of identifiers. The form (e e′) stands for the application of a function e to an argument e′. The form (op e) applies the primitive operation op to the argument e. The expression (lambda (x) e) is the so-called lambda-abstraction that defines the first-class function whose parameter is x and whose result is the value of e.

3.2 Store operations

The arithmetic operations over integers: + and -, the boolean operations: and and or, or even the if construct are typically represented by operators op, because their meaning cannot be explained easily by abstractions and applications. Store operations can also be defined by operators. They operate on reference values, which are indirection cells that can be dynamically allocated, read and written in place.

The operation (new e) initializes a fresh reference to the value of the expression e. The operation (get e) gets the value referenced by the pointer returned by e. The operation ((set e) e′) modifies the content of the reference returned by e and sets it to the value of e′. We use the convention that set returns the unit value u.

3.3 Formulation of the Dynamic Semantics

In this section, we define the dynamic semantics of our language. The dynamic semantics specifies the meaning of expressions. It is defined by an evaluation mechanism that relates expressions to values. To express this relation, we use the formalism of relational semantics [Plotkin, 1981, Kahn, 1988]. It consists of a predicate between expressions and values defined by a set of axioms and inference rules called evaluation judgements. An evaluation judgment tells whether an expression evaluates to a given result.

3.4 Semantic Objects

We present the semantics objects on which the predicate of evaluation is defined. These semantic objects include values, environments, stores and traces.

Values are either the command value u, reference values l or closures c. A closure $(\mathtt{x}, \mathtt{e}, E)$ is composed of a value identifier $\mathtt{x}$, its formal parameter, an expression $\mathtt{e}$, its body, and the environment E where it is defined.

An environment E is represented by a finite map from identifiers to values. In an environment E, we assume that all identifiers are distinct. The empty mapping is written $\{\}$. The domain of the mapping E is written $Dom(E)$ and its range $Im(E)$. If $\mathtt{x}$ belongs to $Dom(E)$, we write $E(\mathtt{x})$ for the value associated with $\mathtt{x}$ in E. Finally, we write $E_{\mathtt{x}}$ for the exclusion of $\mathtt{x}$ from E and $E_{\mathtt{x}} + \{\mathtt{x} \mapsto v\}$ for the extension of E to the mapping of $\mathtt{x}$ to v.

$$
\begin{array}{rcll}
v & \in & \mathit{Value} & = \{u\} + \mathit{Ref} + \mathit{Closure} & \text{values} \\
c & \in & \mathit{Closure} & = \mathit{Id} \times \mathit{Exp} \times \mathit{Env} & \text{closures} \\
l & \in & \mathit{Ref} & & \text{locations} \\
E & \in & \mathit{Env} & = \mathit{Id} \overset{\mathit{fin}}{\to} \mathit{Value} & \text{environments} \\
s & \in & \mathit{Store} & = \mathit{Ref} \overset{\mathit{fin}}{\to} \mathit{Value} & \text{stores}
\end{array}
$$

Values, Environments, Stores and Traces

The presence of references requires the introduction of a notion of state in the dynamic semantics: the store. The store changes during the evaluation of a program and it tells the current contents of all initialized references. We assume that we are given a countable set *Ref* of locations l. Then, a store s is represented by a finite map from references, or locations, in *Ref* to values. Thus, we use for stores the same notations than for environments.

3.5 Axioms and Rules of the Dynamic Semantics

We present, in the vein of [Tofte, 1987, Milner, 1991], the set of rules that inductively defines the predicate of evaluation $s, E \vdash \mathtt{e} \rightarrow v, s'$ on the structure of expressions. Given a store s and an environment E, the predicate $s, E \vdash \mathtt{e} \rightarrow v, s'$ associates each expression $\mathtt{e}$ with a value v and a new store s'.

$$s, E \vdash \mathtt{x} \rightarrow E(\mathtt{x}), s \qquad \text{(var)}$$

$$s, E \vdash \mathtt{(lambda\ (x)\ e)} \rightarrow (\mathtt{x}, \mathtt{e}, E_{\mathtt{X}}), s \qquad \text{(abs)}$$

$$\frac{s, E \vdash \mathtt{e} \rightarrow v, s' \qquad s', E_{\mathtt{X}} + \{\mathtt{x} \mapsto v\} \vdash \mathtt{e}' \rightarrow v', s''}{s, E \vdash \mathtt{(let\ (x\ e)\ e')} \rightarrow v', s''} \qquad \text{(let)}$$

$$\frac{s, E \vdash \mathtt{e} \rightarrow (\mathtt{x}, \mathtt{e}'', E'), s' \quad s', E \vdash \mathtt{e}' \rightarrow v', s'' \quad s'', E' + \{\mathtt{x} \mapsto v'\} \vdash \mathtt{e}'' \rightarrow v'', s'''}{s, E \vdash \mathtt{(e\ e')} \rightarrow v'', s'''} \qquad \text{(app)}$$

Dynamic Semantics

The axiom (var) states that an identifier $\mathtt{x}$ evaluates to the value $E(\mathtt{x})$ bound to it in the environment E, provided that this identifier $\mathtt{x}$ belongs to the domain of E. Otherwise, the expression $\mathtt{x}$ has no meaning. By the axiom (abs), a function definition evaluates to a closure.

As stated by the rule (let), a let binding evaluates the first argument $\mathtt{e}$ to a value v, binds it to the identifier $\mathtt{x}$, and then evaluates its second argument $\mathtt{e}'$ in the environment E extended with $\{\mathtt{x} \mapsto v\}$. The result v' is the result of the let expression. In the case that the evaluation of the first argument does not succeed, the evaluation of the let expression is not defined.

The rule of application (app) is more complex. First, the expression $\mathtt{e}$ must evaluate to a closure $(\mathtt{x}, \mathtt{e}'', E')$. Then, the argument $\mathtt{e}'$ must evaluate to a value v'. Finally, the function body $\mathtt{e}''$ must evaluate to a value v'' with the environment E', captured in the closure, extended with the formal parameter $\mathtt{x}$ bound to v'.

3.6 Dynamic Semantics of Store Operations

Now, we can give the relational semantics for the operations on references. The semantics describes how the store is modified by the evaluation of expressions.

$$\frac{s, E \vdash \mathtt{e} \rightarrow v, s' \qquad l \notin Dom(s')}{s, E \vdash \mathtt{(new\ e)} \rightarrow l, s' + \{l \mapsto v\}} \qquad \text{(new)}$$

$$\frac{s, E \vdash \mathtt{e} \rightarrow l, s' \qquad l \in Dom(s')}{s, E \vdash \mathtt{(get\ e)} \rightarrow s'(l), s'} \qquad \text{(get)}$$

$$\frac{s, E \vdash \mathtt{e} \rightarrow l, s' \qquad s', E \vdash \mathtt{e}' \rightarrow v, s''}{s, E \vdash \mathtt{((set\ e)\ e')} \rightarrow u, s''_l + \{l \mapsto v\}} \qquad \text{(set)}$$

Dynamic Semantics of Store Operations

The rule (new) of reference initialization first evaluates the initial value v of the reference and then picks a fresh location l. This very step is non-deterministic but all choices of l are equivalent modulo a renaming of the locations in s'. The second step is then to extend the store with the binding of l to v and to return l as the value of the expression. The rule (get) evaluates its argument e to a location l, then returns the value v stored at this location in the store s. Finally, by the rule (set), the assignment operator evaluates its first argument e to a location l and its second argument $\mathtt{e}'$ to a value v. Then, it updates the store at the location l, substituting the previous value by v. Note that, by definition of the rule (new), l must be in s when e evaluates to l.

4 Static Semantics

In this section, we present the static semantics of our language. We are first going to equip the language with a type system. Then we will give the inference rules of the static semantics. The rules of the static semantics associate the expressions of the language with their type and effect, in the same way as the rules of the dynamic semantics associate expressions with values. We begin by defining the term algebra for the three basic kinds of semantic objects: regions, effects and types.

$$\begin{array}{lll}
\rho & ::= r \mid \varrho & \text{regions} \\
\sigma & ::= \emptyset \mid init(\rho, \tau) \mid read(\rho, \tau) \mid write(\rho, \tau) \mid \varsigma \mid \sigma \cup \sigma & \text{effects} \\
\tau & ::= unit \mid \alpha \mid ref_\rho(\tau) \mid \tau \xrightarrow{\sigma} \tau & \text{types}
\end{array}$$

Static Semantics Objects

The domain of regions ρ is the disjoint union of a countable set of constants r and variables ϱ. Every location corresponds to a given region in the static semantics. A region abstracts the memory locations that will be initialized at a given program point at runtime.

Effects σ can either be the constant $\emptyset$, that represents the absence of effects, effect variables ς, or store effects $\mathit{init}(\rho, \tau)$, $\mathit{read}(\rho, \tau)$ and $\mathit{write}(\rho, \tau)$, that approximate memory side-effects on the region ρ of references to values of type τ.

The effect $\mathit{init}(\rho, \tau)$ (which could also be named *alloc*) statically records the allocation of a reference in a region ρ and its initialization to a value of type τ. The effects $\mathit{read}(\rho, \tau)$ and $\mathit{write}(\rho, \tau)$ keep track of how and when the references of a given region are used.

We define the range of an effect σ, written $\mathit{Rng}(\sigma)$, is the set of pairs (ρ, τ) such that either $\mathit{init}(\rho, \tau)$, $\mathit{read}(\rho, \tau)$ or $\mathit{write}(\rho, \tau)$ is in σ. We write $\mathit{Regs}(\sigma)$ the set of regions ρ such that (ρ, τ) is in the range of σ.

Effects can be gathered together with the infix operator $\cup$ that denotes the union of effects; effects define a set algebra. The equality on effects is thus defined modulo associativity, commutativity and idempotence with $\emptyset$ as the neutral element. We define the set-inclusive relation $\supseteq$ of subsumption on effects: $\sigma \supseteq \sigma'$, or $\sigma' \subseteq \sigma$, if and only if there exists an effect σ'' such that $\sigma = \sigma' \cup \sigma''$.

The domain of types τ is composed of the constant *unit*, which denotes the type of the trivial value of commands in ML (like the type named *comm* in Algol), type variables α, reference types $\mathit{ref}_{\rho}(\tau)$ in region ρ to values of type τ, function types $\tau \xrightarrow{\sigma} \tau'$ from τ to τ' with a *latent effect* σ. The latent effect of a function encapsulates the side-effects of its body and is the effect incurred when the function is applied.

4.1 Free Variables and Substitutions

We have defined three kinds of variables: type variables, region variables and effect variables. When it is not necessary to specify if a variable represents a type, a region or an effect, we note it v. Also, we adopt the vector notation $\vec{v}$ to represent sequences of terms, such as sequences of variables v.

We write $\mathit{fv}(\tau)$ for the set of free type, region and effect variables in τ. This definition extends pointwise to regions and effects. The function fr is defined in a similar manner and computes the set of region constants and variables free in type and effect terms.

$$\begin{aligned} fv(unit) &= \emptyset \\ fv(\alpha) &= \{\alpha\} \\ fv(ref_{\rho}(\tau)) &= fv(\rho) \cup fv(\tau) \\ fv(\tau \xrightarrow{\sigma} \tau') &= fv(\tau) \cup fv(\tau') \cup fv(\sigma) \\ fv(r) &= \emptyset \\ fv(\varrho) &= \{\varrho\} \end{aligned}$$

$$\begin{aligned} fv(\emptyset) &= \emptyset \\ fv(init(\rho, \tau)) &= fv(\rho) \cup fv(\tau) \\ fv(read(\rho, \tau)) &= fv(\rho) \cup fv(\tau) \\ fv(write(\rho, \tau)) &= fv(\rho) \cup fv(\tau) \\ fv(\sigma \cup \sigma') &= fv(\sigma) \cup fv(\sigma') \end{aligned}$$

Free Variables

Substitutions θ map type variables α to types τ, region variables ϱ to regions ρ and effect variables ς to effects σ. We write $\theta \circ \theta'$ for the composition of the substitution θ and θ', so that $\theta \circ \theta'(v) = \theta(\theta'(v))$. The identity is written *Id*.

4.2 Type Schemes and Environments

We use type schemes, introduced by [Milner, 1978], to generically represent the different types of an expression. A type scheme $\forall\vec{v}.\tau$ consists of a type τ which is universally quantified over a sequence $\vec{v}$ of type variables, region variables and effect variables. A type τ' is an instance of a type scheme $\forall\vec{v}.\tau$, written $\tau' \preceq \forall\vec{v}.\tau$, if the variables $\vec{v}$ can be substituted by some substitution θ so that $\tau' = \theta\tau$.

In the sequence $\vec{v}$, the variables are assumed to be distinct and their order of occurrence is not significant. When that sequence is empty, we do not distinguish τ from $\forall.\tau$. We identify type schemes that differ only by a renaming of their quantified variables or that differ by the introduction or elimination of quantified variables that are not free in the body of the type scheme.

The context in which an expression is associated with a type and an effect is represented by a type environment $\mathcal{E}$ which maps value identifiers to type schemes. The definitions of free variables and free regions are extended to type schemes by $fv(\forall\vec{v}.\tau) = fv(\tau) \setminus \vec{v}$ and to environments $\mathcal{E}$ by considering that a type variable is free in $\mathcal{E}$ if and only if it is free in $\mathcal{E}(\mathtt{x})$ for some value identifier $\mathtt{x}$ in $Dom(\mathcal{E})$. Substitutions are also extended to type schemes $\forall\vec{v}.\tau$ by using alpha-renaming of quantified variables in type schemes to avoid capture of bound variables.

We extend substitutions to type schemes $\theta(\forall\vec{v}.\tau) = \forall\vec{v'}.\theta''(\theta'\tau)$ using the renaming $\theta' = \{\vec{v} \mapsto \vec{v'}\}$ of the bound variables $\vec{v}$ by fresh variables $\vec{v'}$, which are not free in τ and $\theta\tau$. The substitution $\theta''v$ of a variable v by θ'' is defined as v if $v \in \vec{v'}$ and as θv otherwise. The image $\theta(\mathcal{E})$ of a type environment $\mathcal{E}$ by a substitution θ is defined by $(\theta\mathcal{E})(\mathtt{x}) = \theta(\mathcal{E}(\mathtt{x}))$ for every `x` in $Dom(\mathcal{E})$. Our extension of substitution on type schemes and environments satisfies the following lemma.

LEMMA 1 (SUBSTITUTION AND INSTANTIATION) *If $\tau_1 \preceq \theta(\forall\vec{v}.\tau)$ then there exist θ_2 and $\tau_2 \preceq \forall\vec{v}.\tau$ such that $\tau_1 = \theta_2\tau_2$.*

Proof By hypothesis, $\tau_1 \preceq \theta(\forall\vec{v}.\tau)$. By definition, $\theta(\forall\vec{v}.\tau) = \forall\vec{v'}.(\theta''(\theta'\tau))$ where the $\vec{v'}$ are neither free in $\theta\tau$ nor τ, where $\theta' = \{\vec{v} \mapsto \vec{v'}\}$ and where θ'' is defined by θv for every $v \in fv(\tau) \setminus \vec{v}$. By definition of $\preceq$, there exists θ'_1 defined on $\vec{v'}$ such that $\tau_1 = \theta'_1(\theta''(\theta'\tau))$. Let $\theta'_2 = \{\vec{v} \mapsto \theta'_1\vec{v'}\}$ and $\tau_2 = \theta'_2\tau$. We have $\tau_2 \preceq \forall\vec{v}.\tau$. Let θ_2 the restriction of θ on $fv(\tau_2) \setminus \vec{v}$. It verifies that $\tau_1 = \theta_2\tau_2$ □

4.3 Type Generalization

The generalization $Gen(\mathcal{E},\sigma)(\tau)$ of a type τ is performed at `let` boundaries on some of the type, region and effect variables $\vec{v}$ that occur free in τ. A variable cannot be generalized when it is either free in the type environment $\mathcal{E}$ or present in the effect σ.

$$Gen(\mathcal{E},\sigma)(\tau) \;=\; \text{let } \vec{v} = fv(\tau) \setminus (fv(\mathcal{E}) \cup fv(\sigma)) \text{ in } \forall\vec{v}.\tau$$

The first condition is common for purely functional languages [Gordon & al., 1979]. As for the second, just as types are bound to identifiers in the type environment, types are bound to regions in the reconstructed effects. Thus, when these regions are observable from the context, i.e. in the type environment $\mathcal{E}$ or the type τ of the returned value, those types cannot be generalized.

4.4 Rules of the Static Semantics

The next figure summarizes the rules of our static semantics. We formulate type and effect inference by a deductive proof system that assigns a type and an effect to every expression of the language. The context in which an expressions is associated with a type and an effect is represented by a type environment $\mathcal{E}$ which maps value identifiers to types. Deductions produce conclusions of the form $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$ which are called typing judgments and read "in the type environment $\mathcal{E}$ the expression `e` has type τ and effect σ".

$$\frac{\tau \preceq \mathcal{E}(\texttt{x})}{\mathcal{E} \vdash \texttt{x} : \tau, \emptyset} \qquad \text{(var)}$$

$$\frac{\mathcal{E} \vdash \texttt{e} : \tau, \sigma \qquad \mathcal{E}_{\texttt{x}} + \{\texttt{x} \mapsto \mathit{Gen}(\sigma, \mathcal{E})(\tau)\} \vdash \texttt{e}' : \tau', \sigma'}{\mathcal{E} \vdash \texttt{(let (x e) e}'\texttt{)} : \tau', \sigma \cup \sigma'} \qquad \text{(let)}$$

$$\frac{\mathcal{E}_{\texttt{x}} + \{\texttt{x} \mapsto \tau\} \vdash \texttt{e} : \tau', \sigma}{\mathcal{E} \vdash \texttt{(lambda (x) e)} : \tau \xrightarrow{\sigma} \tau', \emptyset} \qquad \text{(abs)}$$

$$\frac{\mathcal{E} \vdash \texttt{e} : \tau \xrightarrow{\sigma} \tau', \sigma' \qquad \mathcal{E} \vdash \texttt{e}' : \tau, \sigma''}{\mathcal{E} \vdash \texttt{(e e}'\texttt{)} : \tau', \sigma \cup \sigma' \cup \sigma''} \qquad \text{(app)}$$

Static Semantics

In the rules (var) and (let), the static semantics manipulates type schemes by using the mechanism of generalization and instantiation specified in the previous section. The communication of the effects from a function definition to a function application is best viewed in the rules of abstraction (abs) and application (app), which show the interesting interplay between types and effects. Via the abstraction rule, the effect of a lambda abstraction body is put inside the function type while, with the application rule, this embedded effect is extracted from the function type to be exercised at the point of call; effects flow from the points where functions are defined to the points where they are used.

4.5 Static Semantics of Store Operations

The store operations `new`, `get` and `set` have been defined by appropriate rules in the dynamic semantics. In the static semantics, they are best defined by using axioms.

$$\mathcal{E} \vdash \texttt{new} : \tau \xrightarrow{\sigma \cup \mathit{init}(\rho,\tau)} \mathit{ref}_{\rho}(\tau), \emptyset$$

$$\mathcal{E} \vdash \texttt{get} : \mathit{ref}_{\rho}(\tau) \xrightarrow{\sigma \cup \mathit{read}(\rho,\tau)} \tau, \emptyset$$

$$\mathcal{E} \vdash \texttt{set} : \mathit{ref}_{\rho}(\tau) \xrightarrow{\sigma} \tau \xrightarrow{\sigma' \cup \mathit{write}(\rho,\tau)} \mathit{unit}, \emptyset$$

Static Semantics for Imperative Operations

These three axioms specify the types that can be assigned to the identifiers `new`, `get` and `set` that implement store operations. For instance, the axiom for the identifier `new` reads: for any environment $\mathcal{E}$, type τ, region ρ and effect σ, the type of `new` is a function from objects of type τ to references $\mathit{ref}_\rho(\tau)$ that has at least the effect $\mathit{init}(\rho, \tau)$ of initializing a reference in the region ρ. The additional effects σ and σ' are used here to allow possible coercions to be performed.

4.6 Observation Criterion

The latent effect of a function derives statically from the state transforming operations that the expression it abstracts performs when it is executed. For example, store operations comprise the initialization, reading and writing of references which are approximated by regions.

Even if a value expression performs certain operations on the store, one may be able to detect that those operations cannot interfere with other expressions. This is the case when the regions over which the effect ranges are unreferenced in the rest of the program. If this is the case, then we shall mask effects which derive from those operations.

In our type and effect inference system, the static determination of the lexical scope of data regions is implemented by an observation criterion. The observation of effects consists of selecting an effect that ranges over regions that refer to data accessible in the environment of an expression or in its value. The accessible data are abstracted by the free regions of the type environment and of the value type of the expression.

In the static semantics, only side-effects that can affect the typing context of an expression, i.e. its type environment $\mathcal{E}$ and its value type τ, are worth reporting. The other effects of σ refer to local references that are freshly created and not exported from the expression e.

$$\frac{\mathcal{E} \vdash \mathsf{e} : \tau, \sigma \qquad \sigma' \supseteq \mathit{Observe}(\mathcal{E}, \tau)(\sigma)}{\mathcal{E} \vdash \mathsf{e} : \tau, \sigma'} \ \text{(sub)}$$

Observation and Subsumption Rule

The observation criterion is specified by the rule (sub) which tells that an expression e has any effect σ' bigger than the observable effects that can be inferred for it. $\mathit{Observe}(\mathcal{E}, \tau)(\sigma)$ is the set of observable effects of σ.

$$\begin{aligned} \mathit{Observe}(\mathcal{E},\tau)(\sigma) \quad &= \{\mathit{init}(\rho,\tau'), \mathit{read}(\rho,\tau'), \mathit{write}(\rho,\tau') \in \sigma \mid \rho \in \mathit{fr}(\mathcal{E}) \cup \mathit{fr}(\tau)\} \\ &\cup \{\varsigma \in \sigma \mid \varsigma \in \mathit{fv}(\mathcal{E}) \cup \mathit{fv}(\tau)\} \end{aligned}$$

Observable Effects

The observable effects are the effect variables ς that occur free in τ or $\mathcal{E}$ and the effects of the form $\mathit{init}(\rho,\tau')$, $\mathit{read}(\rho,\tau')$ and $\mathit{write}(\rho,\tau')$ where ρ occurs free in τ or in $\mathcal{E}$. We write $\mathit{Observe}(\tau)(\sigma)$ for $\mathit{Observe}(\{\},\tau)(\sigma)$ and $\mathit{Observe}(\sigma)(\sigma')$ for $\mathit{Observe}(\mathit{unit} \xrightarrow{\sigma} \mathit{unit})(\sigma')$. The function *Observe* has the following formal property which is widely used in the rest of this paper.

Lemma 2 (*Observe* and free variables) *If* $\sigma' = \mathit{Observe}(\tau)(\sigma)$ *then* $\mathit{fr}(\tau) \cap \mathit{Regs}(\sigma \setminus \sigma') = \emptyset$ *and* $\mathit{fr}(\sigma') \cap \mathit{Regs}(\sigma \setminus \sigma') = \emptyset$*.*

Proof By hypothesis, $\sigma' = \mathit{Observe}(\tau)(\sigma)$. By definition of *Observe* and for any $\rho \in \mathit{Regs}(\sigma \setminus \sigma')$, we have that $\rho \notin \mathit{fr}(\tau) \cup \mathit{fr}(\sigma')$. Thus, $(\mathit{fr}(\tau) \cup \mathit{fr}(\sigma')) \cap \mathit{Regs}(\sigma \setminus \sigma') = \emptyset$ as expected □

With the lemma 3, we show that observation is conserved under substitution, in that combinations of substitutions θ to the function *Observe* can be compared with the application of the function *Observe* to substituted terms.

Lemma 3 (*Observe* and substitution) $\theta(\mathit{Observe}(\tau)(\sigma)) \subseteq \mathit{Observe}(\theta\tau)(\theta\sigma)$ *for any* θ*.*

Proof Let us write $\sigma'' = \mathit{Observe}(\theta\tau)(\theta\sigma)$. We proceed by case analysis. For every $\varsigma \in \sigma'$ and by definition of *Observe*, we have $\varsigma \in \mathit{fv}(\tau)$ and thus $\sigma'' \supseteq \theta\varsigma$. For every $\mathit{init}(\rho,\tau') \in \sigma'$ (respectively, $\mathit{read}(\rho,\tau')$ and $\mathit{write}(\rho,\tau')$) and by definition of *Observe*, we have that $\rho \in \mathit{fr}(\tau)$ and thus $\theta\rho \in \mathit{fr}(\theta\tau)$. This implies that $\mathit{init}(\theta\rho,\theta\tau') \in \sigma''$. This proves that $\sigma'' \supseteq \theta\sigma'$ □

Note, however, that the containment is not proper. An example where we do not have $\theta(\mathit{Observe}(\tau)(\sigma)) = \mathit{Observe}(\theta\tau)(\theta\sigma)$ is $\tau = \mathit{ref}_{\rho_1}(\mathit{int})$, $\sigma = \mathit{init}(\rho_1, \mathit{int}) \cup \mathit{init}(\rho_2, \mathit{unit})$ and $\theta = \{\rho_2 \mapsto \rho_1\}$.

5 Formal Properties of the Static Semantics

The lemma of substitution is used both in the proof of consistency and in the proofs of correctness for the reconstruction algorithm.

Lemma 4 (Substitution) *If* $\mathcal{E} \vdash \mathsf{e} : \tau, \sigma$ *then* $\theta\mathcal{E} \vdash \mathsf{e} : \theta\tau, \theta\sigma$ *for any substitution* θ*.*

Proof The proof is by induction on the typing derivation.

Case of (var) By hypothesis, we have $\mathcal{E} \vdash \mathtt{x} : \tau, \emptyset$. By definition of the rule (var), $\tau \preceq \mathcal{E}(\mathtt{x})$. By the lemma 1, there exists $\tau' \preceq \theta\mathcal{E}(\mathtt{x})$ and θ' such that $\theta\tau = \theta'\tau'$. Thus, by definition of the rule (var), $\theta\mathcal{E} \vdash \mathtt{x} : \theta\tau, \emptyset$.

Case of (abs) By hypothesis, we have $\mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \tau \xrightarrow{\sigma} \tau', \emptyset$. By definition of the rule (abs), $\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e} : \tau', \sigma$. By induction hypothesis on $\mathtt{e}$, $\theta(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\}) \vdash \mathtt{e} : \theta\tau', \theta\sigma$ for any substitution θ. By definition of the rule (abs), $\theta\mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \theta(\tau \xrightarrow{\sigma} \tau'), \emptyset$.

Case of (app) By hypothesis, $\mathcal{E} \vdash \mathtt{(e\ e')} : \tau', \sigma \cup \sigma' \cup \sigma''$. By definition of the rule (app), $\mathcal{E} \vdash \mathtt{e} : \tau \xrightarrow{\sigma''} \tau', \sigma$ and $\mathcal{E} \vdash \mathtt{e'} : \tau, \sigma'$. By induction hypothesis on $\mathtt{e}$ and $\mathtt{e'}$, $\theta\mathcal{E} \vdash \mathtt{e} : \theta(\tau \xrightarrow{\sigma''} \tau'), \theta\sigma$ and $\theta\mathcal{E} \vdash \mathtt{e'} : \theta\tau, \theta\sigma'$ for any substitution θ. By the definition of the rule (app), we conclude that $\theta\mathcal{E} \vdash \mathtt{(e\ e')} : \theta\tau', \theta(\sigma \cup \sigma' \cup \sigma'')$ for any substitution θ.

Case of (let) By hypothesis $\mathcal{E} \vdash \mathtt{(let\ (x\ e_1)\ e_2)} : \tau, \sigma_1 \cup \sigma_2$. By definition of the rule (*let*), we have

$$\mathcal{E} \vdash \mathtt{e}_1 : \tau_1, \sigma_1 \quad \text{and} \quad \mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \mathit{Gen}(\sigma_1, \mathcal{E})(\tau_1)\} \vdash \mathtt{e}_2 : \tau, \sigma_2$$

Let $\forall\vec{v}.\tau_1$ be $\mathit{Gen}(\sigma_1, \mathcal{E})(\tau_1)$. For any substitution θ, let us consider fresh $\vec{v'}$ and define θ' as the extension of $\theta_{\vec{v}}$ with $\{\vec{v} \mapsto \vec{v'}\}$. By the definition of *Gen*, we have that $\theta\mathcal{E} = \theta'\mathcal{E}$ and $\theta\sigma = \theta'\sigma$. By definition of θ', $\theta'(\forall\vec{v}.\tau_1) = \forall\vec{v'}.(\theta'\tau_1)$. Thus:

$$\theta(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \mathit{Gen}(\sigma_1, \mathcal{E})(\tau_1)\}) = \theta\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \mathit{Gen}(\theta\sigma_1, \theta\mathcal{E})(\theta'\tau_1)\}$$

Using the induction hypothesis on $\mathtt{e}_1$ with θ', we get that $\theta'\mathcal{E} \vdash \mathtt{e}_1 : \theta'\tau_1, \theta'\sigma_1$ and thus, by definition of θ', $\theta\mathcal{E} \vdash \mathtt{e}_1 : \theta'\tau_1, \theta\sigma_1$. By induction hypothesis on $\mathtt{e}_2$, we get:

$$\theta(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \mathit{Gen}(\sigma_1, \mathcal{E})(\tau_1)\} \vdash \mathtt{e}_2 : \theta\tau, \theta\sigma_2$$

which is equivalent to $\theta\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \mathit{Gen}(\theta\sigma_1, \theta\mathcal{E})(\theta'\tau_1)\} \vdash \mathtt{e}_2 : \theta\tau, \theta\sigma_2$. By definition of the rule (let), we can then conclude that $\theta\mathcal{E} \vdash \mathtt{(let\ (x\ e_1)\ e_2)} : \theta\tau, \theta\sigma$

Case of (sub) By hypothesis, $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$. By definition of the rule (sub), this requires that there exists σ' such that

$$\mathcal{E} \vdash \mathtt{e} : \tau, \sigma' \quad \text{and} \quad \sigma \supseteq \mathit{Observe}(\mathcal{E}, \tau)(\sigma')$$

Let θ be any substitution. By induction hypothesis on the derivation, we have that

$$\theta\mathcal{E} \vdash \mathsf{e} : \theta\tau, \theta\sigma'$$

Let $\sigma'_1 = \mathit{Observe}(\mathcal{E}, \tau)(\sigma')$ and $\sigma'_2 = \sigma' \setminus \sigma'_1$. Let us define the substitution θ' that maps effect variables in σ'_2 to $\emptyset$ and regions in $\mathit{Regs}(\sigma'_2)$ to fresh regions, not free in $\theta\mathcal{E}$, $\theta\tau$ and $\theta\sigma'_1$. Using the lemma 2, $\mathit{Regs}(\sigma'_2) \cap (\mathit{fr}(\tau) \cup \mathit{fr}(\mathcal{E}) \cup \mathit{fr}(\sigma'_1)) = \emptyset$. Thus, for the substitution $\theta \circ \theta'$, we have

$$\theta\mathcal{E} \vdash \mathsf{e} : \theta\tau, \theta(\sigma'_1 \cup \theta'\sigma'_2)$$

By definition of the rule (sub),

$$\theta\mathcal{E} \vdash \mathsf{e} : \theta\tau, \mathit{Observe}(\theta\mathcal{E}, \theta\tau)(\theta(\sigma'_1 \cup \theta'\sigma'_2))$$

By definition of *Observe*, $\mathit{Observe}(\theta\mathcal{E}, \theta\tau)(\theta(\theta'\sigma'_2)) = \mathit{Observe}(\theta\mathcal{E}, \theta\tau)(\theta'\sigma'_2) = \emptyset$. Thus,

$$\theta\mathcal{E} \vdash \mathsf{e} : \theta\tau, \mathit{Observe}(\theta\mathcal{E}, \theta\tau)(\theta\sigma'_1)$$

By definition of *Observe*, $\mathit{Observe}(\theta\mathcal{E}, \theta\tau)(\theta\sigma'_1) \subseteq \theta\sigma'_1$. By the rule (sub), $\theta\mathcal{E} \vdash \mathsf{e} : \theta\tau, \theta\sigma'_1$. Since $\sigma \supseteq \sigma'_1$, we have $\theta\sigma \supseteq \theta\sigma'_1$, and by the rule (sub),

$$\theta\mathcal{E} \vdash \mathsf{e} : \theta\tau, \theta\sigma \ \Box$$

6 Consistency of Dynamic and Static Semantics

We define a consistency judgment $s{:}\sigma, \mathcal{S} \models v{:}\tau$ which relates values and types according to a given store model $\mathcal{S}$, a store s and observable effects σ. The notion of store model $\mathcal{S}$ is defined below.

Definition 1 (Store Model) *A store model $\mathcal{S}$ is a finite mapping from locations l to pairs (ρ, τ) of regions ρ and types τ. We say that $\mathcal{S}'$ extends $\mathcal{S}$, written $\mathcal{S} \sqsubseteq \mathcal{S}'$, if and only if $\mathit{Dom}(\mathcal{S}) \subseteq \mathit{Dom}(\mathcal{S}')$ and for every $l \in \mathit{Dom}(\mathcal{S})$, $\mathcal{S}(l) = \mathcal{S}'(l)$.*

In the dynamic semantics, when an expression is evaluated, its initial store s possibly mutates to another, s'. Similarly, in the static semantics, the observable effects σ that correspond to the construction of the initial store s may be augmented with the effect σ', inferred for the evaluated expression. Similarly, the store model $\mathcal{S}$ must be updated to $\mathcal{S}'$. However, $\mathcal{S}'$ must agree with $\mathcal{S}$ on the locations l of its domain which refer to observable regions in σ. These considerations are formalized by the following definition 2.

DEFINITION 2 (EXTENSION) *$(\sigma',\mathcal{S}')$ extends $(\sigma,\mathcal{S})$, noted $(\sigma,\mathcal{S}) \sqsubseteq (\sigma',\mathcal{S}')$ or $(\sigma',\mathcal{S}') \sqsupseteq (\sigma,\mathcal{S})$, if and only if $\sigma \subseteq \sigma'$, $Dom(\mathcal{S}) \subseteq Dom(\mathcal{S}')$ and, for all $l \in Dom(\mathcal{S})$, if $\mathcal{S}(l) \in Rng(\sigma)$ or $\mathcal{S}'(l) \in Rng(\sigma)$ then $\mathcal{S}'(l) = \mathcal{S}(l)$. $(\sigma',\mathcal{S}')$ and $(\sigma,\mathcal{S})$ are equivalent, noted $(\sigma,\mathcal{S}) \simeq (\sigma',\mathcal{S}')$, if and only if $(\sigma,\mathcal{S}) \sqsubseteq (\sigma',\mathcal{S}')$ and $(\sigma',\mathcal{S}') \sqsubseteq (\sigma,\mathcal{S})$.*

The relation, presented in the definition 3 below, specifies the consistency between values and types according to observable effects. It refers to an effect which represents the effect of evaluating an expression to a value and the history of the effects that permitted the evaluation of its environment. Unobservable effects may however be needed to show the consistency between unused values, captured within closures, and the types assigned to them.

DEFINITION 3 (CONSISTENT VALUES AND TYPES) *Given the store s, the effect σ and the model $\mathcal{S}$, the consistency relation between a value v and a type τ, written $s{:}\sigma,\mathcal{S} \models v{:}\tau$, satisfies the following properties.*

$$\begin{array}{ll}
s{:}\sigma,\mathcal{S} \models u{:}unit & \\
s{:}\sigma,\mathcal{S} \models l{:}ref_\rho(\tau) \Leftrightarrow (\rho,\tau) \in Rng(\sigma),\ \mathcal{S}(l) = (\rho,\tau) \text{ and } s{:}\sigma,\mathcal{S} \models s(l){:}\tau & \\
s{:}\sigma,\mathcal{S} \models (\mathtt{x},\mathtt{e},E){:}\tau \Leftrightarrow & \text{there exist } \mathcal{E} \text{ such that } \mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \tau,\emptyset, \\
& \sigma' \text{ such that } Observe(\tau)(\sigma') = \emptyset \text{ and } Observe(\sigma)(\sigma') = \emptyset, \\
& \mathcal{S}' \text{ such that } (\sigma,\mathcal{S}) \simeq (\sigma,\mathcal{S}') \text{ and } s{:}\sigma \cup \sigma',\mathcal{S}' \models E{:}\mathcal{E}
\end{array}$$

We write $s{:}\sigma,\mathcal{S} \models v{:}\forall\vec{v}.\tau$ if and only if $s{:}\sigma,\mathcal{S} \models v{:}\theta\tau$ for any substitution θ defined on $\vec{v}$ and $s{:}\sigma,\mathcal{S} \models E{:}\mathcal{E}$ if and only if $Dom(E) = Dom(\mathcal{E})$ and $s{:}\sigma,\mathcal{S} \models E(\mathtt{x}){:}\mathcal{E}(\mathtt{x})$ for any $\mathtt{x} \in Dom(E)$.

It is shown in [Tofte, 1987] that such a structural property, between values and types, does not uniquely define a relation. Because functions can be recursively defined through references, it must be regarded as a fixed point equation. We must define the typing consistency relation as the maximal fixed point of the property defined in 3. This is done by considering the appropriate function $\mathcal{F}$ below.

DEFINITION 4 (F) *Let $\mathcal{R}$ be the set of all tuples $(s,\sigma,\mathcal{S},v,\tau)$. The function $\mathcal{F}$ is defined over the elements $\mathcal{Q}$ of $\mathcal{P}(\mathcal{R})$. The greatest fixed point of $\mathcal{F}$, $gfp(\mathcal{F}) = \cup\{\mathcal{Q} \subseteq \mathcal{R} \mid \mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})\}$, defines our relation: we write $s,\sigma,\mathcal{S} \models v{:}\tau$ if and only if $(s,\sigma,\mathcal{S},v,\tau) \in gfp(\mathcal{F})$.*

$$\begin{array}{l}
\mathcal{F}(\mathcal{Q}) = \{(s,\sigma,\mathcal{S},v,\tau) \mid \\
\qquad \text{if } v = u \text{ then } \tau = unit \\
\qquad \text{if } v = l \text{ then there exists } \tau' \text{ such that } \tau = ref_\rho(\tau'),\ \mathcal{S}(l) = (\rho,\tau') \in Rng(\sigma)
\end{array}$$

and $(s, \sigma, \mathcal{S}, s(l), \tau') \in \mathcal{Q}$

if $v = (\mathtt{x}, \mathtt{e}, E)$ *then*

there exist $\mathcal{E}$ *such that* $\mathcal{E} \vdash$ `(lambda (x) e)` $: \tau, \emptyset$, σ' *and* $\mathcal{S}'$ *such that* $\mathit{Observe}(\tau)(\sigma') = \emptyset$, $\mathit{Observe}(\sigma)(\sigma') = \emptyset$ *and* $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}')$ *and* $(s, \sigma \cup \sigma', \mathcal{S}', E(\mathtt{x}), \tau') \in \mathcal{Q}$ *for any* $\mathtt{x} \in \mathit{Dom}(E)$ *and* $\tau' \preceq \mathcal{E}(\mathtt{x})\}$

To admit a maximal fixed point $\mathit{gfp}(\mathcal{F})$, the function $\mathcal{F}$ must be monotonic. This is the first property that we thus have to verify.

Lemma 5 (Monotony of $\mathcal{F}$) *If* $\mathcal{Q} \subseteq \mathcal{Q}'$ *then* $\mathcal{F}(\mathcal{Q}) \subseteq \mathcal{F}(\mathcal{Q}')$*.*

Proof Let $\mathcal{Q}$ and $\mathcal{Q}'$ be two subsets of $\mathcal{S}$ such that $\mathcal{Q} \subseteq \mathcal{Q}'$. Let q be $(s, \sigma, \mathcal{S}, v, \tau)$ in $\mathcal{F}(Q)$. We prove that $q \in \mathcal{F}(\mathcal{Q}')$.

- If $v = u$ then, by the definition 4, $\tau = \mathit{unit}$ so that $q \in \mathcal{F}(\mathcal{Q}')$.

- If $v = l$ then, by the definition 4, $\tau = \mathit{ref}_\rho(\tau')$, $(\rho, \tau') \in \mathit{Rng}(\sigma)$, $\mathcal{S}(l) = (\rho, \tau')$ and $(s, \sigma, \mathcal{S}, s(l), \tau') \in \mathcal{Q}$. Since $\mathcal{Q} \subseteq \mathcal{Q}'$, $(s, \sigma, \mathcal{S}, s(l), \tau') \in \mathcal{Q}'$. By the definition 4, $q \in \mathcal{F}(\mathcal{Q}')$.

- If $v = (\mathtt{x}, \mathtt{e}, E)$ then, by the definition 4, there exist $\mathcal{E}$ such that $\mathcal{E} \vdash$ `(lambda (x) e)` $:$ $\tau, \emptyset$, σ' such that $\mathit{Observe}(\tau)(\sigma') = \emptyset$ and $\mathit{Observe}(\sigma)(\sigma') = \emptyset$, $\mathcal{S}'$ such that $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}')$ and $(s, \sigma \cup \sigma', \mathcal{S}', E(\mathtt{x}), \tau') \in \mathcal{Q}$ for any $\mathtt{x} \in \mathit{Dom}(E)$ and $\tau' \preceq \mathcal{E}(\mathtt{x})$. Since $\mathcal{Q} \subseteq \mathcal{Q}'$, $(s, \sigma \cup \sigma', \mathcal{S}', E(\mathtt{x}), \tau') \in \mathcal{Q}'$. By the definition 4, $q \in \mathcal{F}(\mathcal{Q}')$ $\Box$

The lemma 6 relates the definition 2 of the relation $\sqsubseteq$ with the consistency relation, given in the definition 3, according to the following respects.

Lemma 6 (Extension) *If* $s{:}\sigma, \mathcal{S} \models v{:}\tau$ *and* $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')$ *then* $s{:}\sigma', \mathcal{S}' \models v{:}\tau$*.*

Proof We consider the set $\mathcal{Q}{=}\{(s, \sigma', \mathcal{S}', v, \tau) \mid s{:}\sigma, \mathcal{S} \models v{:}\tau$ and $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')\ \}$. We show, by case analysis on the structure of v, that $q = (s, \sigma', \mathcal{S}', v, \tau)$ is in $\mathcal{F}(\mathcal{Q})$. Thus, $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$, showing that $\mathcal{Q}$ is $\mathcal{F}$-consistent.

- If $v = u$ then, by the definition 3, $\tau = \mathit{unit}$ so that $q \in \mathcal{F}(\mathcal{Q})$.

- If $v = l$ then, by the definition 3, $\tau = \mathit{ref}_\rho(\tau')$, $(\rho, \tau') \in \mathit{Rng}(\sigma)$, $\mathcal{S}(l) = (\rho, \tau')$ and $s{:}\sigma, \mathcal{S} \models s(l){:}\tau'$. Since $s{:}\sigma, \mathcal{S} \models s(l){:}\tau'$ and by definition of $\mathcal{Q}$, $(s, \sigma', \mathcal{S}', s(l), \tau') \in \mathcal{Q}$. Since $(\rho, \tau') \in \mathit{Rng}(\sigma)$, $\mathcal{S}(l) = (\rho, \tau')$ and by definition 2, $(\rho, \tau') \in \mathit{Rng}(\sigma')$ and $\mathcal{S}'(l) = (\rho, \tau')$. By the definition 4, $q \in \mathcal{F}(\mathcal{Q})$.

- If $v = (\mathtt{x}, \mathtt{e}, E)$ then, by the definition 3, there exist $\mathcal{E}$ such that $\mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \tau, \emptyset$, σ_1 such that $Observe(\tau)(\sigma_1) = \emptyset$ and $Observe(\sigma)(\sigma_1) = \emptyset$, $\mathcal{S}_1$ such that $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}_1)$ and $s{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models E(\mathtt{x}){:}\tau'$ for any $\mathtt{x} \in Dom(E)$ and $\tau' \preceq \mathcal{E}(\mathtt{x})$.

  Let us define θ on $Regs(\sigma_1)$ in such a way that the regions $Regs(\theta\sigma_1)$ are not free in τ, σ' and also $Im(\mathcal{S}')$. Let us write $\mathcal{E}' = \theta\mathcal{E}$, $\sigma_1' = \theta\sigma_1$ and $\mathcal{S}_1' = \theta\mathcal{S}_1$. By the lemma 4, $\mathcal{E}' \vdash \mathtt{(lambda\ (x)\ e)} : \tau, \emptyset$. By the lemma 1, for all $\tau'' \preceq \mathcal{E}'(\mathtt{x})$, there exists $\tau' \preceq \mathcal{E}(\mathtt{x})$ such that the restriction θ' of θ on the free variables of $\mathcal{E}$ verifies $\tau'' = \theta'\tau'$. By the lemma 8, $s{:}\sigma \cup \sigma_1', \mathcal{S}_1' \models E(\mathtt{x}){:}\tau''$ for any $\mathtt{x} \in Dom(E)$ and $\tau'' \preceq \mathcal{E}'(\mathtt{x})$. Let us define $\mathcal{S}_1''$ as follows,

  $$\forall l \in Dom(\mathcal{S}'),\ \mathcal{S}_1''(l) = \begin{cases} \mathcal{S}_1'(l), & \text{if } l \in Dom(\mathcal{S}) \text{ and } \mathcal{S}_1'(l) \in Rng(\sigma \cup \sigma_1') \\ \mathcal{S}'(l) & \text{otherwise} \end{cases}$$

  To prove that $(s, \sigma' \cup \sigma_1', \mathcal{S}_1'', E(\mathtt{x}), \tau'') \in \mathcal{Q}$, it remains to show that $(\sigma \cup \sigma_1', \mathcal{S}_1') \sqsubseteq (\sigma \cup \sigma_1', \mathcal{S}_1'')$. This requires that, for any $l \in Dom(\mathcal{S}_1')$, if $\mathcal{S}_1'(l) \in Rng(\sigma \cup \sigma_1')$ or $\mathcal{S}_1''(l) \in Rng(\sigma \cup \sigma_1')$ then $\mathcal{S}_1''(l) = \mathcal{S}_1'(l)$.

  - If $\mathcal{S}_1'(l) \in Rng(\sigma \cup \sigma_1')$ then, by definition of $\mathcal{S}_1''$, $\mathcal{S}_1''(l) = \mathcal{S}_1'(l)$.
  - If $\mathcal{S}_1''(l) \in Rng(\sigma \cup \sigma_1')$, we proceed by case analysis on the definition of $\mathcal{S}_1''$. First, if $l \in Dom(\mathcal{S})$ and $\mathcal{S}_1'(l) \in Rng(\sigma \cup \sigma_1')$ then $\mathcal{S}_1''(l) = \mathcal{S}_1'(l)$. Otherwise, either $l \notin Dom(\mathcal{S})$ [This is impossible, since we suppose that $l \in Dom(\mathcal{S}_1')$] or $\mathcal{S}_1'(l) \notin Rng(\sigma \cup \sigma_1')$. We show that this is impossible as well.

    By hypothesis, we have that $\mathcal{S}_1''(l) = \mathcal{S}'(l) \in Rng(\sigma \cup \sigma_1')$. Since, by definition of σ_1', $fr(Im(\mathcal{S}')) \cap Regs(\sigma_1') = \emptyset$, we must have $\mathcal{S}_1''(l) = \mathcal{S}'(l) \in Rng(\sigma)$. However, by hypothesis, we have that $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')$. This imposes that, if $\mathcal{S}(l)$ or $\mathcal{S}'(l)$ is in $Rng(\sigma)$, then $\mathcal{S}'(l) = \mathcal{S}(l)$. Since $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}_1)$ and θ does not affect σ, we must have $\mathcal{S}'(l) = \mathcal{S}(l) = \mathcal{S}_1(l) = \mathcal{S}_1'(l)$. However, we cannot have both $\mathcal{S}'(l) \in Rng(\sigma)$ and $\mathcal{S}_1'(l) \notin Rng(\sigma \cup \sigma_1')$.

  We have shown, by the definition 2, that $(\sigma \cup \sigma_1', \mathcal{S}_1') \sqsubseteq (\sigma' \cup \sigma_1', \mathcal{S}_1'')$. Since, for any $\mathtt{x} \in Dom(E)$ and $\tau'' \preceq \mathcal{E}'(\mathtt{x})$, $s{:}\sigma \cup \sigma_1', \mathcal{S}_1' \models E(\mathtt{x}){:}\tau''$, by definition of $\mathcal{Q}$, $(s, \sigma' \cup \sigma_1', \mathcal{S}_1'', E(\mathtt{x}), \tau'') \in \mathcal{Q}$.

  To show that $q \in \mathcal{F}(\mathcal{Q})$, it remains to prove that $(\sigma', \mathcal{S}') \simeq (\sigma', \mathcal{S}_1'')$. Since $Dom(\mathcal{S}') = Dom(\mathcal{S}_1'')$, this reduces to showing that, if $\mathcal{S}'(l) \in Rng(\sigma')$ or $\mathcal{S}_1''(l) \in Rng(\sigma')$, then $\mathcal{S}'(l) = \mathcal{S}_1''(l)$. We proceed by case analysis on σ and $\sigma' \setminus \sigma$.

- ○ If $\mathcal{S}''_1(l) \in \mathit{Rng}(\sigma)$ or $\mathcal{S}'(l) \in \mathit{Rng}(\sigma)$ then either $l \in \mathit{Dom}(\mathcal{S})$ or not. If $l \notin \mathit{Dom}(\mathcal{S})$ then, by definition of $\mathcal{S}''_1$, $\mathcal{S}''_1(l) = \mathcal{S}'(l)$. If $l \in \mathit{Dom}(\mathcal{S})$ then, since $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}_1)$, $\mathcal{S}(l) = \mathcal{S}_1(l)$. Since θ is not defined on σ, $\mathcal{S}_1(l) = \mathcal{S}'_1(l)$. Since, by hypothesis, $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')$, $\mathcal{S}(l) = \mathcal{S}'(l)$. Now, we know that $\mathcal{S}(l) = \mathcal{S}'(l) = \mathcal{S}_1(l) = \mathcal{S}'_1(l)$.

  To prove that $\mathcal{S}''_1(l) = \mathcal{S}'(l)$, it remains to show that $\mathcal{S}''_1(l) = \mathcal{S}'_1(l)$. By hypothesis, $l \in \mathit{Dom}(\mathcal{S})$ and either $\mathcal{S}''_1(l) \in \mathit{Rng}(\sigma)$ or $\mathcal{S}'(l) \in \mathit{Rng}(\sigma)$. If $\mathcal{S}''_1(l) \in \mathit{Rng}(\sigma)$ then $\mathcal{S}''_1(l) = \mathcal{S}'_1(l)$. If $\mathcal{S}'(l) \in \mathit{Rng}(\sigma)$ then, since $\mathcal{S}'(l) = \mathcal{S}'_1(l)$, $\mathcal{S}'_1(l) \in \mathit{Rng}(\sigma)$. Thus, by definition of $\mathcal{S}''_1$, $\mathcal{S}''_1(l) = \mathcal{S}'_1(l)$. It follows that $\mathcal{S}''_1(l) = \mathcal{S}'(l)$. We conclude that, if $\mathcal{S}'(l) \in \mathit{Rng}(\sigma)$, then $\mathcal{S}'(l) = \mathcal{S}''_1(l)$.

- ○ If $\mathcal{S}''_1(l) \in \mathit{Rng}(\sigma' \setminus \sigma)$ then, since $\mathit{Regs}(\sigma'_1) \cap \mathit{fr}(\sigma') = \emptyset$, $\mathcal{S}''_1(l) \notin \mathit{Rng}(\sigma \cup \sigma'_1)$. Thus, by definition of $\mathcal{S}''_1$, $\mathcal{S}''_1(l) = \mathcal{S}'(l)$. Similarly, if $\mathcal{S}'(l) \in \mathit{Rng}(\sigma' \setminus \sigma)$ then, since $\mathit{Regs}(\sigma'_1) \cap \mathit{fr}(\sigma') = \emptyset$, $\mathcal{S}'(l) \notin \mathit{Rng}(\sigma \cup \sigma'_1)$. Thus, either $l \notin \mathit{Dom}(\mathcal{S})$ and then $\mathcal{S}'(l) = \mathcal{S}''_1(l)$, or $l \in \mathit{Dom}(\mathcal{S})$ and then, since $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')$, $\mathcal{S}(l) \notin \mathit{Rng}(\sigma)$. Thus, $\mathcal{S}'_1(l) \notin \mathit{Rng}(\sigma)$ and, by definition of $\mathcal{S}''_1$, $\mathcal{S}''_1(l) = \mathcal{S}'(l)$.

We have proved that $(\sigma', \mathcal{S}') \simeq (\sigma', \mathcal{S}''_1)$. By the definition of σ'_1, $\mathit{Observe}(\tau)(\sigma'_1) = \emptyset$ and $\mathit{Observe}(\sigma')(\sigma'_1) = \emptyset$. Since $\mathcal{E}' \vdash$ `(lambda (x) e)` $: \tau, \emptyset$ and $(s, \sigma' \cup \sigma'_1, \mathcal{S}''_1, E(\mathtt{x}), \tau'') \in \mathcal{Q}$ for every $\mathtt{x} \in \mathit{Dom}(E)$ and every $\tau'' \preceq \mathcal{E}'(\mathtt{x})$, by the definition 4, $q \in \mathcal{F}(\mathcal{Q})$ □

The lemma 7 relates the definition 2 of the relation $\simeq$ with the consistency relation, defined in 3.

Lemma 7 (Equivalence) *If $(\sigma, \mathcal{S}) \simeq (\sigma', \mathcal{S}')$ then $s{:}\sigma, \mathcal{S} \models v{:}\tau$ if and only if $s{:}\sigma', \mathcal{S}' \models v{:}\tau$.*

Proof By hypothesis, $(\sigma, \mathcal{S}) \simeq (\sigma', \mathcal{S}')$. By definition 2, $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')$ and $(\sigma, \mathcal{S}) \sqsupseteq (\sigma', \mathcal{S}')$. If $s{:}\sigma, \mathcal{S} \models v{:}\tau$ then, since $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')$ and by the lemma 6, $s{:}\sigma', \mathcal{S}' \models v{:}\tau$. If $s{:}\sigma', \mathcal{S}' \models v{:}\tau$ then, since $(\sigma, \mathcal{S}) \sqsupseteq (\sigma', \mathcal{S}')$ and by the lemma 6, $s{:}\sigma, \mathcal{S} \models v{:}\tau$ □

The lemma 8 states that the typing judgment $s{:}\sigma, \mathcal{S} \models v{:}\tau$ is stable under substitution.

Lemma 8 (Substitution) *If $s{:}\sigma, \mathcal{S} \models v{:}\tau$ then $s{:}\theta\sigma, \theta\mathcal{S} \models v{:}\theta\tau$ for any substitution θ.*

Proof We consider the set $\mathcal{Q}{=}\{(s, \theta\sigma, \theta\mathcal{S}, v, \theta\tau) \mid s{:}\sigma, \mathcal{S} \models v{:}\tau\}$. We show, by case analysis on the structure of v, that $q = (s, \theta\sigma, \theta\mathcal{S}, v, \theta\tau)$ is in $\mathcal{F}(\mathcal{Q})$. Thus, $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$, showing that $\mathcal{Q}$ is $\mathcal{F}$-consistent.

- If $v = u$ then, by the definition 3, $\tau = \mathit{unit}$ so that $q \in \mathcal{F}(\mathcal{Q})$.

- If $v = l$ then, by the definition 3, $\tau = \mathit{ref}_\rho(\tau')$, $(\rho, \tau') \in \mathit{Rng}(\sigma)$, $\mathcal{S}(l) = (\rho, \tau')$ and $s{:}\sigma, \mathcal{S} \models s(l){:}\tau'$. Since $s{:}\sigma, \mathcal{S} \models s(l){:}\tau'$, by definition of $\mathcal{Q}$, $(s, \theta\sigma, \theta\mathcal{S}, s(l), \theta\tau') \in \mathcal{Q}$, $\theta(\rho, \tau') = \theta\mathcal{S}(l) \in \mathit{Rng}(\theta\sigma)$. By the definition 4, $q \in \mathcal{F}(\mathcal{Q})$.

- If $v = (\mathtt{x}, \mathtt{e}, E)$ then, by the definition 3, there exist $\mathcal{E}$ such that $\mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \tau, \emptyset$, σ' such that $\mathit{Observe}(\tau)(\sigma') = \emptyset$ and $\mathit{Observe}(\sigma)(\sigma') = \emptyset$, $\mathcal{S}'$ such that $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}')$ and $s{:}\sigma \cup \sigma', \mathcal{S}' \models E(\mathtt{x}){:}\tau'$ for any $\mathtt{x} \in \mathit{Dom}(E)$ and $\tau' \preceq \mathcal{E}(\mathtt{x})$. Let us define the substitution θ' of $\mathit{Regs}(\sigma')$ by the regions $\mathit{Regs}(\theta'\sigma')$ not free in $\theta\tau$ or $\theta\sigma$ and let θ' be same as θ elsewhere.

  Let us write $\mathcal{E}' = \theta'\mathcal{E}$, $\mathcal{S}'' = \theta'\mathcal{S}'$ and $\sigma'' = \theta'\sigma'$. By the lemma 1, for all $\tau'' \preceq \mathcal{E}'(\mathtt{x})$, there exists $\tau' \preceq \mathcal{E}(\mathtt{x})$ such that the restriction θ'' of θ' on the free variables of $\mathcal{E}$ verifies $\tau'' = \theta''\tau'$. By definition of $\mathcal{Q}$, $(s, \theta\sigma \cup \sigma'', \mathcal{S}'', E(\mathtt{x}), \tau'') \in \mathcal{Q}$ for every $\mathtt{x} \in \mathit{Dom}(E)$ and every $\tau'' \preceq \mathcal{E}'(\mathtt{x})$. By definition of *Observe* and θ', $\mathit{Observe}(\theta\sigma)(\sigma'') = \emptyset$ and $\mathit{Observe}(\theta\tau)(\sigma'') = \emptyset$. By the definition 2, $(\theta\sigma, \theta\mathcal{S}) \simeq (\theta\sigma, \mathcal{S}'')$. By the lemma 4 with θ', $\mathcal{E}' \vdash \mathtt{(lambda\ (x)\ e)} : \theta\tau, \emptyset$. By the definition 4, $q \in \mathcal{F}(\mathcal{Q})$ □

The lemma 9 is a refinement of lemma 8 and states that the judgment $s{:}\sigma, \mathcal{S} \models v{:}\tau$ reduces to $s{:}\sigma, \mathcal{S} \models v{:}\theta\tau$ for any substitution θ that affects τ but not σ. It is used in the proof of consistency, theorem 6, to show that our type generalization criterion is correct.

Lemma 9 (Instantiation) *If $s{:}\sigma, \mathcal{S} \models v{:}\tau$ and θ is defined on $\mathit{fv}(\tau) \setminus \mathit{fv}(\sigma)$ then $s{:}\sigma, \mathcal{S} \models v{:}\theta\tau$.*

Proof By hypothesis, $s{:}\sigma, \mathcal{S} \models v{:}\tau$ and θ is defined on $\mathit{fv}(\tau) \setminus \mathit{fv}(\sigma)$. By the lemma 8, $s{:}\sigma, \theta\mathcal{S} \models v{:}\theta\tau$. Since $\theta\sigma = \sigma$, by the definition 2, $(\sigma, \mathcal{S}) \simeq (\sigma, \theta\mathcal{S})$. By the lemma 7, $s{:}\sigma, \mathcal{S} \models v{:}\theta\tau$ □

During the evaluation of an expression, the store is extended and updated in an organized way. The definition 5 specifies the requirements for preserving consistency between types and values in the presence of side-effects.

Definition 5 (Succession) *$(s, \sigma, \mathcal{S})$ becomes $(s', \sigma', \mathcal{S}')$, noted $(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma', \mathcal{S}')$, if and only if $\mathit{Dom}(s) \subseteq \mathit{Dom}(s')$, $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')$ and $s{:}\sigma, \mathcal{S} \models v{:}\tau$ implies $s'{:}\sigma', \mathcal{S}' \models v{:}\tau$ for any v and τ.*

The lemma 10 represents the situation that arises when a reference is initialized. In the lemma 11, we address the situation arising when a value is assigned to a reference.

Lemma 10 (Initialization) *Let* $\sigma' = \mathit{init}(\rho, \tau)$*,* $s' = s + \{l \mapsto v\}$ *(*$l \notin \mathit{Dom}(s)$*),* $\mathcal{S}' = \mathcal{S} + \{l \mapsto (\rho, \tau)\}$ *(*$l \notin \mathit{Dom}(\mathcal{S})$*) and* $s{:}\sigma, \mathcal{S} \models v{:}\tau$*. If* $s{:}\sigma, \mathcal{S} \models v'{:}\tau'$ *then* $s'{:}\sigma \cup \sigma', \mathcal{S}' \models v'{:}\tau'$*.*

Proof We consider the set $\mathcal{Q}{=}\{(s', \sigma \cup \sigma', \mathcal{S}', v', \tau') \mid s{:}\sigma, \mathcal{S} \models v'{:}\tau'\}$. We show, by case analysis on the structure of v, that $q = (s', \sigma \cup \sigma', \mathcal{S}', v', \tau')$ is in $\mathcal{F}(\mathcal{Q})$. Thus, $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$, showing that $\mathcal{Q}$ is $\mathcal{F}$-consistent.

- If $v' = u$ then, by the definition 3, $\tau' = \mathit{unit}$ so that $q \in \mathcal{F}(\mathcal{Q})$.

- If $v' = l'$ then, by the definition 3, $\tau' = \mathit{ref}_{\rho'}(\tau'')$, $(\rho', \tau'') \in \mathit{Rng}(\sigma)$, $\mathcal{S}(l') = (\rho', \tau'')$ and $s{:}\sigma, \mathcal{S} \models s(l'){:}\tau''$. Since $s{:}\sigma, \mathcal{S} \models s(l'){:}\tau''$, by definition of $\mathcal{Q}$, $(s', \sigma \cup \sigma', \mathcal{S}', s(l'), \tau'') \in \mathcal{Q}$.

  If $l' \neq l$, since $(\rho', \tau'') \in \mathit{Rng}(\sigma)$ and $\mathcal{S}(l') = (\rho', \tau'')$ then $(\rho', \tau'') \in \mathit{Rng}(\sigma \cup \sigma')$, $\mathcal{S}'(l') = (\rho', \tau'')$, $s'(l') = s(l')$ and $(s', \sigma \cup \sigma', \mathcal{S}', s(l'), \tau'') \in \mathcal{Q}$. If $l' = l$ then, by hypothesis, $v = s'(l')$, $\tau'' = \tau$ and $s{:}\sigma, \mathcal{S} \models v{:}\tau$. Thus, by definition of $\mathcal{Q}$, $(s', \sigma \cup \sigma', \mathcal{S}', s'(l'), \tau) \in \mathcal{Q}$. By the definition 4, $q \in \mathcal{F}(\mathcal{Q})$.

- If $v' = (\texttt{x}, \texttt{e}, E)$ then, by the definition 3, there exist $\mathcal{E}$ such that $\mathcal{E} \vdash \texttt{(lambda (x) e)} : \tau', \emptyset$, σ_1 such that $\mathit{Observe}(\tau')(\sigma_1) = \emptyset$ and $\mathit{Observe}(\sigma)(\sigma_1) = \emptyset$, $\mathcal{S}_1$ such that $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}_1)$ and $s{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models E(\texttt{x}){:}\tau_1$ for any $\texttt{x} \in \mathit{Dom}(E)$ and $\tau_1 \preceq \mathcal{E}(\texttt{x})$. Let θ be defined on $\mathit{Regs}(\sigma_1)$ and such that the regions $\mathit{Regs}(\theta\sigma_1)$ are not free in τ', σ and σ'. Let $\sigma_1' = \theta\sigma_1$, $\mathcal{S}_1' = \theta\mathcal{S}_1$ and $\mathcal{E}' = \theta\mathcal{E}$. Since, by the lemma 2, $\mathit{Regs}(\sigma_1) \cap \mathit{fv}(\tau') = \emptyset$, by the lemma 4 with θ, $\mathcal{E}' \vdash \texttt{(lambda (x) e)} : \tau', \emptyset$.

  By the lemma 1, for all $\tau_1' \preceq \mathcal{E}'(\texttt{x})$, there exists $\tau_1 \preceq \mathcal{E}(\texttt{x})$ such that the restriction θ_1 of θ on $\mathcal{E}$ verifies $\tau_1' = \theta_1\tau_1$. Thus, by the lemma 8, $s{:}\sigma \cup \sigma_1', \mathcal{S}_1' \models E(\texttt{x}){:}\tau_1'$ for any $\texttt{x} \in \mathit{Dom}(E)$ and $\tau_1' \preceq \mathcal{E}'(\texttt{x})$. By definition of $\mathcal{Q}$, $(s, \sigma \cup \sigma' \cup \sigma_1', \mathcal{S}_1', E(\texttt{x}), \tau_1') \in \mathcal{Q}$ for every $\texttt{x} \in \mathit{Dom}(E)$ and every $\tau_1' \preceq \mathcal{E}'(\texttt{x})$. We have defined σ_1' such that $\mathit{Observe}(\sigma \cup \sigma')(\sigma_1') = \emptyset$ and $\mathit{Observe}(\tau')(\sigma_1') = \emptyset$ and $\mathcal{S}_1'$ such that $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}_1')$. By the definition 4, $q \in \mathcal{F}(\mathcal{Q})$ □

Lemma 11 (Assignment) *Let* $\sigma' = \mathit{write}(\rho, \tau)$*,* $s' = s_l + \{l \mapsto v\}$*,* $s{:}\sigma, \mathcal{S} \models l{:}\mathit{ref}_\rho(\tau)$ *and* $s{:}\sigma, \mathcal{S} \models v{:}\tau$*. If* $s{:}\sigma, \mathcal{S} \models v'{:}\tau'$ *then* $s'{:}\sigma \cup \sigma', \mathcal{S} \models v'{:}\tau'$*.*

Proof We consider the set $\mathcal{Q}{=}\{(s', \sigma \cup \sigma', \mathcal{S}, v', \tau') \mid s{:}\sigma, \mathcal{S} \models v'{:}\tau'\}$. We show, by case analysis on the structure of v, that $q = (s', \sigma \cup \sigma', \mathcal{S}, v', \tau')$ is in $\mathcal{F}(\mathcal{Q})$. Thus, $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$, showing that $\mathcal{Q}$ is $\mathcal{F}$-consistent.

- If $v' = u$ then, by the definition 3, $\tau' = \mathit{unit}$ so that $q \in \mathcal{F}(\mathcal{Q})$.

- If $v' = l'$ then, by the definition 3, $\tau' = \mathit{ref}_{\rho'}(\tau'')$, $(\rho', \tau'') \in \mathit{Rng}(\sigma)$, $\mathcal{S}(l') = (\rho', \tau'')$ and $s{:}\sigma, \mathcal{S} \models s(l'){:}\tau''$. By definition of $\mathcal{Q}$, $(s, \sigma \cup \sigma', \mathcal{S}, s(l'), \tau'') \in \mathcal{Q}$.

  Since $(\rho', \tau'') \in \mathit{Rng}(\sigma)$, by definition of σ', $(\rho', \tau'') \in \mathit{Rng}(\sigma \cup \sigma')$. If $l' = l$ then $v = s'(l')$ and $s{:}\sigma, \mathcal{S} \models v{:}\tau$. Thus, by definition of $\mathcal{Q}$, $(s', \sigma \cup \sigma', \mathcal{S}, v, \tau) \in \mathcal{Q}$. Otherwise, $s'(l') = s(l')$ and $(s', \sigma \cup \sigma', \mathcal{S}, s'(l'), \tau'') \in \mathcal{Q}$. By the definition 4, $q \in \mathcal{F}(\mathcal{Q})$.

- If $v' = (\mathtt{x}, \mathtt{e}, E)$ then, by the definition 3, there exist $\mathcal{E}$ such that $\mathcal{E} \vdash$ `(lambda (x) e)` $: \tau, \emptyset, \sigma_1$ such that $\mathit{Observe}(\sigma)(\sigma_1) = \emptyset$ and $\mathit{Observe}(\tau)(\sigma_1) = \emptyset$, $\mathcal{S}_1$ such that $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}_1)$ and $s'{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models E(\mathtt{x}){:}\tau'$ for any $\mathtt{x} \in \mathit{Dom}(E)$ and $\tau' \preceq \mathcal{E}(\mathtt{x})$.

  Since $s{:}\sigma, \mathcal{S} \models l{:}\mathit{ref}_{\rho}(\tau)$, by definition 3, $(\rho, \tau) \in \mathit{Rng}(\sigma)$ and then $\mathit{Rng}(\sigma') \subseteq \mathit{Rng}(\sigma)$. Thus, $\mathit{Observe}(\sigma \cup \sigma')(\sigma_1) = \emptyset$. By definition of $\mathcal{Q}$, $(s, \sigma \cup \sigma' \cup \sigma_1, \mathcal{S}_1, E(\mathtt{x}), \tau') \in \mathcal{Q}$ for every $\mathtt{x} \in \mathit{Dom}(E)$ and every $\tau' \preceq \mathcal{E}(\mathtt{x})$. By the definition 4, $q \in \mathcal{F}(\mathcal{Q})$ □

The lemma 12 is used in the inner proof case of the theorem 6 in the case the rule (sub) is used. It tells that the consistency of the rest of the computation is not affected by unobservable effects.

Lemma 12 (Observability) *If $s{:}\sigma, \mathcal{S} \models v{:}\tau$ and $\mathit{Observe}(\tau)(\sigma) \subseteq \sigma' \subseteq \sigma$ then $s{:}\sigma', \mathcal{S} \models v{:}\tau$.*

Proof We consider the set $\mathcal{Q}$=$\{(s, \sigma', \mathcal{S}, v, \tau) \mid s{:}\sigma, \mathcal{S} \models v{:}\tau\}$ for any σ' such that $\sigma' \subseteq \sigma$ and $\mathit{Observe}(\tau)(\sigma) \subseteq \sigma'$. We show, by case analysis on the structure of v, that $q = (s, \sigma', \mathcal{S}, v, \tau)$ is in $\mathcal{F}(\mathcal{Q})$. Thus, $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$, showing that $\mathcal{Q}$ is $\mathcal{F}$-consistent.

- If $v = u$ then, by the definition 3, $\tau = \mathit{unit}$ so that $q \in \mathcal{F}(\mathcal{Q})$.

- If $v = l$ then, by the definition 3, $\tau = \mathit{ref}_{\rho}(\tau')$, $(\rho, \tau') \in \mathit{Rng}(\sigma)$, $\mathcal{S}(l) = (\rho, \tau')$ and $s{:}\sigma, \mathcal{S} \models s(l){:}\tau'$. Since $s{:}\sigma, \mathcal{S} \models s(l){:}\tau'$, by definition of $\mathcal{Q}$, $(s, \sigma', \mathcal{S}, s(l), \tau') \in \mathcal{Q}$. Since $(\rho, \tau') \in \mathit{Rng}(\sigma)$ and $\tau = \mathit{ref}_{\rho}(\tau')$, by definition of *Observe*, $(\rho, \tau') \in \mathit{Rng}(\mathit{Observe}(\tau)(\sigma))$. Since $\mathit{Observe}(\tau)(\sigma) \subseteq \sigma'$, $(\rho, \tau') \in \mathit{Rng}(\sigma')$. Thus, by the definition 4, $q \in \mathcal{F}(\mathcal{Q})$.

- If $v = (\mathtt{x}, \mathtt{e}, E)$ then, by the definition 3, there exist $\mathcal{E}$ such that $\mathcal{E} \vdash$ `(lambda (x) e)` $: \tau, \emptyset, \sigma_1$ such that $\mathit{Observe}(\tau)(\sigma_1) = \emptyset$ and $\mathit{Observe}(\sigma)(\sigma_1) = \emptyset$, $\mathcal{S}_1$ such that $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}_1)$ and $s{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models E{:}\mathcal{E}$.

  By the definition 3, for any $\mathtt{x} \in \mathit{Dom}(\mathcal{E})$ and $\tau' \preceq \mathcal{E}(\mathtt{x})$, $s{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models E(\mathtt{x}){:}\tau'$. By definition of $\mathcal{Q}$, $(s, \sigma \cup \sigma_1, \mathcal{S}_1, E(\mathtt{x}), \tau') \in \mathcal{Q}$

Let $\sigma_1' = \sigma_1 \cup (\sigma \setminus \sigma')$. Since $\mathit{Observe}(\tau)(\sigma_1) = \emptyset$ and $\mathit{fr}(\tau) \cap \mathit{Regs}(\sigma \setminus \sigma') = \emptyset$, we have $\mathit{Observe}(\tau)(\sigma_1') = \emptyset$. Similarly, since $\mathit{Observe}(\sigma)(\sigma_1) = \emptyset$ and $\mathit{fr}(\sigma') \cap \mathit{Regs}(\sigma \setminus \sigma') = \emptyset$, we have $\mathit{Observe}(\sigma')(\sigma_1') = \emptyset$.

We have that $\sigma \cup \sigma_1 = \sigma' \cup \sigma_1'$ and finally, since $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}_1)$ and $\sigma' \subseteq \sigma$, $(\sigma', \mathcal{S}) \simeq (\sigma', \mathcal{S}_1)$. By the definition 4, $q \in \mathcal{F}(\mathcal{Q})$ □

The consistency theorem appears below. The effect σ corresponds to the effect of evaluating the environment of the expression e. Let E and $\mathcal{E}$ be consistent with respect to this initial effect σ and such that the initial store s and a store model $\mathcal{S}$ satisfy $s{:}\sigma \cup \sigma', \mathcal{S} \models E{:}\mathcal{E}$. If e is such that $s, E \vdash \mathsf{e} \rightarrow v, s'$ and $\mathcal{E} \vdash \mathsf{e} : \tau, \sigma'$, then there exists a store model $\mathcal{S}'$ such that $(s, \sigma, \mathcal{S})$ becomes $(s', \sigma \cup \sigma', \mathcal{S}')$ and that the value v is consistent with its type, according to the model $\mathcal{S}'$ satisfying $s'{:}\sigma \cup \sigma', \mathcal{S}' \models v{:}\tau$.

Theorem 1 (Consistency of dynamic and static semantics) *If* $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$, $\mathcal{E} \vdash \mathsf{e} : \tau, \sigma'$ *and* $s, E \vdash \mathsf{e} \rightarrow v, s'$ *then there exists* $\mathcal{S}'$ *such that* $(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}')$ *and* $s'{:}\sigma \cup \sigma', \mathcal{S}' \models v{:}\tau$.

Proof The proof is by induction on the length of the dynamic evaluation. Before detailling the case analysis that corresponds to each syntactic form, we detail the inner proof case that corresponds to the application of the rule (sub) in the static semantics. The situation is that $\mathcal{E} \vdash \mathsf{e} : \tau, \sigma'$, $s, E \vdash \mathsf{e} \rightarrow v, s'$ and $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$. The judgment $\mathcal{E} \vdash \mathsf{e} : \tau, \sigma'$ was inferred from the rule of observation.

$$\frac{\mathcal{E} \vdash \mathsf{e} : \tau, \sigma_1 \qquad \mathit{Observe}(\mathcal{E}, \tau)(\sigma_1) \subseteq \sigma'}{\mathcal{E} \vdash \mathsf{e} : \tau, \sigma'}$$

Let us write $\sigma_1' = \mathit{Observe}(\mathcal{E}, \tau)(\sigma_1)$ and $\sigma_2' = \sigma_1 \setminus \sigma_1'$. Let us define the substitution θ on $\mathit{Regs}(\sigma_2')$ such that the regions $\mathit{Regs}(\theta\sigma_2')$ are not free in $\mathcal{E}$, $\sigma \cup \sigma_1'$ and τ. Let us write $\sigma_2'' = \theta\sigma_2'$. Since $\mathit{Observe}(\mathcal{E}, \tau)(\sigma_2') = \emptyset$, by the lemma 2, $\mathit{Regs}(\sigma_2') \cap (\mathit{fr}(\mathcal{E}) \cup \mathit{fr}(\tau)) = \emptyset$. By the lemma 4,

$$\mathcal{E} \vdash \mathsf{e} : \tau, \sigma_1' \cup \sigma_2''$$

By hypothesis on the inner proof, there exists $\mathcal{S}'$ such that $s'{:}\sigma \cup \sigma_1' \cup \sigma_2'', \mathcal{S}' \models v{:}\tau$ and that $(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma_1' \cup \sigma_2'', \mathcal{S}')$. Since $\mathit{Observe}(\mathcal{E}, \tau)(\sigma_2'') = \emptyset$ (and of course $\mathit{Observe}(\tau)(\sigma_2'') \subseteq \mathit{Observe}(\mathcal{E}, \tau)(\sigma_2'')$), by the lemma 12,

$$s'{:}\sigma \cup \sigma_1', \mathcal{S}' \models v{:}\tau$$

Let us consider any v' and τ' such that $s{:}\sigma, \mathcal{S} \models v'{:}\tau'$. Since $(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma_1' \cup \sigma_2'', \mathcal{S}')$, by the definition 5, $s'{:}\sigma \cup \sigma_1' \cup \sigma_2'', \mathcal{S}' \models v'{:}\tau'$. We can freely choose $\sigma_2'' = \theta\sigma_2'$, in the judgment $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma_1' \cup \sigma_2''$, so that $\mathit{Regs}(\sigma_2'') \cap \mathit{fr}(\tau') = \emptyset$. By the definition of *Observe*, $\mathit{Observe}(\tau')(\sigma_2'') = \emptyset$ and thus, by the lemma 12, $s'{:}\sigma \cup \sigma_1', \mathcal{S}' \models v'{:}\tau'$. This holds for any judgment $s{:}\sigma, \mathcal{S} \models v'{:}\tau'$. Thus, by the definition 2,

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma_1', \mathcal{S}')$$

Since $\sigma_1' \subseteq \sigma'$, by the lemma 6,

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}') \text{ and } s'{:}\sigma \cup \sigma', \mathcal{S}' \models v{:}\tau$$

Case of (var) By hypothesis $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$, $s, E \vdash \mathtt{x} \to v, s$ and $\mathcal{E} \vdash \mathtt{x} : \tau, \emptyset$. By definition of the rule (var) this requires that $E(\mathtt{x}) = v$ and that $\tau \preceq \mathcal{E}(\mathtt{x})$. By the definition 3, $s{:}\sigma, \mathcal{S} \models v{:}\tau$. We conclude, taking $s' = s$, $\sigma' = \emptyset$ and $\mathcal{S}' = \mathcal{S}$, that

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}') \text{ and } s'{:}\sigma \cup \sigma', \mathcal{S}' \models v{:}\tau$$

Case of (abs) By hypothesis $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$. The rules (abs) of the dynamic and static semantics impose that $s, E \vdash \mathtt{(lambda\ (x)\ e)} \to (\mathtt{x}, \mathtt{e}, E_{\mathtt{x}}), s$ and $\mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \tau, \emptyset$. By the definition 3, taking $s' = s$, $\sigma' = \emptyset$ and $\mathcal{S}' = \mathcal{S}$, we conclude that

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}') \text{ and } s'{:}\sigma \cup \sigma', \mathcal{S}' \models (\mathtt{x}, \mathtt{e}, E_{\mathtt{x}}){:}\tau$$

Case of (let) By hypothesis, $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$. The rule (let) of the dynamic semantics imposes that:

$$\frac{s, E \vdash \mathtt{e}_1 \to v_1, s_1 \qquad s_1, E_{\mathtt{x}} + \{\mathtt{x} \mapsto v_1\} \vdash \mathtt{e}_2 \to v_2, s'}{s, E \vdash \mathtt{(let\ (x\ e_1)\ e_2)} \to v_2, s'}$$

In the static semantics, writing $\sigma' = \sigma_1 \cup \sigma_2$, we have

$$\frac{\mathcal{E} \vdash \mathtt{e}_1 : \tau_1, \sigma_1 \qquad \mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \mathit{Gen}(\sigma_1, \mathcal{E})(\tau_1)\} \vdash \mathtt{e}_2 : \tau_2, \sigma_2}{\mathcal{E} \vdash \mathtt{(let\ (x\ e_1)\ e_2)} : \tau_2, \sigma_1 \cup \sigma_2}$$

Let us write $\vec{v'} = \mathit{fv}(\tau_1) \setminus (\mathit{fv}(\sigma_1) \cup \mathit{fv}(\mathcal{E}))$. Let us define the substitution θ' on $\vec{v'}$ such that the variables $\vec{v} = \theta'(\vec{v'})$ are distinct and not free in $\sigma \cup \sigma_1$ and $\mathcal{E}$. Let us write $\tau = \theta'\tau_1$. By the lemma 4,

$$\mathcal{E} \vdash \mathtt{e}_1 : \tau, \sigma_1$$

By induction hypothesis on $\mathtt{e}_1$, there exists $\mathcal{S}_1$ such that

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s_1, \sigma \cup \sigma_1, \mathcal{S}_1) \text{ and } s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models v_1{:}\tau$$

Since $(s, \sigma, \mathcal{S}) \sqsubseteq (s_1, \sigma \cup \sigma_1, \mathcal{S}_1)$, by the definition 5, $s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models E{:}\mathcal{E}$. Since $\vec{v} = \mathit{fv}(\tau) \setminus (\mathit{fv}(\sigma_1) \cup \mathit{fv}(\mathcal{E}))$, we have $\vec{v} \cap \mathit{fv}(\sigma_1) = \emptyset$. Let θ be any substitution defined on $\vec{v}$. Since $\vec{v} \cap \mathit{fv}(\sigma_1) = \emptyset$ and $\vec{v} \cap \mathit{fv}(\sigma) = \emptyset$, by the lemma 9, $s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models v_1{:}\theta\tau$. By definition of $\preceq$ and definition 3,

$$s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models v_1{:}\forall\vec{v}.\tau$$

Let us write $E' = E_{\mathtt{x}} + \{\mathtt{x} \mapsto v_1\}$ and $\mathcal{E}' = \mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \forall\vec{v}.\tau\}$. By induction hypothesis on $\mathtt{e}_2$, there exists $\mathcal{S}'$ such that $(s_1, \sigma \cup \sigma_1, \mathcal{S}_1) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}')$ and that $s'{:}\sigma \cup \sigma', \mathcal{S}' \models v_2{:}\tau_2$. Since $(s, \sigma, \mathcal{S}) \sqsubseteq (s_1, \sigma \cup \sigma_1, \mathcal{S}_1)$ and $(s_1, \sigma \cup \sigma_1, \mathcal{S}_1) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}')$, by the definition 5,

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}') \text{ and } s'{:}\sigma \cup \sigma', \mathcal{S}' \models v_2{:}\tau_2$$

Case of (app) By hypothesis, $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$. The rule (app) of the dynamic semantics imposes that:

$$\frac{s, E \vdash \mathtt{e}_1 \rightarrow (\mathtt{x}, \mathtt{e}_3, E'), s_1 \quad s_1, E \vdash \mathtt{e}_2 \rightarrow v_2, s_2 \quad s_2, E' + \{\mathtt{x} \mapsto v_2\} \vdash \mathtt{e}_3 \rightarrow v_3, s_3}{s, E \vdash \mathtt{(e}_1\ \mathtt{e}_2\mathtt{)} \rightarrow v_3, s_3}$$

Let us write $\sigma' = \sigma_1 \cup \sigma_2 \cup \sigma_3$. In the static semantics, we have:

$$\frac{\mathcal{E} \vdash \mathtt{e}_1 : \tau_2 \stackrel{\sigma_3}{\rightarrow} \tau_3, \sigma_1 \qquad \mathcal{E} \vdash \mathtt{e}_2 : \tau_2, \sigma_2}{\mathcal{E} \vdash \mathtt{(e}_1\ \mathtt{e}_2\mathtt{)} : \tau_3, \sigma_1 \cup \sigma_2 \cup \sigma_3}$$

By induction hypothesis on $\mathtt{e}_1$, there exists $\mathcal{S}_1$ such that

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s_1, \sigma \cup \sigma_1, \mathcal{S}_1) \text{ and } s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models (\mathtt{x}, \mathtt{e}_3, E'){:}\tau_2 \stackrel{\sigma_3}{\rightarrow} \tau_3$$

Since $(s, \sigma, \mathcal{S}) \sqsubseteq (s_1, \sigma \cup \sigma_1, \mathcal{S}_1)$, by the definition 5, $s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models E{:}\mathcal{E}$. By induction hypothesis on $\mathtt{e}_2$, there exists $\mathcal{S}_2$ such that

$$(s_1, \sigma \cup \sigma_1, \mathcal{S}_1) \sqsubseteq (s_2, \sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2) \text{ and } s_2{:}\sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2 \models v_2{:}\tau_2$$

Since $s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models (\mathtt{x}, \mathtt{e}_3, E'){:}\tau_2 \stackrel{\sigma_3}{\rightarrow} \tau_3$ and by the definition 5, $s_2{:}\sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2 \models (\mathtt{x}, \mathtt{e}_3, E'){:}\tau_2 \stackrel{\sigma_3}{\rightarrow} \tau_3$. By the definition 3, this requires that there exist $\mathcal{E}'$, $\mathcal{S}'_2$ and σ'_2 such that $(\sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2) \simeq (\sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}'_2)$, $\mathit{Observe}(\tau_2 \stackrel{\sigma_3}{\rightarrow} \tau_3)(\sigma'_2) = \emptyset$ and $\mathit{Observe}(\sigma \cup \sigma_1 \cup \sigma_2)(\sigma'_2) = \emptyset$, verifying

$$\mathcal{E}' \vdash \mathtt{(lambda\ (x)\ e}_3\mathtt{)} : \tau_2 \stackrel{\sigma_3}{\rightarrow} \tau_3, \emptyset \text{ and } s_2{:}\sigma \cup \sigma_1 \cup \sigma_2 \cup \sigma'_2, \mathcal{S}'_2 \models E'{:}\mathcal{E}'$$

Let us write $\vec{\rho} = \mathit{Regs}(\sigma_2')$ and define θ on $\vec{\rho}$ such that the regions $\theta(\vec{\rho})$ are not free in $\mathcal{E}$, $\sigma \cup \sigma_1 \cup \sigma_2$ and $\tau_2 \xrightarrow{\sigma_3} \tau_3$. Since $\mathit{Observe}(\tau_2 \xrightarrow{\sigma_3} \tau_3)(\sigma_2') = \emptyset$ and $\mathit{Observe}(\sigma \cup \sigma_1 \cup \sigma_2)(\sigma_2') = \emptyset$, by the lemma 2,

$$\mathit{fv}(\tau_2 \xrightarrow{\sigma_3} \tau_3) \cap \mathit{Regs}(\sigma_2') = \emptyset \text{ and } \mathit{fv}(\sigma \cup \sigma_1 \cup \sigma_2) \cap \mathit{Regs}(\sigma_2') = \emptyset$$

Let us write $\sigma_2'' = \theta\sigma_2'$ and $\mathcal{S}_2'' = \theta\mathcal{S}_2'$, we have $(\sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2) \simeq (\sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2'')$, $\mathit{Observe}(\tau_2 \xrightarrow{\sigma_3} \tau_3)(\sigma_2'') = \emptyset$ and $\mathit{Observe}(\sigma \cup \sigma_1 \cup \sigma_2)(\sigma_2'') = \emptyset$. By the lemmas 4 and 8, $\mathcal{E}'$, σ_2'' and $\mathcal{S}_2''$ verify

$$\theta\mathcal{E}' \vdash \texttt{(lambda (x) e}_3\texttt{)} : \tau_2 \xrightarrow{\sigma_3} \tau_3, \emptyset \text{ and } s_2{:}\sigma \cup \sigma_1 \cup \sigma_2 \cup \sigma_2'', \mathcal{S}_2'' \models E'{:}\theta\mathcal{E}'$$

Let us write $E'' = E'_{\texttt{x}} + \{\texttt{x} \mapsto v_2\}$ and $\mathcal{E}'' = \theta\mathcal{E}'_{\texttt{x}} + \{\texttt{x} \mapsto \tau_2\}$. By the definition 3, the lemmas 6 and 7 $s_2{:}\sigma \cup \sigma_1 \cup \sigma_2 \cup \sigma_2'', \mathcal{S}_2'' \models E''{:}\mathcal{E}''$. By induction on $\texttt{e}_3$, there exists $\mathcal{S}'$ such that

$$(s_2, \sigma \cup \sigma_1 \cup \sigma_2 \cup \sigma_2'', \mathcal{S}_2'') \sqsubseteq (s', \sigma \cup \sigma' \cup \sigma_2'', \mathcal{S}') \text{ and that } s', \sigma \cup \sigma' \cup \sigma_2'', \mathcal{S}' \models v_3{:}\tau_3$$

Since $s', \sigma \cup \sigma' \cup \sigma_2'', \mathcal{S}' \models v_3{:}\tau_3$ and $\mathit{Observe}(\tau_3)(\sigma_2') = \emptyset$, by the lemma 12,

$$s', \sigma \cup \sigma', \mathcal{S}' \models v_3{:}\tau_3$$

In the same manner, let us consider any v' and τ' such that $s{:}\sigma, \mathcal{S} \models v'{:}\tau'$. Since $(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma' \cup \sigma_2'', \mathcal{S}')$, by the definition 5, $s'{:}\sigma \cup \sigma' \cup \sigma_2'', \mathcal{S}' \models v'{:}\tau'$. Since we can freely choose $\sigma_2'' = \theta\sigma_2'$ so that $\mathit{Regs}(\sigma_2'') \cap \mathit{fr}(\tau') = \emptyset$, by the definition of $\mathit{Observe}$, $\mathit{Observe}(\tau')(\sigma_2'') = \emptyset$. Thus, by the lemma 12, $s'{:}\sigma \cup \sigma', \mathcal{S}' \models v'{:}\tau'$. This holds for any judgment $s{:}\sigma, \mathcal{S} \models v'{:}\tau'$. Thus, by the definition 2,

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}')$$

Case of (new) By hypothesis, $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$. The rule (new) of the dynamic semantics imposes that:

$$\frac{s, E \vdash \texttt{e} \rightarrow v, s_1 \qquad l \notin \mathit{Dom}(s_1)}{s, E \vdash \texttt{(new e)} \rightarrow l, s_1 + \{l \mapsto v\}}$$

In the static semantics, the situation is

$$\frac{\mathcal{E} \vdash \texttt{e} : \tau, \sigma_1}{\mathcal{E} \vdash \texttt{(new e)} : \mathit{ref}_\rho(\tau), \sigma_1 \cup \mathit{init}(\rho, \tau)}$$

Let us write $\sigma' = \mathit{init}(\rho, \tau) \cup \sigma_1$. By induction hypothesis on $\texttt{e}$, there exists $\mathcal{S}_1$ such that $(s, \sigma, \mathcal{S}) \sqsubseteq (s_1, \sigma \cup \sigma_1, \mathcal{S}_1)$ and that $s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models v{:}\tau$. Let us write $(s', \sigma \cup \sigma', \mathcal{S}') = (s_1 + \{l \mapsto v\}, \sigma \cup \sigma', \mathcal{S}_1 + \{l \mapsto (\rho, \tau)\})$. Since $l \notin \mathit{Dom}(s_1)$ and $s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models v{:}\tau$, by the lemma 10, $(s_1, \sigma \cup \sigma_1, \mathcal{S}_1) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}')$. Thus, by the definitions 5 and 6,

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}') \text{ and } s'{:}\sigma \cup \sigma', \mathcal{S}' \models l{:}\mathit{ref}_\rho(\tau)$$

Case of (get) By hypothesis, $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$. The rule (get) of the dynamic semantics imposes that:

$$\frac{s, E \vdash \texttt{e} \rightarrow l, s' \qquad l \in \mathit{Dom}(s')}{s, E \vdash \texttt{(get e)} \rightarrow s'(l), s'}$$

In the static semantics, the rule (get) reads

$$\frac{\mathcal{E} \vdash \texttt{e} : \mathit{ref}_\rho(\tau), \sigma_1}{\mathcal{E} \vdash \texttt{(get e)} : \tau, \sigma_1 \cup \mathit{read}(\rho, \tau)}$$

Let us write $\sigma' = \mathit{read}(\rho, \tau) \cup \sigma_1$. By induction hypothesis on $\texttt{e}$, there exists $\mathcal{S}'$ such that $(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma_1, \mathcal{S}')$ and that $s'{:}\sigma \cup \sigma_1, \mathcal{S}' \models l{:}\mathit{ref}_\rho(\tau)$. By the definitions 5 and 3,

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}') \text{ and } s'{:}\sigma \cup \sigma', \mathcal{S}' \models s'(l){:}\tau$$

Case of (set) By hypothesis, $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$. The rule (set) of the dynamic semantics imposes that:

$$\frac{s, E \vdash \texttt{e}_1 \rightarrow l, s_1 \qquad s_1, E \vdash \texttt{e}_2 \rightarrow v, s_2}{s, E \vdash \texttt{((set e}_1\texttt{) e}_2\texttt{)} \rightarrow u, s_{2_l} + \{l \mapsto v\}}$$

In the static semantics, we have that

$$\frac{\mathcal{E} \vdash \texttt{e}_1 : \mathit{ref}_\rho(\tau), \sigma_1 \qquad \mathcal{E} \vdash \texttt{e}_2 : \tau, \sigma_2}{\mathcal{E} \vdash \texttt{((set e}_1\texttt{) e}_2\texttt{)} : \mathit{unit}, \sigma_1 \cup \sigma_2 \cup \mathit{write}(\rho, \tau)}$$

Let us write $\sigma' = \sigma_1 \cup \sigma_2 \cup \mathit{write}(\rho, \tau)$. By induction hypothesis on $\texttt{e}_1$, there exists $\mathcal{S}_1$ such that

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s_1, \sigma \cup \sigma_1, \mathcal{S}_1) \text{ and that } s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models l{:}\mathit{ref}_\rho(\tau)$$

By the definition 5, $s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models E{:}\mathcal{E}$. By induction hypothesis on $\texttt{e}_2$, there exists $\mathcal{S}_2$ such that

$$(s_1, \sigma \cup \sigma_1, \mathcal{S}_1) \sqsubseteq (s_2, \sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2) \text{ and that } s_2{:}\sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2 \models v{:}\tau$$

By the definition 5, $s_2{:}\sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2 \models l{:}\mathit{ref}_\rho(\tau)$. Let us write $(s', \sigma \cup \sigma', \mathcal{S}') = (s_{2_l} + \{l \mapsto v\}, \sigma \cup \sigma', \mathcal{S}_2)$. By the lemma 11, $(s_2, \sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}')$. By the definitions 5 and 3, we conclude that

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}') \text{ and } s'{:}\sigma \cup \sigma', \mathcal{S}' \models u{:}\mathit{unit} \,\Box$$

7 The Reconstruction Algorithm

We present the inference algorithm $\mathcal{I}$ that reconstructs the principal type and effect of expressions with respect to the static semantics. The inference algorithm $\mathcal{I}$ uses a double recursion scheme that separates the reconstruction of types and effects from the process of restricting effects with regard to the observation criterion. The next section explains the notions introduced in the algorithm.

7.1 Constrained Type Schemes

We use *constrained type schemes* to generically represent the possible types and constraint sets of `let`-bound expressions. In the static semantics, type schemes were of the form $\forall\vec{v}.\tau$. But now, since effect variables occur in function types, constraint sets involving these effect variables have to be kept within type schemes. In the algorithm, the type environment $\mathcal{E}$ binds value identifiers to such constrained type schemes.

Constrained type schemes, written $\forall\vec{v}.(\tau,\kappa)$ or $\forall\vec{\alpha}.\forall\vec{\varrho}.\forall\vec{\varsigma}.(\tau,\kappa)$, are composed of a type τ and a set of inequalities κ universally quantified over type, effect and region variables. The type and constraint set associated with `e` only depend on the free variables of `e` and, thereby, on the type environment $\mathcal{E}$. We write $\forall\vec{v}.(\tau,\emptyset) = \forall\vec{v}.\tau$

In order to relate the constrained type schemes and environments of the algorithm to the static semantics, we define a relation from the former to the latter by using the principal model $\overline{\kappa}$ of a constraint set κ, as defined in the section 8. We write: $\overline{\forall\vec{v}.(\tau,\kappa)} = \forall\vec{v}.(\overline{\kappa}\tau)$ and $\overline{\mathcal{E}}(\mathtt{x}) = \overline{\mathcal{E}(\mathtt{x})}$ for all $\mathtt{x} \in \mathit{Dom}(\mathcal{E})$.

$$\mathit{Gen}_\kappa(\mathcal{E},\sigma)(\tau) = \text{let } \{\vec{v}\} = \mathit{fv}(\overline{\kappa}\tau) \setminus (\mathit{fv}(\overline{\kappa}\overline{\mathcal{E}}) \cup \mathit{fv}(\overline{\kappa}\sigma)) \text{ in } (\forall\vec{v}.(\tau,\kappa_{\vec{v}}), \kappa \setminus \kappa_{\vec{v}})$$

$$\mathit{Inst}(\forall\vec{v}.(\tau,\kappa)) = \text{let } \vec{v'} \text{ new and } \theta = \{\vec{v} \mapsto \vec{v'}\} \text{ in } (\theta\tau, \theta\kappa)$$

Generalization and Instantiation

For a given constraint set κ, the function Gen_κ generalizes the type τ of an expression upon the variables that are neither free in its environment $\mathcal{E}$ nor present in its observed effect σ. We write $\kappa_{\vec{v}}$ for the restriction $\kappa_{\vec{v}} = \{\varsigma \supseteq \sigma \in \kappa \mid \varsigma \in \vec{v}\}$ of κ on the effect variables $\vec{v}$. We write $\kappa \setminus \kappa_{\vec{v}}$ the complement of $\kappa_{\vec{v}}$ in κ. The instantiation of type schemes for value identifiers and operators is done by using the function *Inst*.

7.2 Constrained Type Schemes of Store Operations

In the reconstruction algorithm, the store operation `new`, `get` and `set` are viewed as operators. They are related with appropriate constrained type schemes by the function $\mathit{TypeOf}[\![.]\!]$

$$\begin{aligned}
\mathit{TypeOf}[\![\texttt{set}]\!] &= \forall \alpha\varrho\varsigma\varsigma'.(\mathit{ref}_{\varrho}(\alpha) \xrightarrow{\varsigma} \alpha \xrightarrow{\varsigma'} \mathit{unit}, \{\varsigma' \supseteq \mathit{write}(\varrho, \alpha)\}) \\
\mathit{TypeOf}[\![\texttt{get}]\!] &= \forall \alpha\varrho\varsigma.(\mathit{ref}_{\varrho}(\alpha) \xrightarrow{\varsigma} \alpha, \{\varsigma \supseteq \mathit{read}(\varrho, \alpha)\}) \\
\mathit{TypeOf}[\![\texttt{new}]\!] &= \forall \alpha\varrho\varsigma.(\alpha \xrightarrow{\varsigma} \mathit{ref}_{\varrho}(\alpha), \{\varsigma \supseteq \mathit{init}(\varrho, \alpha)\})
\end{aligned}$$

Constrained Type Schemes for Store Operations

7.3 The Reconstruction Algorithm

In the first phase of the reconstruction, the algorithm $\mathcal{I}'$, given an environment $\mathcal{E}$ and a constraint set κ, reconstructs the type τ and the effect σ of an expression `e`, together with a substitution θ that ranges over the free variables of the environment $\mathcal{E}$ and an updated constraint set κ'. In its second phase, the algorithm $\mathcal{I}$ takes into account the observation criterion *Observe* in order to restrict the effect σ computed by the algorithm $\mathcal{I}'$.

```
$\mathcal{I}(\mathcal{E}, \kappa, \texttt{e}) = $ let $(\theta, \tau, \sigma, \kappa') = \mathcal{I}'(\mathcal{E}, \kappa, \texttt{e})$ in $(\theta, \tau, \mathit{Observe}(\overline{\kappa}'(\theta\overline{\mathcal{E}}), \overline{\kappa}'\tau)(\overline{\kappa}'\sigma), \kappa')$

$\mathcal{I}'(\mathcal{E},\kappa,\texttt{e}) = $ case e of
          op ⇒ let $(\tau', \kappa') = \mathit{Inst}(\mathit{TypeOf}[\![\texttt{op}]\!])$ in $(\mathit{Id}, \tau', \emptyset, \kappa \cup \kappa')$
           x ⇒ if $\texttt{x} \in \mathit{Dom}(\mathcal{E})$
                   then let $(\tau', \kappa') = \mathit{Inst}(\mathcal{E}(\texttt{x}))$ in $(\mathit{Id}, \tau', \emptyset, \kappa \cup \kappa')$
                   else fail
(lambda (x) e) ⇒ let $\alpha$, $\varsigma$ new
                   $(\theta, \tau, \sigma, \kappa') = \mathcal{I}(\mathcal{E}_{\texttt{x}} + \{\texttt{x} \mapsto \alpha\}, \kappa, \texttt{e})$
               in $(\theta, \theta\alpha \xrightarrow{\varsigma} \tau, \emptyset, \kappa' \cup \{\varsigma \supseteq \sigma\})$
```

Reconstruction Algorithm $\mathcal{I}$

```
        $(\mathtt{e}_1\ \mathtt{e}_2) \Rightarrow$ let $(\theta_1, \tau_1, \sigma_1, \kappa_1) = \mathcal{I}(\mathcal{E}, \kappa, \mathtt{e}_1)$
                        $(\theta_2, \tau_2, \sigma_2, \kappa_2) = \mathcal{I}(\theta_1\mathcal{E}, \kappa_1, \mathtt{e}_2)$
                        $\alpha, \varsigma$ new
                        $\theta_3 = \mathcal{U}_{\kappa_2}(\theta_2\tau_1, \tau_2 \xrightarrow{\varsigma} \alpha)$
                        $\kappa' = \theta_3(\kappa_2)$ and $\theta = \theta_3 \circ \theta_2 \circ \theta_1$
                    in $(\theta, \theta_3\alpha, \theta_3(\theta_2\sigma_1 \cup \sigma_2 \cup \varsigma), \kappa')$
$(\mathtt{let}\ (\mathtt{x}\ \mathtt{e}_1)\ \mathtt{e}_2) \Rightarrow$ let $(\theta_1, \tau_1, \sigma_1, \kappa_1) = \mathcal{I}(\mathcal{E}, \kappa, \mathtt{e}_1)$
                        $(\forall\vec{v}.(\tau_1, \kappa_1'), \kappa_1'') = Gen_{\kappa_1}(\theta_1\mathcal{E}, \sigma_1)(\tau_1)$
                        $(\theta_2, \tau, \sigma_2, \kappa') = \mathcal{I}(\theta_1\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \forall\vec{v}.(\tau_1, \kappa_1')\}, \kappa_1'', \mathtt{e}_2)$
                    in $(\theta_2 \circ \theta_1, \tau, \theta_2\sigma_1 \cup \sigma_2, \kappa')$
```

Reconstruction Algorithm $\mathcal{I}$ (continued)

An important invariant of the algorithm $\mathcal{I}$ is that latent effects of functions are always represented by effect variables in the algorithm. This technique was introduced in [Talpin & Jouvelot, Sept. 1992] and makes the problem of solving equations tractable by a simple extension to a unification algorithm on free algebras [Robinson, 1965] over effect variables. The unification algorithm $\mathcal{U}_\kappa(\tau, \tau')$, presented in section 8.2, solves the equations $\tau = \tau'$ on the types built by the algorithm $\mathcal{I}$ (We note $\mathcal{U}_\kappa$ for $\mathcal{U}$ curried with its first argument, the constraint set κ, in order to simplify the notation).

8 Constraint Resolution

We view the inference of types and effects of an expression as a constraint satisfaction problem. The algorithm builds equations on types and inequations on effects. In the algorithm, indirections between types and effects are introduced by the notion of constraint sets. Among the solutions of a constraint set κ, the principal model, $\overline{\kappa}$, defined below, satisfies the lemma 14.

Definition 6 (Model of Constraints) *A substitution θ is a model of a constraint set κ, written $\theta \models \kappa$, if and only if $\theta\varsigma \supseteq \theta\sigma$ for every constraint $\varsigma \supseteq \sigma$ in κ. The principal model $\overline{\kappa}$ of κ is inductively defined by:*

$$\overline{\emptyset} = Id \quad \textit{and} \quad \overline{\kappa \cup \{\varsigma \supseteq \sigma'\}} = \{\varsigma \mapsto \sigma'\} \circ \overline{\kappa} \quad \textit{where} \quad \sigma' = \overline{\kappa}(\varsigma \cup \sigma)$$

However, recall that we introduced types τ in effects σ as well as effects σ in types $\tau \xrightarrow{\sigma} \tau'$. Consequently, some expressions may now have recursively defined types and effects and shall thus be rejected by the static semantics.

Example The static semantics might constrain some expressions to have an effect σ containing $\mathit{init}(\rho, \tau \xrightarrow{\sigma} \tau')$ itself [Leroy, 1991]. The simplest known example producing such an ill-formed constraint set is:

```
(lambda (f)
  (let (x (new (new (lambda (x) x))))
    (if true f (lambda (y) (set x (new f)) y)))
```

In this program, the type of the function `f` has to match the type of the lambda-expression `(lambda (y) (set x (new f)) y)` that initializes an observable reference to `f` (Note that the incriminated effect must be observable for this situation to appear). Giving type τ to `x`, the type τ_f of `f` is recursively defined by:

$$\tau_f = \tau \xrightarrow{\varsigma \cup \mathit{write}(\rho', \mathit{ref}_{\rho}(\tau_f)) \cup \mathit{init}(\rho, \tau_f)} \tau$$

For our algorithm to be effectively implemented constraint sets must be checked for well-formedness. It must be enforced that no indirect cycles are introduced through *init* effects ■

8.1 Well-Formed Constraint Sets

Our solution is to only use well-formed constraint sets within the algorithm $\mathcal{I}$. They correspond to sound assignments of effect variables in the static semantics.

Definition 7 (Well-Formed Constraint Sets) *A constraint set κ is well-formed, written $\mathit{wf}(\kappa)$, if and only if, for every $\varsigma \supseteq \sigma$ such that $\kappa = \kappa' \cup \{\varsigma \supseteq \sigma\}$, we have:*

$$\forall\ (\rho, \tau) \in \mathit{Rng}(\overline{\kappa}'(\sigma)),\ \ \varsigma \notin \mathit{fv}(\tau)$$

The notation $\mathit{wf}(\kappa)$ is extended to type schemes by: $\mathit{wf}(\forall \vec{v}.(\tau, \kappa))$ iff $\mathit{wf}(\kappa)$ and to type environments by: $\mathit{wf}(\mathcal{E})$ iff $\mathit{wf}(\mathcal{E}(\mathtt{x}))$ for every `x` *in $\mathit{Dom}(\mathcal{E})$.*

The definition of well-formed constraint sets comes here with the following lemmas that state that well-formed constraint sets are solvable by finite substitutions.

Lemma 13 (Well-Formed Constraint Sets) *$\mathit{wf}(\kappa)$ if and only if $\overline{\kappa} \models \kappa$*

Proof [If *wf*(κ) then $\overline{\kappa} \models \kappa$] We proceed by induction on the number of constraints in κ. If $\kappa = \emptyset$, then $\overline{\kappa} = \mathit{Id}$ solves κ. Consider $\kappa = \kappa' \cup \{\varsigma \supseteq \sigma\}$ where $\kappa' = \kappa \setminus \{\varsigma \supseteq \sigma\}$. By definition, we have $\overline{\kappa} = \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\} \circ \overline{\kappa}'$ and by induction hypothesis on κ', $\overline{\kappa}'$ solves κ'. For every constraint $\varsigma' \supseteq \sigma'$ in κ', $\overline{\kappa}(\varsigma') = \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\}(\overline{\kappa}'(\varsigma'))$ and $\overline{\kappa}\sigma' = \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\}(\overline{\kappa}'(\sigma'))$.

- If $\varsigma \in \overline{\kappa}'(\varsigma')$ then $\overline{\kappa}(\varsigma') = (\overline{\kappa}'(\varsigma') \setminus \varsigma) \cup \overline{\kappa}'(\varsigma \cup \sigma)$. Since $\varsigma \in \overline{\kappa}'(\varsigma)$ we have $\overline{\kappa}(\varsigma') = \overline{\kappa}'(\varsigma') \cup \overline{\kappa}'(\varsigma \cup \sigma)$. By induction we have $\overline{\kappa}'(\varsigma') \supseteq \overline{\kappa}'(\sigma')$. If $\varsigma \in \overline{\kappa}'(\sigma')$ then $\overline{\kappa}(\sigma') = (\overline{\kappa}'(\sigma') \setminus \varsigma) \cup \overline{\kappa}'(\varsigma \cup \sigma)$ so that $\overline{\kappa}(\varsigma') \supseteq \overline{\kappa}(\sigma')$. Otherwise $\varsigma \notin \overline{\kappa}'(\sigma')$ so $\overline{\kappa}(\sigma') = \overline{\kappa}'(\sigma')$; thus $\overline{\kappa}(\varsigma') = \overline{\kappa}'(\varsigma') \cup \overline{\kappa}'(\varsigma \cup \sigma) \supseteq \overline{\kappa}'(\varsigma') \supseteq \overline{\kappa}'(\sigma') = \overline{\kappa}(\sigma')$
- Otherwise $\varsigma \notin \overline{\kappa}'(\varsigma')$ and $\overline{\kappa}(\varsigma') = \overline{\kappa}'(\varsigma')$. Since $\varsigma \notin \overline{\kappa}'(\varsigma')$ and $\overline{\kappa}'(\varsigma') \supseteq \overline{\kappa}'(\sigma')$ we have $\varsigma \notin \overline{\kappa}'(\sigma')$, so $\overline{\kappa}(\sigma') = \overline{\kappa}'(\sigma')$. Since $\overline{\kappa}'$ solves κ', we have that $\overline{\kappa}'(\varsigma') \supseteq \overline{\kappa}'(\sigma')$ so that $\overline{\kappa}(\varsigma') \supseteq \overline{\kappa}(\sigma')$.

It follows that $\overline{\kappa}\varsigma' \supseteq \overline{\kappa}\sigma'$ in both cases. For every constraint $\varsigma' \supseteq \sigma'$ in κ'; so $\overline{\kappa}$ solves κ'. It remains to show that $\overline{\kappa}$ solves $\{\varsigma \supseteq \sigma\}$. By definition $\overline{\kappa}(\varsigma) = \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\}(\overline{\kappa}'(\varsigma))$. Since $\varsigma \in \overline{\kappa}'(\varsigma)$, $\overline{\kappa}(\varsigma) = (\overline{\kappa}'(\varsigma) \setminus \varsigma) \cup \overline{\kappa}'(\varsigma \cup \sigma) = \overline{\kappa}'(\varsigma) \cup \overline{\kappa}'(\sigma)$. Also, $\overline{\kappa}(\sigma) = \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\}(\overline{\kappa}'(\sigma))$. If $\varsigma \in \overline{\kappa}'(\sigma)$, then $\overline{\kappa}(\sigma) = \overline{\kappa}'(\varsigma) \cup \overline{\kappa}'(\sigma)$, otherwise $\overline{\kappa}(\sigma) = \overline{\kappa}'(\sigma)$. In both cases, we have that $\overline{\kappa}$ solves $\{\varsigma \supseteq \sigma\}$. We have thus proved that $\overline{\kappa}$ solves κ.

Proof [If $\overline{\kappa} \models \kappa$ then *wf*(κ)] Let us assume that $\overline{\kappa} \models \kappa$ and rewrite κ as:

$$\kappa' = \{\varsigma \supseteq \cup_{(\varsigma \supseteq \sigma) \in \kappa} \sigma \setminus \varsigma \mid (\varsigma \supseteq \sigma') \in \kappa\}$$

If $\overline{\kappa} \models \kappa$ then, by definition of $\models$, $\overline{\kappa}(\varsigma) \supseteq \overline{\kappa}(\sigma)$ for every constraint $(\varsigma \supseteq \sigma) \in \kappa$. By construction of κ', $\overline{\kappa}'(\varsigma) \supseteq \overline{\kappa}'(\sigma)$. Thus, for every constraint $(\varsigma \supseteq \cup_{(\varsigma \supseteq \sigma) \in \kappa} \sigma \setminus \varsigma)$, $\overline{\kappa}'(\varsigma) \supseteq \overline{\kappa}'(\cup_{(\varsigma \supseteq \sigma) \in \kappa} \sigma \setminus \varsigma)$. Finally, $\overline{\kappa}' \models \kappa'$.

Consider any constraint $\varsigma \supseteq \sigma$ in κ' and define $\kappa'' = \kappa \setminus \{\varsigma \supseteq \sigma\}$. Since $\overline{\kappa}' \models \kappa'$ then, by definition of $\models$, $\overline{\kappa}'(\varsigma) \supseteq \overline{\kappa}'(\sigma)$. Since, by construction of $\overline{\kappa}'$, $\varsigma \in \overline{\kappa}'(\varsigma)$, this is equivalent to $\overline{\kappa}''(\varsigma) \cup \overline{\kappa}''(\sigma) \supseteq \overline{\kappa}'(\sigma)$. Since, by construction of κ', $\overline{\kappa}''(\varsigma) = \varsigma$, this is equivalent to $\varsigma \cup \overline{\kappa}''(\sigma) \supseteq \overline{\kappa}'(\sigma)$. Since, by construction of κ', $\varsigma \notin \sigma$, this is equivalent to $\overline{\kappa}''(\sigma) \supseteq \overline{\kappa}'(\sigma)$. Since, by definition of $\overline{\kappa}'$, $\overline{\kappa}'(\sigma) = \{\varsigma \mapsto \overline{\kappa}''(\varsigma \cup \sigma)\} \circ \overline{\kappa}''(\sigma)$, this implies that, for every pair $(\rho, \tau) \in \overline{\kappa}''(\sigma)$, $\tau = \{\varsigma \mapsto \overline{\kappa}''(\varsigma \cup \sigma)\}(\tau)$. This implies that $\varsigma \notin \mathit{fv}(\tau)$ and, by definition of *wf*, that *wf*(κ').

Now, if *wf*(κ') then, writing every constraint in κ' as $\varsigma \supseteq \cup_{(\varsigma \supseteq \sigma) \in \kappa} \sigma \setminus \varsigma$, we have that $\varsigma \notin \mathit{fv}(\tau)$ for any pair (ρ, τ) of $\overline{\kappa}''(\cup_{(\varsigma \supseteq \sigma) \in \kappa} \sigma \setminus \varsigma)$. This implies that it is not in any of the $\overline{\kappa}''(\sigma)$. Thus, we conclude that *wf*(κ) □

Lemma 14 (Principal Model) *If θ solves κ, then $\theta = \theta \circ \overline{\kappa}$.*

Proof By induction on the number of constraints in κ. If $\kappa = \emptyset$, then $\overline{\kappa} = \mathit{Id}$, so $\theta = \theta \circ \overline{\kappa}$. Now let us consider $\kappa = \kappa' \cup \{\varsigma \supseteq \sigma\}$ where $\kappa' = \kappa \setminus \{\varsigma \supseteq \sigma\}$ and let θ be a solution of κ. Note that θ solves κ', so by induction, $\theta = \theta \circ \overline{\kappa'}$. Let ς' be any effect variable; we wish to show $\theta(\varsigma') = \theta(\overline{\kappa}(\varsigma'))$. By definition, $\overline{\kappa}(\varsigma') = \{\varsigma \mapsto \overline{\kappa'}(\varsigma \cup \sigma)\}(\overline{\kappa'}(\varsigma'))$. Then, there are two cases:

- If $\varsigma \notin \overline{\kappa'}(\varsigma')$ then $\theta(\overline{\kappa}(\varsigma')) = \theta(\overline{\kappa'}(\varsigma')) = \theta(\varsigma')$, by induction.

- Otherwise, $\varsigma \in \overline{\kappa'}(\varsigma')$, so that
$$\begin{array}{rcll} \theta(\overline{\kappa}(\varsigma')) & = & \theta(\{\varsigma \mapsto \overline{\kappa'}(\varsigma \cup \sigma)\}(\overline{\kappa'}(\varsigma'))) & \\ & = & \theta(\overline{\kappa'}(\varsigma') \setminus \varsigma) \cup \theta(\overline{\kappa'}(\varsigma \cup \sigma)) & \text{as } \varsigma \in \overline{\kappa'}(\varsigma') \\ & = & \theta(\overline{\kappa'}(\varsigma')) \cup \theta(\overline{\kappa'}(\varsigma \cup \sigma)) & \text{as } \varsigma \in \overline{\kappa'}(\varsigma) \\ & = & \theta(\varsigma') \cup \theta(\varsigma \cup \sigma) & \text{by induction} \\ & = & \theta(\varsigma') \cup \theta(\varsigma) & \text{as } \theta \text{ solves } \kappa \end{array}$$

Now $\varsigma \in \overline{\kappa'}(\varsigma')$ implies that $\theta(\varsigma) \subseteq \theta(\overline{\kappa'}(\varsigma')) = \theta(\varsigma')$. Thus $\theta(\varsigma') \cup \theta(\varsigma) = \theta(\varsigma')$. Thus $\theta(\overline{\kappa}(\varsigma')) = \theta(\varsigma')$ in this case as well □

We state that ill-formed constraint sets cannot be satisfied by substitutions of effect variables by finite effect terms.

Lemma 15 (Ill-Formed Constraint Sets) *If κ is ill-formed, then there does not exist a substitution satisfying κ.*

Proof We show that, in order to satisfy an ill-formed constraint set κ, any substitution θ must substitute at least one effect variable by a non finite effect term.

We assume that $\neg \mathit{wf}(\kappa)$. By definition, this implies that there exists a constraint $\{\varsigma \supseteq \sigma\}$ in κ such that $(\rho, \tau) \in \mathit{Rng}(\sigma)$ and $\varsigma \in \mathit{fv}(\overline{\kappa}'\tau)$ where $\kappa' = \kappa \setminus \{\varsigma \supseteq \sigma\}$. Suppose that there exists a substitution θ such that $\theta \models \kappa$.

By the lemma 14, we know that $\theta = \theta \circ \overline{\kappa}$. By definition of $\overline{\kappa}$, we know that $\overline{\kappa} = \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\} \circ \overline{\kappa}'$. Then, from $\theta \models \kappa$ and by definition, θ must verify:

$$\theta(\overline{\kappa}(\varsigma)) = \theta(\overline{\kappa}'(\varsigma \cup \sigma)) \supseteq \theta(\overline{\kappa}(\sigma))$$

Thus, $\mathit{Rng}(\theta(\overline{\kappa}(\sigma)))$ must be in $\mathit{Rng}(\theta(\overline{\kappa}'(\varsigma \cup \sigma)))$. Since $(\rho, \tau) \in \mathit{Rng}(\sigma)$, the substitution θ must verify $\theta(\overline{\kappa}'(\tau)) = \theta(\overline{\kappa}(\tau))$. However,

$$\begin{array}{rll} \theta(\overline{\kappa}\tau) & = \theta(\{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\}(\overline{\kappa}'\tau)) & \text{by definition of } \overline{\kappa} \\ & = \theta(\{\varsigma \mapsto \varsigma \cup \overline{\kappa}'(\sigma)\}(\{\varsigma \mapsto \overline{\kappa}'\varsigma\}(\overline{\kappa}'\tau))) & \text{since } \varsigma \in \overline{\kappa}'\varsigma \text{ and } \overline{\kappa}'\varsigma = \varsigma \cup \overline{\kappa}'\varsigma \\ & = \theta(\{\varsigma \mapsto \varsigma \cup \overline{\kappa}'\sigma\}(\overline{\kappa}'\tau)) & \text{since } \{\varsigma \mapsto \overline{\kappa}'\varsigma\} \circ \overline{\kappa}' = \overline{\kappa}' \end{array}$$

Since $(\rho, \overline{\kappa}'(\tau)) \in \mathit{Rng}(\overline{\kappa}'(\sigma))$, $\theta(\overline{\kappa}'(\tau)) = \theta(\{\varsigma \mapsto (\varsigma \cup \overline{\kappa}'(\sigma))\} \circ \overline{\kappa}'(\tau))$ and $\varsigma \in \mathit{fv}(\overline{\kappa}'(\tau))$, the term $\theta\varsigma$ must satisfy $\theta\varsigma = \theta(\varsigma \cup \overline{\kappa}'\sigma)$. But since $(\rho, \overline{\kappa}'(\tau)) \in \mathit{Rng}(\overline{\kappa}'(\sigma))$ and $\varsigma \in \mathit{fv}(\overline{\kappa}'(\tau))$, $\theta\varsigma$ occurs in $\theta(\overline{\kappa}'(\tau))$. Thus, the substitution θ satisfying κ cannot be defined, since the term $\theta\varsigma$ must be defined recursively □

8.2 Unification Algorithm

In the reconstruction algorithm $\mathcal{I}$, instead of checking the well-formedness of the constructed constraint set after each expression is typechecked, we implement an extended occurrence check test, reporting the construction of ill-formed constraint at the point of unifying effect variables together.

```
                     $\mathcal{U}_\kappa(\tau, \tau')$ = case $(\tau, \tau')$ of
                  $(\mathit{unit}, \mathit{unit})$ ⇒ $\mathit{Id}$
                        $(\alpha, \alpha')$ ⇒ $\{\alpha \mapsto \alpha'\}$
             $(\alpha, \tau) | (\tau, \alpha)$ ⇒ if $\alpha \in \mathit{fv}(\overline{\kappa}\tau)$ then fail else $\{\alpha \mapsto \tau\}$
  $(\mathit{ref}_\varrho(\tau), \mathit{ref}_{\varrho'}(\tau'))$ ⇒ let $\theta = \{\varrho \mapsto \varrho'\}$ in $\mathcal{U}_{\theta\kappa}(\theta\tau, \theta\tau') \circ \theta$
  $(\tau_i \xrightarrow{\varsigma} \tau_f, \tau'_i \xrightarrow{\varsigma'} \tau'_f)$ ⇒ let $\theta_i = \mathcal{U}_\kappa(\tau_i, \tau'_i)$
                                 $\theta_f = \mathcal{U}_{\theta_i\kappa}(\theta_i\tau_f, \theta_i\tau'_f)$
                                 $\theta = \{\theta_f(\theta_i\varsigma) \mapsto \theta_f(\theta_i\varsigma')\} \circ \theta_f \circ \theta_i$
                              in if $\mathit{wf}(\theta\kappa)$ then $\theta$ else fail
                        otherwise ⇒ fail
```

Unification Algorithm

The unification algorithm $\mathcal{U}_\kappa(\tau, \tau')$, presented above, solves the equations $\tau = \tau'$ on types built by the algorithm $\mathcal{I}$ and checks the constraint set κ generated by the algorithm for well-formedness. It either fails or returns a substitution θ standing for the most general unifier of the two given type terms τ and τ', also checking that the substitution θ preserves the well-formedness of the given constraint set κ. The following soundness and completeness lemma give the invariants of the unification algorithm $\mathcal{U}$ (We note $\mathcal{U}_\kappa$ for $\mathcal{U}$ curried with its first argument, the constraint set κ, in order to lighten the notation).

Lemma 16 (Soundness of $\mathcal{U}$) *If κ is well-formed and $\mathcal{U}_\kappa(\tau, \tau') = \theta$, then $\theta\kappa$ is well-formed and $\theta\tau = \theta\tau'$.*

Proof The algorithm $\mathcal{U}$ unifies the terms of a free algebra and its soundness proof only departs from [Robinson, 1965] in the case that enforces the well-formedness of the constraint set: $\tau' = \alpha$. By hypothesis, κ is well formed and $\mathcal{U}_\kappa(\tau, \alpha) = \theta$. By definition of $\mathcal{U}_\kappa$, this requires that $\alpha \notin \mathit{fv}(\overline{\kappa}\tau)$ and $\theta = \{\alpha \mapsto \tau\}$. Thus, $\theta\alpha = \theta\tau$ and it remains to prove that $\theta\kappa$ is well formed.

By hypothesis, we have that κ is well-formed. By definition, this requires that for every constraint $\varsigma \supseteq \sigma$ in κ, considering $\kappa' = \kappa \setminus \{\varsigma \supseteq \sigma\}$, we have:

$$\forall\ (\rho, \tau') \in \mathit{Rng}(\overline{\kappa'}\sigma),\ \ \varsigma \notin \mathit{fv}(\tau')$$

We want to show that $\theta\kappa$ is well-formed. By definition, this requires to show that for every $\theta\varsigma \supseteq \theta\sigma$ in $\theta\kappa$, considering $\theta\kappa' = \theta\kappa \setminus \{\theta\varsigma \supseteq \theta\sigma\}$, we have:

$$\forall\ (\theta\rho, \theta\tau') \in \mathit{Rng}(\overline{\theta\kappa'}(\theta\sigma)),\ \ \theta\varsigma \notin \mathit{fv}(\theta\tau')$$

By definition, $\theta = \{\alpha \mapsto \tau\}$. Thus, we have to show that $\varsigma \notin \mathit{fv}(\theta\tau')$. If $\alpha \notin \mathit{fv}(\tau')$, then $\theta\tau' = \tau'$ so that we have $\varsigma \notin \mathit{fv}(\theta\tau')$. Otherwise, $\alpha \in \mathit{fv}(\tau')$, and since $\varsigma \notin \mathit{fv}(\tau')$ and $\theta\alpha = \tau$, it remains to show that $\varsigma \notin \mathit{fv}(\tau)$.

Assume that there exists $(\rho, \tau') \in \mathit{Rng}(\overline{\kappa'}\sigma)$ such that $\alpha \in \mathit{fv}(\tau')$. By definition of $\overline{\kappa}$, this requires that $\alpha \in \mathit{fv}(\overline{\kappa}\varsigma)$. We have shown that supposing $\varsigma \in \mathit{fv}(\tau)$ implies $\alpha \in \mathit{fv}(\overline{\kappa}\tau)$, which contradicts the hypothesis: $\alpha \notin \mathit{fv}(\overline{\kappa}\tau)$.

We have proved that for every $\varsigma \supseteq \theta\sigma$ in $\theta\kappa$, considering $\theta\kappa' = \theta\kappa \setminus \{\varsigma \supseteq \theta\sigma\}$, we have that $\varsigma \notin \mathit{fv}(\theta\tau')$ for every $(\rho, \theta\tau')$ in $\mathit{Rng}(\overline{\theta\kappa'}(\theta\sigma))$. By definition, this proves that $\theta\kappa$ is well-formed □

Lemma 17 (Completeness of $\mathcal{U}$) *Let κ be well-formed. Whenever $\theta'\tau = \theta'\tau'$ for a substitution θ' satisfying κ, then $\mathcal{U}_\kappa(\tau, \tau') = \theta$, then $\theta\kappa$ is well-formed and there exists a substitution model θ'' satisfying $\theta\kappa$ such that $\theta' = \theta'' \circ \theta$.*

Proof By hypothesis, κ is well-formed and there exists a substitution θ' satisfying κ such that $\theta'\tau = \theta'\tau'$. The cases that differ from the completeness proof of [Robinson, 1965] are those which require the well-formedness of the constraint set to be checked: $\tau = \alpha$ or $(\tau, \tau') = (\tau_i \xrightarrow{\varsigma} \tau_f, \tau_i' \xrightarrow{\varsigma'} \tau_f')$.

In the case of $\tau = \alpha$, the hypothesis is that $\theta'\tau = \theta'\alpha$. By the lemma 14, we have that $\theta' = \theta' \circ \overline{\kappa}$, so that $\theta'(\overline{\kappa}\tau) = \theta'\alpha$. This implies that α is not in $\mathit{fv}(\overline{\kappa}\tau)$, so that we get that $\theta = \{\alpha \mapsto \tau\} = \mathcal{U}_\kappa(\alpha, \tau)$ by definition. Consider θ'' defined by $\theta''\alpha = \alpha$ and $\theta''v = \theta'v$

otherwise. We have that $\theta' = \theta'' \circ \theta$ and that θ'' satisfies $\theta\kappa$ since θ' satisfies κ. By the lemma 15, this implies that $\theta\kappa$ is well-formed.

In the case of $(\tau, \tau') = (\tau_i \xrightarrow{\varsigma} \tau_f, \tau_i' \xrightarrow{\varsigma'} \tau_f')$, the hypothesis requires that $\theta'\tau_i = \theta'\tau_i'$, $\theta'\tau_f = \theta'\tau_f'$ and $\theta'\varsigma = \theta'\varsigma'$.

By induction hypothesis on τ_i and τ_i', $\mathcal{U}_\kappa(\tau_i, \tau_i') = \theta_i$, then $\mathit{wf}(\theta_i\kappa)$ and $\theta' = \theta_i'' \circ \theta_i$ for some $\theta_i'' \models \theta_i\kappa$. Since $\mathit{wf}(\theta_i\kappa)$ and since there exists a substitution θ_i'' satisfying $\theta_i\kappa$ such that $\theta_i''(\theta_i\tau) = \theta_i''(\theta_i\tau')$, then, by induction hypothesis on τ_f and τ_f', we have $\mathcal{U}_{\theta_i\kappa}(\theta_i\tau_f, \theta_i\tau_f') = \theta_f$, then $\mathit{wf}(\theta_f(\theta_i\kappa))$ and $\theta_i'' = \theta_f'' \circ \theta_f$ for some $\theta_f'' \models \theta_f(\theta_i\kappa)$.

Thus, there exists a substitution θ_f'' satisfying $\theta_f(\theta_i\kappa)$ such that $\theta_f''(\theta_f(\theta_i\tau)) = \theta_f''(\theta_f(\theta_i\tau'))$. But $\theta_f''(\theta_f(\theta_i\tau)) = \theta_f''(\theta_f(\theta_i\tau'))$ requires that $\theta_f''(\theta_f(\theta_i\varsigma)) = \theta_f''(\theta_f(\theta_i\varsigma'))$. Let us write $\theta'' = \theta_f''$ and $\theta = \{\theta_f(\theta_i\varsigma) \mapsto \theta_f(\theta_i\varsigma')\} \circ \theta_f \circ \theta_i$.

We have $\theta_f'' \circ \theta_f \circ \theta_i = \theta'' \circ \theta$. Since the substitution θ_f'' satisfies $\theta_f(\theta_i\kappa)$, then $\theta'' \circ \theta$ satisfies κ so that θ'' satisfies $\theta\kappa$. By the lemma 15, this implies that $\theta\kappa$ is well formed. As a conclusion, we get $\mathcal{U}_\kappa(\tau, \tau') = \theta$, $\mathit{wf}(\theta\kappa)$ and the substitution θ'' satisfies $\theta\kappa$ such that $\theta' = \theta'' \circ \theta$ □

9 Correctness of the Reconstruction Algorithm

In this section, we prove the correctness of the algorithm with respect to the static semantics. The soundness theorem states that the type and effect computed by $\mathcal{I}$ are provable in the static semantics, assuming any solution of the inferred constraints.

Theorem 2 (Soundness) *Let $\mathcal{E}$ and κ be well-formed. If $\mathcal{I}(\mathcal{E}, \kappa, \mathsf{e}) = (\theta, \tau, \sigma, \kappa')$ then $\overline{\kappa'}(\theta\overline{\mathcal{E}}) \vdash \mathsf{e} : \overline{\kappa'}\tau, \overline{\kappa'}\sigma$*

Proof The proof is by induction on the structure of expressions. We assume that the constrained type schemes in the environment $\mathcal{E}$ are constructed with the function *Gen*, so that, for every $\mathtt{x}$ in *Dom*$(\mathcal{E})$, we have that $\mathcal{E}(\mathtt{x}) = \forall\vec{v}.(\tau, \kappa)$ with κ restricted to the $\vec{v}$. Then, a consequence is that for any substitution θ, we have that $\theta\overline{\mathcal{E}}(\mathtt{x}) = \overline{\theta\mathcal{E}}(\mathtt{x})$, ignoring capture of bound variables. Also note that, by definition of the reconstruction algorithm, the constraint set κ' extends $\theta\kappa$; every model of κ' is thus a model of $\theta\kappa$.

Case of (var) By hypothesis $\mathcal{I}(\mathcal{E}, \kappa, \mathtt{x}) = (\mathit{Id}, \theta\tau, \emptyset, \kappa \cup \theta\kappa')$. By the definition of the algorithm, this requires that $\mathcal{E}(\mathtt{x}) = \forall\vec{v}.(\tau, \kappa')$, that $\theta = \{\vec{v} \mapsto \vec{v'}\}$ and that the $\vec{v'}$ are fresh. Since θ renames the $\vec{v}$ with fresh $\vec{v'}$, we have that $\theta(\overline{\kappa'}\tau) = \overline{\theta\kappa'}(\theta\tau)$. Since $\overline{\kappa'}$ and θ are only defined on $\vec{v}$, we have $\overline{\theta\kappa'}(\overline{\mathcal{E}}) = \overline{\mathcal{E}}$. By definition of $\preceq$, $\theta(\overline{\kappa}'\tau) \preceq \overline{\mathcal{E}}(\mathtt{x})$. By definition of the rule (var):

$$\overline{\theta\kappa'}\ \overline{\mathcal{E}} \vdash \mathtt{x} : \overline{\theta\kappa'}(\theta\tau), \emptyset$$

Because θ substitutes $\vec{v}$ with fresh $\vec{v'}$, we have that $\overline{\kappa \cup \theta\kappa'} = \overline{\kappa} \circ \overline{\theta\kappa'}$. By the lemma 4 used with $\overline{\kappa}$, we can conclude that:

$$\overline{\kappa}(\overline{\theta\kappa'}\ \overline{\mathcal{E}}) \vdash \mathtt{x} : \overline{\kappa}(\overline{\theta\kappa'}(\theta\tau)), \emptyset$$

Case of (let) By hypothesis $\mathcal{I}(\mathcal{E}, \kappa, \mathtt{(let\ (x\ e_1)\ e_2)}) = (\theta, \tau, \sigma, \kappa')$. By the definition of the algorithm $\mathcal{I}$, this requires that:

$$\mathcal{I}'(\mathcal{E}, \kappa, \mathtt{(let\ (x\ e_1)\ e_2)}) = (\theta, \tau, \sigma', \kappa') \quad \text{and} \quad \sigma = \mathit{Observe}(\overline{\kappa}'(\theta\overline{\mathcal{E}}), \overline{\kappa}'\tau)(\overline{\kappa}'\sigma')$$

By the definition of $\mathcal{I}'$, this requires that there exist θ_1, θ_2 such that $\theta = \theta_2 \circ \theta_1$ and σ_1, σ_2 such that $\sigma' = \theta_2\sigma_1 \cup \sigma_2$ satisfying:

$$(\theta_1, \tau_1, \sigma_1, \kappa_1) = \mathcal{I}(\mathcal{E}, \kappa, \mathtt{e}_1) \quad \text{and} \quad (\theta_2, \tau, \sigma_2, \kappa') = \mathcal{I}(\theta_1\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \forall\vec{v}.(\tau_1, \kappa_1')\}, \kappa_1'', \mathtt{e}_2)$$

where $\mathit{Gen}_{\kappa_1}(\sigma_1, \theta_1\mathcal{E})(\tau_1) = (\forall\vec{v}.(\tau_1, \kappa_1'), \kappa_1'')$. By induction hypothesis on $\mathtt{e}_1$, we get:

$$\overline{\kappa}_1(\theta_1\overline{\mathcal{E}}) \vdash \mathtt{e}_1 : \overline{\kappa}_1\tau_1, \overline{\kappa}_1\sigma_1$$

By the definition of *Gen*, we have that $\overline{\kappa}_1\tau_1 = \overline{\kappa}_1''(\overline{\kappa}_1'\tau_1)$, because κ_1' is the restriction of κ_1 on $\vec{v}$ and κ_1'' its complement in κ_1. We also have that $\overline{\kappa}_1(\theta_1\overline{\mathcal{E}}) = \overline{\kappa}_1''(\theta_1\overline{\mathcal{E}})$ and that $\overline{\kappa}_1\sigma_1 = \overline{\kappa}_1''\sigma_1$ since the $\vec{v}$ are neither free in $\overline{\kappa}_1(\overline{\theta_1\mathcal{E}})$ nor in $\overline{\kappa}_1\sigma_1$. Thus, we have that:

$$\overline{\kappa}_1''(\theta_1\overline{\mathcal{E}}) \vdash \mathtt{e}_1 : \overline{\kappa}_1''(\overline{\kappa}_1'\tau_1), \overline{\kappa}_1''\sigma_1$$

Since κ' extends $\theta_2\kappa_1''$, $\overline{\kappa}'$ satisfies $\theta_2\kappa_1''$. Thus $\overline{\kappa}' \circ \theta_2$ satisfies κ_1''. By the lemma 14, we have that $\overline{\kappa}' \circ \theta_2 = \overline{\kappa}' \circ \theta_2 \circ \overline{\kappa}_1''$. By the lemma 4 used with $\overline{\kappa}' \circ \theta_2$,

$$\overline{\kappa}'(\theta_2(\theta_1\overline{\mathcal{E}})) \vdash \mathtt{e}_1 : \overline{\kappa}'(\theta_2(\overline{\kappa}_1'\tau_1)), \overline{\kappa}'(\theta_2\sigma_1)$$

By induction hypothesis on $\mathtt{e}_2$, with κ_1'' and $\theta_1\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \forall\vec{v}.(\tau_1, \kappa_1')\}$, we get:

$$\overline{\kappa}'(\theta_2(\overline{\theta_1\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \overline{\forall\vec{v}.(\tau_1, \kappa_1')}\})) \vdash \mathtt{e}_2 : \overline{\kappa}'\tau, \overline{\kappa}'\sigma_2$$

Since $\overline{\theta_1\mathcal{E}_{\mathtt{x}}} = \theta_1\overline{\mathcal{E}_{\mathtt{x}}}$, then $\overline{\theta_1\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \overline{\forall\vec{v}.(\tau_1, \kappa_1')}\} = \theta_1\overline{\mathcal{E}_{\mathtt{x}}} + \{\mathtt{x} \mapsto \forall\vec{v}.(\overline{\kappa}_1'\tau_1)\}$. Thus,

$$\overline{\kappa}'(\theta_2(\theta_1\overline{\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \forall\vec{v}.\overline{\kappa}_1'\tau_1\})) \vdash \mathtt{e}_2 : \overline{\kappa}'\tau, \overline{\kappa}'\sigma_2$$

Since $\theta = \theta_2 \circ \theta_1$ and by the definition of the rule (let), this implies that:

$$\overline{\kappa}'(\theta\overline{\mathcal{E}}) \vdash \texttt{(let (x e}_1\texttt{) e}_2\texttt{)} : \overline{\kappa}'\tau, \overline{\kappa}'(\theta_2\sigma_1 \cup \sigma_2)$$

We know that $\mathit{Observe}(\overline{\kappa}'(\theta\overline{\mathcal{E}}), \overline{\kappa}'\tau)(\overline{\kappa}'\sigma') = \sigma$, so that, using the rule (sub), we get:

$$\overline{\kappa}'(\theta\overline{\mathcal{E}}) \vdash \texttt{(let (x e}_1\texttt{) e}_2\texttt{)} : \overline{\kappa}'\tau, \sigma$$

By the lemma 4 used with $\overline{\kappa}'$ and since $\overline{\kappa}' \circ \overline{\kappa}' = \overline{\kappa}'$, we conclude that:

$$\overline{\kappa}'(\theta\overline{\mathcal{E}}) \vdash \texttt{(let (x e}_1\texttt{) e}_2\texttt{)} : \overline{\kappa}'\tau, \overline{\kappa}'\sigma$$

Case of (abs) By hypothesis $\mathcal{I}(\mathcal{E}, \kappa, \texttt{(lambda (x) e)}) = (\theta, \theta\alpha \xrightarrow{\varsigma} \tau, \emptyset, \kappa' \cup \{\varsigma \supseteq \sigma\})$. By the definition of the algorithm $\mathcal{I}$, this requires that $\mathcal{I}(\mathcal{E}_{\texttt{x}} + \{\texttt{x} \mapsto \alpha\}, \kappa, \texttt{e}) = (\theta, \tau, \sigma, \kappa')$. By induction hypothesis on $\texttt{e}$,

$$\overline{\kappa}'(\theta(\overline{\mathcal{E}}_{\texttt{x}} + \{\texttt{x} \mapsto \alpha\})) \vdash \texttt{e} : \overline{\kappa}'\tau, \overline{\kappa}'\sigma$$

Since ς is fresh, $\overline{\kappa}'(\sigma \cup \varsigma) = \overline{\kappa}'\sigma \cup \varsigma \supseteq \overline{\kappa}'\sigma$. Thus, by the rule (sub):

$$\overline{\kappa}'(\theta(\overline{\mathcal{E}}_{\texttt{x}} + \{\texttt{x} \mapsto \alpha\})) \vdash \texttt{e} : \overline{\kappa}'\tau, \overline{\kappa}'(\varsigma \cup \sigma)$$

Let us write $\kappa'' = \kappa' \cup \{\varsigma \supseteq \sigma\}$. By definition $\overline{\kappa}'' = \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\} \circ \overline{\kappa}'$. Since ς is fresh, we get:

$$\overline{\kappa}''(\theta(\overline{\mathcal{E}}_{\texttt{x}} + \{\texttt{x} \mapsto \alpha\})) \vdash \texttt{e} : \overline{\kappa}''\tau, \overline{\kappa}''\varsigma$$

By the rule (abs), we conclude that:

$$\overline{\kappa}''(\theta\overline{\mathcal{E}}) \vdash \texttt{(lambda (x) e)} : \overline{\kappa}''(\theta\alpha \xrightarrow{\varsigma} \tau), \emptyset$$

Case of (app) By hypothesis $\mathcal{I}(\mathcal{E}, \kappa, \texttt{(e}_1\texttt{ e}_2\texttt{)}) = (\theta, \theta_3\alpha, \sigma, \kappa')$. By definition of the algorithm $\mathcal{I}$, this requires that

$$\mathcal{I}(\mathcal{E}, \kappa, \texttt{e}_1) = (\theta_1, \tau_1, \sigma_1, \kappa_1), \quad \mathcal{I}(\theta_1\mathcal{E}, \kappa_1, \texttt{e}_2) = (\theta_2, \tau_2, \sigma_2, \kappa_2) \quad \text{and} \quad \theta_3 = \mathcal{U}_{\kappa_2}(\theta_2\tau_1, \tau_2 \xrightarrow{\varsigma} \alpha)$$

where $\sigma = \mathit{Observe}(\overline{\kappa}'(\theta\overline{\mathcal{E}}), \overline{\kappa}'(\theta_3\alpha))(\overline{\kappa}'\sigma_3)$, $\sigma_3 = \theta_3(\theta_2\sigma_1 \cup \sigma_2 \cup \varsigma)$ and $\kappa' = \theta_3\kappa_2$. By induction hypothesis on $\texttt{e}_1$, we get

$$\overline{\kappa}_1(\theta_1\overline{\mathcal{E}}) \vdash \texttt{e}_1 : \overline{\kappa}_1\tau_1, \overline{\kappa}_1\sigma_1$$

Since κ_2 extends $\theta_2\kappa_1$ and $\theta_3 = \mathcal{U}_{\kappa_2}(\theta_2\tau_1, \tau_2 \xrightarrow{\varsigma} \alpha)$, $\kappa' = \theta_3\kappa_2$ is well-formed. Since κ' extends $(\theta_3(\theta_2\kappa_1)$, $\overline{\kappa}'$ satisfies $(\theta_3(\theta_2\kappa_1)$. Thus $\overline{\kappa}' \circ \theta_3 \circ \theta_2$ satisfies κ_1. By the lemma 14 on κ_1, we have $\overline{\kappa}' \circ \theta_3 \circ \theta_2 = \overline{\kappa}' \circ \theta_3 \circ \theta_2 \circ \overline{\kappa}_1$. By the lemma 4 used with $\overline{\kappa}' \circ \theta_3 \circ \theta_2$, we get:

$$\overline{\kappa}'(\theta\overline{\mathcal{E}}) \vdash \texttt{e}_1 : \overline{\kappa}'(\theta_3(\theta_2\tau_1)), \overline{\kappa}'(\theta_3(\theta_2\sigma_1))$$

Since $\theta_3 = \mathcal{U}_{\kappa_2}(\theta_2\tau_1, \tau_2 \xrightarrow{\varsigma} \alpha)$ then $\theta_3\kappa_2$ is well-formed and $\theta_3(\theta_2\tau_1) = \theta_3(\tau_2 \xrightarrow{\varsigma} \alpha)$ by the correctness lemma on unification, yielding:

$$\overline{\kappa}'(\theta\overline{\mathcal{E}}) \vdash \mathtt{e}_1 : \overline{\kappa}'(\theta_3\tau_2) \overset{\overline{\kappa}'(\theta_3\varsigma)}{\to} \overline{\kappa}'(\theta_3\alpha), \overline{\kappa}'(\theta_3(\theta_2\sigma_1))$$

Since $\overline{\theta_1\mathcal{E}} = \theta_1\overline{\mathcal{E}}$ and by induction hypothesis on $\mathtt{e}_2$ with $\theta_1\mathcal{E}$,

$$\overline{\kappa}_2(\theta_2(\theta_1\overline{\mathcal{E}})) \vdash \mathtt{e}_2 : \overline{\kappa}_2\tau_2, \overline{\kappa}_2\sigma_2$$

Since $\overline{\kappa}'$ satisfies $\theta_3\kappa_2$, $\overline{\kappa}' \circ \theta_3$ satisfies κ_2. By the lemma 14 using $\overline{\kappa}_2$, we have $\overline{\kappa}' \circ \theta_3 = \overline{\kappa}' \circ \theta_3 \circ \overline{\kappa}_2$. By lemma 4 used with $\overline{\kappa}' \circ \theta_3$, we get:

$$\overline{\kappa}'(\theta\overline{\mathcal{E}}) \vdash \mathtt{e}_2 : \overline{\kappa}'(\theta_3\tau_2), \overline{\kappa}'(\theta_3\sigma_2)$$

By definition of σ_3 and the rule (app),

$$\overline{\kappa}'(\theta\overline{\mathcal{E}}) \vdash \mathtt{(e}_1\ \mathtt{e}_2\mathtt{)} : \overline{\kappa}'(\theta_3\alpha), \overline{\kappa}'\sigma_3$$

Since $\sigma = \mathit{Observe}(\overline{\kappa}'(\theta\overline{\mathcal{E}}), \overline{\kappa}'(\theta_3\alpha))(\overline{\kappa}'\sigma_3)$ and by the rule (sub),

$$\overline{\kappa}'(\theta\overline{\mathcal{E}}) \vdash \mathtt{(e}_1\ \mathtt{e}_2\mathtt{)} : \overline{\kappa}'(\theta_3\alpha), \sigma$$

By the lemma 4 used with $\overline{\kappa}'$, we get:

$$\overline{\kappa}'(\theta\overline{\mathcal{E}}) \vdash \mathtt{(e}_1\ \mathtt{e}_2\mathtt{)} : \overline{\kappa}'(\theta_3\alpha), \overline{\kappa}'\sigma \ \Box$$

The completeness theorem states that the inferred type is principal with respect to substitutions on variables and that the reconstructed effect is minimal with respect to subsumption on effects.

Theorem 3 (Completeness) *Let $\mathcal{E}$ and κ be well-formed and θ'' be a model of κ. If $\theta''\overline{\mathcal{E}} \vdash \mathtt{e} : \tau', \sigma'$ then $\mathcal{I}(\mathcal{E}, \kappa, \mathtt{e}) = (\theta, \tau, \sigma, \kappa')$ and there exists $\theta' \models \kappa'$ such that $\theta''\overline{\mathcal{E}} = \theta'(\theta\overline{\mathcal{E}})$, $\tau' = \theta'\tau$ and $\sigma' \supseteq \theta'\sigma$*

Proof The proof is by induction on the structure of expressions.

Case of (var) By hypothesis, $\theta''\overline{\mathcal{E}} \vdash \mathtt{x} : \tau', \sigma'$. This requires that $\tau' \preceq \theta''\overline{\mathcal{E}}(\mathtt{x})$ by definition of the rule (var) and that $\mathcal{E}(\mathtt{x}) = \forall\vec{v}.(\tau, \kappa')$ by definition of $\overline{\mathcal{E}}$. By definition of the algorithm $\mathcal{I}'$, we get:

$$\mathcal{I}(\mathcal{E}, \kappa, \mathtt{x}) = (\mathit{Id}, \theta\tau, \emptyset, \kappa \cup \theta\kappa') \quad \text{and} \quad \theta = \{\vec{v} \mapsto \vec{v'}\}$$

By hypothesis, we have that $\tau' \preceq \theta''\overline{\mathcal{E}}(\mathtt{x})$. By definition of $\preceq$, this requires that there exists a substitution θ'_1 defined on $\vec{v}$ such that:

$$\theta'_1(\theta''(\overline{\kappa}'\tau)) = \tau'$$

By definition of the algorithm $\mathcal{I}'$, θ substitutes $\vec{v}$ with fresh $\vec{v'}$. Let θ'_2 be defined on $\vec{v'}$ by $\theta'_2(\vec{v'}) = \theta'_2(\theta(\vec{v})) = \theta'_1(\vec{v'})$. Avoiding capture, the substitution θ'_2 satisfies:

$$\theta'_2(\theta''(\theta(\overline{\kappa}'\tau))) = \tau'$$

Since θ renames $\vec{v}$ with fresh $\vec{v'}$, we have $\theta \circ \overline{\kappa}' = \overline{\theta\kappa'} \circ \theta$. As a consequence, $\theta' = \theta'_2 \circ \theta'' \circ \overline{\theta\kappa'}$ satisfies $\kappa \cup \theta\kappa'$ and is such that:

$$\tau' = \theta'(\theta\tau) \quad \text{and} \quad \theta''\overline{\mathcal{E}} = \theta'\overline{\mathcal{E}} \quad \text{and} \quad \sigma' \supseteq \emptyset$$

Case of (let) By hypothesis, $\theta''\overline{\mathcal{E}} \vdash \mathtt{(let\ (x\ e_1)\ e_2)} : \tau', \sigma'_1 \cup \sigma'_2$. By definition of the rule (let), this requires that

$$\theta''\overline{\mathcal{E}} \vdash \mathtt{e}_1 : \tau'_1, \sigma'_1 \quad \text{and} \quad \theta''\overline{\mathcal{E}_\mathtt{x}} + \{\mathtt{x} \mapsto \mathit{Gen}(\sigma'_1, \theta''\overline{\mathcal{E}})(\tau'_1)\} \vdash \mathtt{e}_2 : \tau', \sigma'_2$$

By induction hypothesis on $\mathtt{e}_1$, $\mathcal{I}(\mathcal{E}, \kappa, \mathtt{e}_1) = (\theta_1, \tau_1, \sigma_1, \kappa_1)$ and there exists a substitution θ'_1 satisfying κ_1 such that:

$$\theta''\overline{\mathcal{E}} = \theta'_1(\theta_1\overline{\mathcal{E}}) \qquad \tau'_1 = \theta'_1\tau_1 \quad \text{and} \quad \sigma'_1 \supseteq \theta'_1\sigma_1$$

Thus, by definition of Gen,

$$\forall\tau, \quad \tau \preceq \mathit{Gen}(\sigma'_1, \theta''\overline{\mathcal{E}})(\tau'_1) \;\Rightarrow\; \tau \preceq \mathit{Gen}(\theta'_1\sigma_1, \theta'_1(\theta_1\overline{\mathcal{E}}))(\theta'_1\tau_1)$$

As a consequence, given that $\theta''\overline{\mathcal{E}_\mathtt{x}} = \theta'_1(\theta_1\overline{\mathcal{E}_\mathtt{x}})$,

$$\theta'_1(\theta_1\overline{\mathcal{E}_\mathtt{x}}) + \{\mathtt{x} \mapsto \mathit{Gen}(\theta'_1\sigma_1, \theta'_1(\theta_1\overline{\mathcal{E}}))(\theta'_1\tau_1)\} \vdash \mathtt{e}_2 : \tau', \sigma'_2$$

Let $(\forall\vec{v}.(\tau_1, \kappa'_1), \kappa''_1) = \mathit{Gen}_{\kappa_1}(\sigma_1, \theta_1\mathcal{E})(\tau_1)$ and define θ''_1 by Id on $\vec{v}$ and by θ'_1 elsewhere. By definition of θ''_1,

$$\theta''_1(\theta_1\overline{\mathcal{E}}) = \theta'_1(\theta_1\overline{\mathcal{E}}) \quad \text{and} \quad \theta''_1\sigma_1 = \theta'_1\sigma_1$$

Let us consider any τ such that $\tau \preceq \mathit{Gen}(\theta'_1\sigma_1, \theta'_1(\theta_1\overline{\mathcal{E}}))(\theta'_1\tau_1)$. By definition of $\preceq$, there exists a substitution θ defined on $\mathit{fv}(\theta'_1\tau_1) \setminus (\mathit{fv}(\theta'_1\sigma_1) \cup \mathit{fv}(\theta'_1(\theta_1\overline{\mathcal{E}})))$ such that

$$\tau = \theta(\theta'_1\tau_1)$$

Let us define θ' by $\theta \circ \theta'_1$ on $\vec{v}$. Since θ is defined on $\mathit{fv}(\theta'_1\tau_1) \setminus (\mathit{fv}(\theta'_1\sigma_1) \cup \mathit{fv}(\theta'_1(\theta_1\overline{\mathcal{E}})))$, any variable v' in $\mathit{fv}(\theta'_1 v)$ satisfies $\theta v' = v'$. Thus

$$\theta \circ \theta'_1 = \theta' \circ \theta''_1$$

Since θ'_1 satisfies κ_1, by the lemma 14, $\theta'_1 = \theta'_1 \circ \overline{\kappa}_1 = \theta'_1 \circ \overline{\kappa}'_1$. Thus,

$$\tau = \theta(\theta'_1\tau_1) = \theta(\theta'_1(\overline{\kappa}'_1\tau_1)) = \theta'(\theta''_1(\overline{\kappa}'_1\tau_1))$$

By definition of $\preceq$, this implies that:

$$\tau \preceq \theta''_1(\forall\vec{v}.\overline{\kappa}'_1\tau_1)$$

We have shown that for any τ such that $\tau \preceq \mathit{Gen}(\theta'_1\sigma_1, \theta'_1(\theta_1\overline{\mathcal{E}}))(\theta'_1\tau_1)$, $\tau \preceq \theta''_1(\forall\vec{v}.\overline{\kappa}'_1\tau_1)$. Thus, since $\theta'_1(\theta_1\overline{\mathcal{E}_{\mathtt{x}}}) + \{\mathtt{x} \mapsto \mathit{Gen}(\theta'_1\sigma_1, \theta'_1(\theta_1\overline{\mathcal{E}}))(\theta'_1\tau_1)\} \vdash \mathsf{e}_2 : \tau', \sigma'_2$, there exists a derivation of:

$$\theta''_1(\overline{\theta_1\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \overline{\forall\vec{v}.(\tau_1, \kappa'_1)}\}) \vdash \mathsf{e}_2 : \tau', \sigma'_2$$

Since $\overline{\theta_1\mathcal{E}_{\mathtt{x}}} = \theta_1\overline{\mathcal{E}_{\mathtt{x}}}$, since θ'_1 satisfies κ''_1 and by induction hypothesis on e_2 with κ''_1 and $\theta_1\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \forall\vec{v}.(\tau_1, \kappa'_1)\}$,

$$\mathcal{I}(\theta_1\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \forall\vec{v}.(\tau_1, \kappa'_1)\}, \kappa''_1, \mathsf{e}_2) = (\theta_2, \tau, \sigma_2, \kappa')$$

and there exists a substitution θ'_2 satisfying κ' such that $\tau' = \theta'_2\tau$, $\sigma'_2 \supseteq \theta'_2\sigma_2$ and

$$\begin{aligned}\theta''_1(\overline{\theta_1\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \overline{\forall\vec{v}.(\tau_1, \kappa'_1)}\}) &= \theta'_2(\theta_2\overline{\theta_1\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \overline{\forall\vec{v}.(\tau_1, \kappa'_1)}\}) \\ &= \theta'_2(\theta_2\theta_1\overline{\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \forall\vec{v}.(\overline{\kappa}'_1\tau_1)\})\end{aligned}$$

Let us write $\theta = \theta_2 \circ \theta_1$, $\sigma_3 = \theta_2\sigma_1 \cup \sigma_2$ and $\sigma = \mathit{Observe}(\overline{\kappa}'(\theta\overline{\mathcal{E}}), \overline{\kappa}'\tau)(\overline{\kappa}'\sigma_3)$. By definition of the algorithm, we get that:

$$\mathcal{I}(\mathcal{E}, \kappa, \mathtt{(let\ (x\ }\mathsf{e}_1\mathtt{)\ }\mathsf{e}_2\mathtt{)}) = (\theta, \tau, \sigma, \kappa')$$

Let V be the free variables of $\overline{\kappa}'\theta\overline{\mathcal{E}}$, $\overline{\kappa}'\tau_2$ and $\overline{\kappa}'\sigma_2$. Define θ' by θ'_2 on V and θ''_1 otherwise. By definition, θ' satisfies κ' and satisfies

$$\tau' = \theta'\tau \qquad \theta''\overline{\mathcal{E}} = \theta'(\theta\overline{\mathcal{E}}) \quad \text{and} \quad \sigma'_1 \cup \sigma'_2 \supseteq \theta'\sigma_3$$

Since $\theta'\sigma = \theta'\mathit{Observe}(\overline{\kappa}'(\theta\overline{\mathcal{E}}), \overline{\kappa}'\tau)(\overline{\kappa}'\sigma_3)$ and $\theta' = \theta' \circ \overline{\kappa}'$, then, by the lemma 3,

$$\theta'\mathit{Observe}(\overline{\kappa}'(\theta\overline{\mathcal{E}}), \overline{\kappa}'\tau)(\overline{\kappa}'\sigma_3) \subseteq \mathit{Observe}(\theta'(\theta\overline{\mathcal{E}}), \theta'\tau)(\theta'\sigma_3)$$

We conclude that $\sigma'_1 \cup \sigma'_2 \supseteq \theta'\sigma$. We have proved that $\mathcal{I}(\mathcal{E}, \kappa, \mathtt{(let\ (x\ }\mathsf{e}_1\mathtt{)\ }\mathsf{e}_2\mathtt{)}) = (\theta, \tau, \sigma, \kappa')$ and that there exists a substitution θ' satisfying κ' such that:

$$\theta''\overline{\mathcal{E}} = \theta'(\theta\overline{\mathcal{E}}),\ \tau' = \theta'\tau \text{ and } \sigma'_1 \cup \sigma'_2 \supseteq \theta'\sigma$$

Case of (abs) By hypothesis, $\theta''\overline{\mathcal{E}} \vdash \mathtt{(lambda\ (x)\ e)} : \tau_i \xrightarrow{\sigma'} \tau_f, \sigma''$. By definition of the rule (abs), this requires that

$$\theta''\overline{\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau_i\} \vdash \mathtt{e} : \tau_f, \sigma'$$

With a fresh variable α, this is equivalent to:

$$(\theta'' \circ \{\alpha \mapsto \tau_i\})(\overline{\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\}) \vdash \mathtt{e} : \tau_f, \sigma'$$

By induction hypothesis on $\mathtt{e}$ since $\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\}$ is well-formed, we have that:

$$\mathcal{I}(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\}, \kappa, \mathtt{e}) = (\theta, \tau, \sigma, \kappa')$$

and there exists a substitution θ'_1 satisfying κ' such that:

$$(\theta'' \circ \{\alpha \mapsto \tau_i\})(\overline{\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\}) = \theta'_1(\theta(\overline{\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\})),\ \tau_f = \theta'_1\tau \quad \text{and} \quad \sigma' \supseteq \theta'_1\sigma$$

By definition of the algorithm, we get that:

$$\mathcal{I}(\mathcal{E}, \kappa, \mathtt{(lambda\ (x)\ e)}) = (\theta, \theta\alpha \xrightarrow{\varsigma} \tau, \emptyset, \kappa' \cup \{\varsigma \supseteq \sigma\})$$

where ς is new. Let us consider the model $\theta' = \theta'_1 \circ \{\varsigma \mapsto \sigma'\}$ of $\kappa' \cup \{\varsigma \supseteq \sigma\}$. We get:

$$\theta''\overline{\mathcal{E}} = \theta'(\theta\overline{\mathcal{E}}) \qquad \tau_i \xrightarrow{\sigma'} \tau_f = \theta'(\theta\alpha \xrightarrow{\varsigma} \tau) \quad \text{and} \quad \sigma'' \supseteq \emptyset$$

Case of (app) The hypothesis is $\theta''\overline{\mathcal{E}} \vdash \mathtt{(e_1\ e_2)} : \tau', \sigma'_1 \cup \sigma'_2 \cup \sigma'_3$. By the definition of the rule (app), this requires that,

$$\theta''\overline{\mathcal{E}} \vdash \mathtt{e}_1 : \tau'_2 \xrightarrow{\sigma'_3} \tau', \sigma'_1 \quad \text{and} \quad \theta''\overline{\mathcal{E}} \vdash \mathtt{e}_2 : \tau'_2, \sigma'_2$$

By induction hypothesis on $\mathtt{e}_1$, $(\theta_1, \tau_1, \sigma_1, \kappa_1) = \mathcal{I}(\mathcal{E}, \kappa, \mathtt{e}_1)$ and there exists θ'_1 satisfying κ_1 such that:

$$\theta''\overline{\mathcal{E}} = \theta'_1(\theta_1\overline{\mathcal{E}}) \qquad \tau'_1 = \tau'_2 \xrightarrow{\sigma'_3} \tau' = \theta'_1\tau_1 \quad \text{and} \quad \sigma'_1 \supseteq \theta'_1\sigma_1$$

Since $\theta_1\overline{\mathcal{E}} = \overline{\theta_1\mathcal{E}}$ and $\theta_1\mathcal{E}$ is well-formed, by induction hypothesis on $\mathtt{e}_2$, we get

$$\mathcal{I}(\theta_1\mathcal{E}, \kappa_1, \mathtt{e}_2) = (\theta_2, \tau_2, \sigma_2, \kappa_2)$$

and there exists θ'_2 satisfying κ_2 such that:

$$\theta''\overline{\mathcal{E}} = \theta'_2(\theta_2(\theta_1\overline{\mathcal{E}})) \qquad \tau'_2 = \theta'_2\tau_2 \quad \text{and} \quad \sigma'_2 \supseteq \theta'_2\sigma_2$$

Let V be the set of free variables in $\overline{\kappa}_2(\theta_2(\theta_1\overline{\mathcal{E}}))$, $\overline{\kappa}_2\tau_2$ and $\overline{\kappa}_2\sigma_2$. Take α and ς new and define θ'_3 as follows:

$$\theta'_3 v = \begin{cases} \theta'_2 v, & v \in V \\ \tau', & v = \alpha \\ \sigma'_3, & v = \varsigma \\ \theta'_1 v, & \text{otherwise} \end{cases}$$

By this definition, θ'_3 satisfies κ_2 and we get:

$$\theta''\overline{\mathcal{E}} = \theta'_3(\theta_2(\theta_1\overline{\mathcal{E}})) \qquad \tau'_2 \xrightarrow{\sigma'_3} \tau' = \theta'_3(\tau_2 \xrightarrow{\varsigma} \alpha) \quad \text{and} \quad \theta'_2\sigma_2 = \theta'_3\sigma_2$$

By definition of $\mathcal{I}$ every v in $\overline{\kappa}_1\tau_1$ is either fresh or in $\mathit{fv}(\overline{\kappa}_1\theta_1\overline{\mathcal{E}})$. Since $\theta'_3(\theta_2(\theta_1\overline{\mathcal{E}})) = \theta'_2(\theta_2(\theta_1\overline{\mathcal{E}})) = \theta'_1(\theta_1\overline{\mathcal{E}})$, for every v in $\mathit{fv}(\overline{\kappa}_1\theta_1\overline{\mathcal{E}})$, we have $\theta'_3(\theta_2 v) = \theta'_2(\theta_2 v) = \theta'_1 v$. For every fresh v, since $\theta_2 v = v$, we have $\theta'_3(\theta_2 v) = \theta'_3 v = \theta'_1 v$. Thus,

$$\tau'_2 \xrightarrow{\sigma'_3} \tau' = \theta'_3(\theta_2\tau_1) \qquad \theta'_1\sigma_1 = \theta'_3(\theta_2\sigma_1)$$

Since θ'_3 satisfies κ_2 and $\theta'_3(\theta_2\tau_1) = \theta'_3(\tau_2 \xrightarrow{\varsigma} \alpha)$, by the lemma 17, there exists a substitution θ_3 such that $\theta_3 = \mathcal{U}_{\kappa_2}(\theta_2\tau_1, \tau_2 \xrightarrow{\varsigma} \alpha)$ verifying:

$$\theta_3(\theta_2\tau_1) = \theta_3(\tau_2 \xrightarrow{\varsigma} \alpha)$$

and such that $\theta_3\kappa_2$ is well-formed. By the definition of the algorithm, we get:

$$\mathcal{I}'(\mathcal{E}, \kappa, \mathtt{(e_1\ e_2)}) = (\theta, \tau, \sigma_3, \kappa')$$

where $\theta = \theta_3 \circ \theta_2 \circ \theta_1$, $\tau = \theta_3\alpha$, $\sigma_3 = \theta_3(\theta_2\sigma_1 \cup \sigma_2 \cup \varsigma)$ and $\kappa' = \theta_3\kappa_2$. By the lemma 17, there exists a substitution θ' satisfying $\theta_3\kappa_2$ such that $\theta'_3 = \theta' \circ \theta_3$. We get:

$$\theta''\overline{\mathcal{E}} = \theta'(\theta\overline{\mathcal{E}}) \qquad \tau'_1 = \theta'(\theta_3(\tau_2 \xrightarrow{\varsigma} \alpha)) \quad \text{and} \quad \sigma'_1 \cup \sigma'_2 \cup \sigma'_3 \supseteq \theta'(\theta_3\sigma_3)$$

By definition of $\mathcal{I}$, we have that:

$$(\theta, \tau, \sigma, \kappa') = \mathcal{I}(\mathcal{E}, \kappa, \mathtt{(e_1\ e_2)}) \quad \text{and} \quad \sigma = \mathit{Observe}(\overline{\kappa}'\theta\overline{\mathcal{E}}, \overline{\kappa}'\tau)(\overline{\kappa}'\sigma_3)$$

By definition, $\sigma'_1 \cup \sigma'_2 \cup \sigma'_3 \supseteq \theta'\sigma_3$ and by the lemma 3,

$$\sigma'_1 \cup \sigma'_2 \cup \sigma'_3 \supseteq \mathit{Observe}(\theta'(\theta\overline{\mathcal{E}}), \theta'\tau)(\sigma') \supseteq \mathit{Observe}(\theta'(\theta\overline{\mathcal{E}}), \theta'\tau)(\theta'\sigma_3)$$

Since $\theta'_3 \models \kappa_2$ and $\theta'_3 = \theta' \circ \theta_3$, then, by definition of κ', we get that $\theta' \models \kappa'$. By the lemma 14, $\theta' \circ \overline{\kappa}' = \theta'$. By the lemma 3, this implies:

$$\sigma'_1 \cup \sigma'_2 \cup \sigma'_3 \supseteq \theta'\mathit{Observe}(\overline{\kappa}'\theta\overline{\mathcal{E}}, \overline{\kappa}'\tau)(\overline{\kappa}'\sigma_3) = \theta'\sigma$$

We can conclude that $(\theta, \tau, \sigma, \kappa') = \mathcal{I}(\mathcal{E}, \kappa, \mathtt{(e_1\ e_2)})$, $\theta''\overline{\mathcal{E}} = \theta'(\theta\overline{\mathcal{E}})$, $\tau' = \theta'\tau$ and $\sigma'_1 \cup \sigma'_2 \cup \sigma'_3 \supseteq \theta'\sigma$ □

10 Examples

By introducing some well-known examples of list processing procedures, we show that our type and effect system permits the assignment of the same type and effect to its functional and imperative implementations.

$$\begin{aligned}
&\texttt{nil} &&: \forall\alpha.\mathit{list}(\alpha)\\
&\texttt{cons} &&: \forall\alpha\varsigma.\alpha \times \mathit{list}(\alpha) \xrightarrow{\varsigma} \mathit{list}(\alpha)\\
&\texttt{car} &&: \forall\alpha\varsigma.\mathit{list}(\alpha) \xrightarrow{\varsigma} \alpha\\
&\texttt{null?} &&: \forall\alpha\varsigma.\mathit{list}(\alpha) \xrightarrow{\varsigma} \mathit{bool}\\
&\texttt{cdr} &&: \forall\alpha\varsigma.\mathit{list}(\alpha) \xrightarrow{\varsigma} \mathit{list}(\alpha)
\end{aligned}$$

List Processing Functions

We introduce the type $\mathit{list}(\tau)$ of immutable lists together with the following constant and functions for manipulating them. `nil` is the empty list. The predicate `null?` tests if a list is empty. The constructor `cons` pairs up an element of type α with a list of type $\mathit{list}(\alpha)$. The procedure `car` returns the first element of a list and `cdr` the rest.

Example Our first example is the function `fold`, that successively applies function `f` over every element of a list `l` and its intermediate result `i`.

```
(define fold (lambda (f i)
               (lambda (l) ((rec (loop l i)
                              (if (null? l) i
                                  (loop (cdr l) (f (car l) i))))
                            l i))))
```

By considering observable effects, implementing the function `fold` recursively or by a loop using temporary locatives does not affect its typing.

```
(define fold (lambda (f i)
               (lambda (l)
                 (let (result (new i))
                   (let (data (new l))
                     (until (null? (get data))
                       ((set result) (f (car (get data)) (get result)))
```

```
                  ((set data) (cdr (get data)))))
                (get result)))))
```

Both implementations of the function `fold` have type $\forall \alpha\alpha'\varsigma\varsigma'\varsigma''.(\alpha \times \alpha' \xrightarrow{\varsigma} \alpha') \times \alpha' \xrightarrow{\varsigma'} (\mathit{list}(\alpha) \xrightarrow{\varsigma\cup\varsigma''} \alpha')$. Then, if one defines the function `reverse` for reversing the elements of a list by `(lambda (l) ((fold cons nil) l))`, its polymorphic type is $\forall \alpha\varsigma.\mathit{list}(\alpha) \xrightarrow{\varsigma} \mathit{list}(\alpha)$ with both implementations of `fold` ■

Example In the same vein, we consider the typing of two implementations of the function `map`. First we implement `map` by using `reverse` and imperative constructs.

```
(define map (lambda (f l)
              (let (r (new nil))
                (let (x (new l))
                  (until (null? (get x))
                    ((set r) (cons (f (car (get x))) (get r)))
                    ((set x) (cdr (get x)))))
                (reverse (get r)))))
```

The details of the imperative implementation of the function `map` are similar to those of the function `fold` and we can also implement `map` by reusing the function `fold` defined above.

```
(define map (lambda (f l)
              ((fold (lambda (x r) (cons (f x) r)) nil) l)))
```

The same polymorphic type is assigned to both of them, by using our type and effect discipline. So, for instance, the application of `map` to the identity function and the empty list has the polymorphic type $\forall \alpha.\mathit{list}(\alpha)$ of `nil`.

```
(define nil1 (map (lambda (x) x) nil))
```

The application of `map` to `new` and `nil` has a monomorphic type $\mathit{list}(\mathit{ref}_\rho(\alpha))$, accounting for the use of the function `new` on a region ρ, with an observable effect $\mathit{init}(\rho, \alpha)$.

```
(define nil2 (map new nil))
```

11 Comparison with the Related Work

To give a detailed comparison of our system with the related work, the criterion of expressiveness seems at very first sight best suited, because it is a formal criterion. The relative expressiveness of a type system with respect to another defines itself as the capability of strictly accepting more programs. Such a proposition requires a formal proof and in most cases, it is thus easier to show the contrary by giving a counter example.

In practice, it turns out that the criterion of expressiveness is not appropriate and many examples, among which some are presented in the next section, show that there usually is no proper inclusion between type systems. Unfortunately, it appears that our system does not permit to recognize some sophisticated examples as sound (for example, the function `f`, section 8 or the function `id5`, section 11.1), although those examples could be recognized as such using, for example, the typing discipline described in [Tofte, 1987] or [Appel & Mac Queen, 1990].

Nonetheless, in order to argue in favor of our system, we have already carried out the comparison on practical situations, which aimed at demonstrating that the problem of integrating imperative features to a functional language is best viewed in terms of a type and effect system, because it permits reasoning at a more intuitive level about the problem of typing references in a language such as ML. The goal of integrating imperative and functional paradigms, which is a fundamental aspect in our system, is particularly well suited within a programming environment that supports separate compilation features and modular programming paradigms.

11.1 Comparative Examples

This section present a series of more or less sophisticated examples, adapted from a survey paper on this subject [O'Toole, 1990] and from [Leroy, 1992], that establishes frontiers between the related type systems. Note that the results of the system presented in [Jouvelot & Gifford, 1991] are given for programs with explicit polymorphic types.

```
(define id1 (let ((x (id 1))) rid))
(define id2 (lambda (y) ((rid id) y)))
(define id3 ((nop rid) id))
```

In these three examples, we write `rid` the "imperative" version of the identity function `(lambda (x) (get (new x)))` and `id` for the usual definition of the identity functional. We also use the function `nop` defined as below.

```
(define nop (lambda (f)
              (lambda (x)
                (let ((g (lambda (y) (f x))))
                  x))))
```

The next example, quoted from [Leroy, 1990], shows that the observation of effects can be useful for removing type dependencies introduced by otherwise dead-code. Our inference system generalizes the type of id1 below, contrarily to the ones defined in [Leroy, 1990] or [Wright, 1992], which are not able to deal with the spurious allocation (new x).

```
(lambda (z)
 (let ((id4 (lambda (x)
              (if true z (lambda (y) (begin (new x) y)))
              x)))
   (id4 id4)))
```

Nonetheless, one can force such an effect to be observable, as is shown in this other contrived example. An occurrence of the type of y appears on the arrow of the function type for f and is constrained to match the type of f which occurs in the context of id5. As a consequence, the type of id5 cannot be generalized.

```
(lambda (f)
  (let ((id5 (lambda (y)
               (let ((r (new y)))
                 (if true (lambda (z) (if true r (new y)) z) f)) y)))
    (id5 id5)))
```

In addition to this comparison, the example of recursive typing presented in section 8 (the function eta-ref below requires such a recursive typing) is well typed using the system of [Leroy, 1992]. Note that it would have been in ours, if we had chosen to make indirections between types and effects explicit in the static semantics, using constraint sets, as is done in the inference algorithm.

```
(define eta-ref (lambda (f)
                  (let (r (new f))
                    (if true f (lambda (x)
                                 (get (if true r (new f))) x)))))
```

11.2 Benchmarks

The following benchmark summarizes the discussion above and suggests that our type and effect discipline favorably competes with some earlier polymorphic type generalization policies.

Example	[Tofte]	[Appel]	[Jouvelot]	[Wright]	[Leroy, 90]	[Leroy, 92]	4.4
`id1`	no	yes	yes	yes	yes	yes	yes
`id2`	yes	no	yes	yes	yes	yes	yes
`id3`	no	no	yes	yes	yes	yes	yes
`id4`	yes	yes	no	no	no	yes	yes
`id5`	yes	yes	no	no	no	no	no
`(id1 id1)`	no	no	no	no	yes	yes	yes
`(id2 id2)`	no	no	no	no	yes	yes	yes
`(id3 id3)`	no	no	yes	yes	yes	yes	yes
`(id4 id4)`	no	no	no	no	no	yes	yes
`(id5 id5)`	no	no	no	no	no	no	no

Leroy's system, based on closure typing, essentially differs from ours, based on effect inference, in that it associates functions with a static information: the type of their free variables. Type generalization then relies on a chasing of dangerous types. This resembles the process of garbage collection, which chases referenced values and marks them before sweeping the rest.

In our approach, type generalization is expressed in a much more natural way, because it states what information is important: accessible reference values. Instead of chasing for every possibly referenced value at any time, as in a closure-typing system, we define a notion of accessibility in the static semantics, which is represented by an observation criterion.

This explains the differences arising in the examples `eta-ref`, defined in the above section 11.1, or `fake-ref`, below, adapted from [Leroy, 1992].

```
(define ref-id (lambda (x) (get x) x))
(define fake-ref (ref-id ((rec loop (x) (loop x)) 0)))
```

A type system based on closure typing cannot detect the fake character of the reference introduced in this example. A type system based on effect inference can. It detects that no initialization effects occur.

12 Extensions

An appealing direction for further extensions would be the treatment of first-class continuations. Continuations are objects that allow programs to capture current state of their evaluation, using the higher-order `callcc` operation: "call with current continuation", and to manipulate it, using the `throw` construct. These two very simple and general operations define sophisticated control structures that may be needed to implement interleaving or backtracking mechanisms, for instance.

Continuation objects were originally introduced in Scheme [Rees & al., 1988] and then, but not without trouble, in the implementation of Standard ML [Duba & al., 1991]. Continuation values where first proposed by [Duba & al., 1991] as an extension of Standard ML, and also implemented by an abstract data type. Latter, the implementation of continuations in Standard ML of New Jersey [Appel & Mac Queen, 1990] was shown unsound in the presence of type polymorphism [Harper & Lillibridge, 1992].

An accessible continuation value allows the evaluation of an expression to be restarted in a context different from the context in which it was typed. Thus, it appears very natural to restrict polymorphism for continuations in the same vein as for references: accessible continuations shall remain monomorphic.

In our typing discipline with effects, the typing of continuations can be integrated in a manner very similar to reference values by the following static semantics.

$$\begin{array}{rcl} \mathit{TypeOf}[\![\texttt{callcc}]\!] & = & \forall \alpha\rho\varsigma\varsigma'.(\mathit{cont}_\rho(\alpha) \xrightarrow{\varsigma} \alpha) \overset{\varsigma' \cup \varsigma \cup \mathit{comefrom}(\rho,\alpha)}{\rightarrow} \alpha \\ \mathit{TypeOf}[\![\texttt{throw}]\!] & = & \forall \alpha\alpha'\rho\varsigma\varsigma'.(\mathit{cont}_\rho(\alpha) \xrightarrow{\varsigma} \alpha) \overset{\varsigma' \cup \varsigma \cup \mathit{goto}(\rho,\alpha)}{\rightarrow} \alpha' \end{array}$$

Static Semantics for Continuations

We define $\mathit{cont}_\rho(\tau)$ to be the type of continuation values and the effects $\mathit{comefrom}(\rho, \tau)$, for capturing the current continuation, and $\mathit{goto}(\rho, \tau)$, for invoking a continuation, as in [Jouvelot & Gifford, 1989].

The addition of continuations to our language does not, however, show up as a very natural extension as far as the dynamic semantics and the proofs of consistency are concerned. The dynamic semantics needs to be completely reformulated in terms of a continuation based semantics, as in [Duba & al., 1991], or in terms of a reduction semantics, as in [Wright & Felleisen, 1992].

In [Leroy, 1992], following along the lines of [Duba & al., 1991], the author gives a "weak soundness" result for the extension of its typing discipline to continuation objects, which states that "a well-typed expressions cannot go *wrong*". Reduction semantics permits the formulation of a strong soundness result for continuations in the typing discipline of Standard ML [Felleisen & Friedman, 1989, Wright & Felleisen, 1992].

13 Conclusion

Imperative features are required to make functional programming realistic, but integrating polymorphic typing in an imperative language appears problematic. Our claim is that solving this tension is best tackled by using an effect systems. Effect systems answer the lack of specification for polymorphic typing in the presence of effects by approximating state transformations.

Using effects, our typing discipline reconstructs the principal type and the minimal observable effects of expressions. We use effect information to control type generalization. We use an observation criterion to precisely delimit the scope of side-effecting operations. Our observation criterion generalizes the abstraction properties of functions in the presence of a state. A function can use a mutable object locally in a given region without making mention of this region outside. Altogether, this allows type generalization to be performed in `let` expressions in a more efficient and uniform way than previous systems.

The initial design goal of polymorphic effect systems [Lucassen & Gifford, 1988] was to safely integrate functional and imperative constructs. We showed how effect systems can also be put to work for solving the problem of polymorphic type reconstruction in the presence of imperative constructs. Our typing discipline permits full integration of an imperative programming style in a polymorphic functional language.

Acknowledgements

We are grateful to Fritz Henglein and Mads Tofte (DIKU), Xavier Leroy and Didier Remy (INRIA), David K. Gifford (MIT) and Andrew K. Wright (Rice University) for insightful comments and valuable feedback on the work presented in this paper.

References

[Appel & Mac Queen, 1990] Appel, A. W., and Mac Queen, D. B. Standard ML Reference Manual. AT&T Bell Laboratories and Princeton University, October 1990.

[Damas, 1985] Damas, L. Type Assignment in Programming Languages. *Ph.D. Thesis*, University of Edinburgh, April 1985.

[Duba & al., 1991] Duba B. F., Harper R., and MacQueen D. Typing First-Class Continuations in ML. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*. ACM, New-York, 1991.

[Felleisen & Friedman, 1989] Felleisen, M., and Friedman, D. P. A Syntactic Theory of Sequential State. In *Theoretical Computer Science*, volume 69, number 3, pages 243-287, 1989.

[Gordon & al., 1979] Gordon, M.J., Milner, R., and Wadsworth, C.P. *Edinburgh LCF*. Lectures Notes in Computer Science No. 78. Springer Verlag, Berlin, 1979.

[Harper & Lillibridge, 1992] Harper, R., and Lillibridge, M. Polymorphic type assignment and CPS conversion. In *1992 SIGPLAN Continuations Workshop*, 1992.

[Gifford & al., 1987] Gifford, D. K., Jouvelot, P., Lucassen, J. M., and Sheldon, M. A. FX-87 Reference Manual. *MIT/LCS/TR-407*, MIT Laboratory for Computer Science, September 1987.

[Jouvelot & Gifford, 1989] Jouvelot, P., and Gifford, D. K. Reasoning about Continuations with Control Effects. In *1989 ACM SIGPLAN International Conference on Programming Language Design and Implementation*, Portland, June 1989.

[Jouvelot & Gifford, 1991] Jouvelot, P., and Gifford, D. K. Algebraic Reconstruction of Types and Effects. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*. ACM, New-York, 1991.

[Kahn, 1988] Kahn, G. Natural Semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237-257. Elsevier, 1988.

[Leroy, 1990] Leroy, X. The ZINC Experiment: an Economical Implementation of the ML Language. Technical report 117, INRIA, 1990.

[Leroy & Weis, 1991] LEROY, X., AND WEIS, P. Polymorphic Type Inference and Assignment. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages.* ACM, New-York, 1991.

[Leroy, 1991] LEROY, X. Private Communication, 1991.

[Leroy, 1992] LEROY, X. *Typage polymorphe d'un langage algorithmique.* Doctoral dissertation, Université Paris VII, 1992.

[Lucassen, 1987] LUCASSEN, J. M. Types and Effects, Towards the Integration of Functional and Imperative Programming. *MIT/LCS/TR-408* (Ph. D. Thesis). MIT Laboratory for Computer Science, August 1987.

[Lucassen & Gifford, 1988] LUCASSEN, J. M., AND GIFFORD, D. K. Polymorphic Effect Systems. In *Proceedings of the 1988 ACM Conference on Principles of Programming Languages.* ACM, New-York, 1988.

[Milner, 1978] MILNER, R. A Theory for Type Polymorphism in Programming. In *Journal of Computer and Systems Sciences*, Vol. 17, pages 348-375. 1978.

[Milner & al., 1990] MILNER, R., TOFTE, M., HARPER, R. The definition of Standard ML. *The MIT Press*, Cambridge, 1990.

[Milner, 1991] MILNER, R., AND TOFTE, M. Co-Induction in Relational Semantics. *Theoretical Computer Science*, 87:209-220, 1991.

[O'Toole, 1989] O'TOOLE, J. W. Polymorphic Type Reconstruction. *Master Thesis.* MIT Laboratory for Computer Science, May 1989.

[O'Toole, 1990] O'TOOLE, J. W. Type Abstraction Rules for References: a Comparison of Four Which Have Achieved Notoriety. Technical Report 390, MIT Laboratory for Computer Science, 1990.

[Plotkin, 1981] PLOTKIN, G. A structural Approach to Operational Semantics. *Technical report DAIMI-FN-19.* Aarhus University, 1981.

[Robinson, 1965] ROBINSON, J. A. A Machine Oriented Logic Based on the Resolution Principle. In *Journal of the ACM*, Vol. 12(1), pages 23-41. ACM, New-York, 1965.

[Rees & al., 1988] REES, J., AND CLINGER W., EDITORS. Fourth Report on the Algorithmic Language Scheme. September 1988.

[Talpin & Jouvelot, Sept. 1992] TALPIN, J. P., AND JOUVELOT, P. Polymorphic Type, Region and Effect Inference. In the *Journal of Functional Programming*, volume 2, number 3. Cambridge University Press, 1992.

[Talpin & Jouvelot, June 1992] TALPIN, J. P., AND JOUVELOT, P. The Type and Effect Discipline. In *the proceedings of the 1992's IEEE Conference on Logic in Computer Science*. Santa Cruz, California, June 1992.

[Tofte, 1987] TOFTE, M. Operational Semantics and Polymorphic Type Inference. *PhD Thesis* and Technical Report *ECS-LFCS-88-54*, University of Edinburgh, 1987.

[Wright & Felleisen, 1992] WRIGHT, A. K., AND FELLEISEN, M. A Syntactic Approach to Type Soundness. Technical Report *TR91-160*, Rice University, 1992.

[Wright, 1992] WRIGHT, A. K. Typing References by Effect Inference. In *the proceedings of the 1992's European Symposium on Programming*, volume 582 of the *Lectures Notes in Computer Science*, pages 473-491. Springer Verlag, 1992.