Polymorphic Type, Region and Effect Inference*

Jean-Pierre Talpin[1]
Pierre Jouvelot[1,2]

[1]CRI, Ecole Nationale Supérieure des Mines de Paris, France
[2]MIT Laboratory for Computer Science, USA

Abstract

We present a new static system that reconstructs the types, regions and effects of expressions in an implicitly typed functional language that supports imperative operations on reference values. Just as types structurally abstract collections of concrete values, *regions* represent sets of possibly aliased reference values and *effects* represent approximations of the imperative behavior on regions.

We introduce a static semantics for inferring types, regions and effects and prove that it is consistent with respect to the dynamic semantics of the language. We present a reconstruction algorithm that computes the types and effects of expressions and assigns regions to reference values. We prove the correctness of the reconstruction algorithm with respect to the static semantics. Finally, we discuss potential applications of our system to automatic stack allocation and parallel code generation.

1 Introduction

Type and effect reconstruction is the process that automatically determines the types and effects of expressions in a program. Types specify the structure of values denoted by expressions. Milner-style polymorphic type reconstruction [Milner] is a typical example for functional programming languages. It is the subject of much theoretical investigation and practical developments, in particular to extend it to imperative language constructs and module systems ([Tofte], [Harper], [Sheldon]). Effect systems [Lucassen] are such an extension. Similar to types, effects describe how expressions affect the store in a functional language extended with imperative constructs. Types and effects can be statically computed by algebraic reconstruction [Jouvelot].

Types provide useful information for both the programmer, who can describe the intended specification of its programs, and the compiler, which can use types to generate more efficient code by avoiding type tags. Effects, as generic abstractions of expression behaviors over sets of possibly aliased references (represented by regions), can be used to generate parallel code while preserving the sequential semantics of programs [Lucassen, Hammel]. They can also be used in code optimizations for standard architectures, e.g. for stack allocation of temporary data structures.

This paper builds upon both the ideas of algebraic reconstruction of effects and the ML-style type discipline to statically compute the store effects of expressions over inferred regions

*In the *Journal of Functional Programming*, Vol. 2, No. 2. Cambridge University Press, 1992.

of references. Our algorithm obtains for each expression its maximal type with respect to type substitutions, the lower bound of its effect, and assigns regions to reference values in a way that minimizes spurious aliasing among references.

The structure of the report is as follows. Section 2 presents the related work. We describe the syntax, the dynamic semantics (section 3) and the static semantics (section 4) of the language. In section 5, we state and prove that the static and dynamic semantics of the language are consistent. Section 6 presents our type, region and effect reconstruction algorithm the correctness of which is proved in section 7. Before concluding in section 9, we show how our algorithm works on a few examples (section 8).

2 Related Work

Our language is equivalent to Core-ML [Mitchell] extended to allow references. The classical way of dealing with non referentially transparent constructs is described in [Gordon] where some ad-hoc rules are introduced to avoid creating inconsistencies within the type system. [Tofte] introduces a nicer imperative type discipline within which types are categorized between applicative and imperative types; only applicative types can be generalized in `let` bindings. An extension of this approach, based on so-called weak type variables, is used inside the implementation of Standard ML done at Bell Labs [Appel]. Another extension is proposed by [Leroy] in which function types are labeled with sets of types that are used by reference values. The notions of regions and effects provide more intuitive information about programs and are presented here as a natural extension of the Hindley-Milner type discipline. Our static semantics thus gives a more straightforward abstraction of the dynamic semantics than [Leroy]'s system. However, since the problem of polymorphic type generalization escapes the scope of this paper, our system falls short of allowing some type-safe programs that are correctly seen as such by other systems.

Abstract interpretation [Cousot] is the usual framework to obtain a computable representation of the properties of program executions such as value aliasing and side-effects [Neirynck]. This approach usually requires complex representations of abstract states that consist of environment and store approximations via graphs. To deal with functional languages [Larus, Harrison, Deutsch], this approach is usually coupled with an interprocedural data flow analysis; this incurs a heavy computational cost [Rosen].

[Gifford] proposes a static semantics that includes a polymorphic type, region and effect checking system. However, the need to specify types, regions and effects are burdensome in real-life programs. [Jouvelot] shows that effect reconstruction can be seen as a constraint satisfaction problem, in the vein of [Morris] who used this approach for type reconstruction. However, the matching of effects required by the static semantics, together with the use of explicit polymorphism, imply the non-existence of syntactic principal types. Effect matching also somewhat limits the kind of accepted programs; the following example is not type correct in [Jouvelot]'s system but is in our's:

```
(if true (lambda (x) x) (lambda (x) (get (new x))))
```

Our system reconstructs the type and effect of such programs by the addition of subeffecting. *Subeffecting* is tantamount to subtyping in the domain of effects. It is required here since the latent effects of both arms of the conditional are different, but can be coerced to a common effect upper bound.

3 Dynamic Semantics

We present the syntax and dynamic semantics of our language.

3.1 Syntax

The syntax of expressions $\mathtt{e} \in \mathit{Exp}$ in the language is described below. It uses enclosing parentheses in the reminiscence of *Scheme* [Scheme] and shares its dynamic semantics with Core-ML language, in the ususal call-by-value fashion. We implement operations on references as special forms since they are of particular interest in the static semantics.

`e` ::=	`x` |	*value identifier*
	`(e e')` |	*application*
	`(lambda (x) e)` |	*abstraction*
	`(rec (f x) e)` |	*recursive function definition*
	`(let (x e) e')` |	*lexical binding*
	`(new e)` | `(get e)` | `(set e e')`	*initialization, dereference and assignment*

Language Syntax

3.2 Domains

The dynamic semantics is defined by a set of operational rules [Plotkin] that specify the evaluation of expressions.

Computable values are either the command value u, reference values l or closures. A closure c is composed of the syntactic value identifier of the argument, a body expression and the lexical environment E where it is defined. A store s is a finite map from references to values. A trace f is a set of labeled reference values that indicate initialized, read and written locations; a trace is the dynamic counterpart of a static side-effect (described in section 4).

$$
\begin{array}{rlll}
v \in & \mathit{Value} & = \{u\} + \mathit{Ref} + \mathit{Closure} & \mathit{values} \\
l \in & \mathit{Ref} & & \mathit{locations} \\
c \in & \mathit{Closure} & = \mathit{Id} \times \mathit{Exp} \times \mathit{Env} & \mathit{closures} \\
E \in & \mathit{Env} & = \mathit{Id} \stackrel{\mathit{fin}}{\to} \mathit{Value} & \mathit{environments} \\
s \in & \mathit{Store} & = \mathit{Ref} \stackrel{\mathit{fin}}{\to} \mathit{Value} & \mathit{stores} \\
f \in & \mathit{Trace} & = \mathcal{P}_{\mathit{fin}}(\mathit{init}(\mathit{Ref}) + \mathit{read}(\mathit{Ref}) + \mathit{write}(\mathit{Ref})) & \mathit{traces}
\end{array}
$$

Computable Values

3.3 Dynamic Semantics

Given a store s and an environment E, the dynamic semantics associates an expression $\mathtt{e}$ with the value v it computes, the trace f of the side-effects it performs during its evaluation and the possibly updated store s'. This is noted $s, E \vdash \mathtt{e} \to v, f, s'$.

For any map m, we note $Dom(()m)$ the domain of m, m_x the map m with x unbound, $\{x \mapsto v\}$ the map from x to v and $m \cup \{x \mapsto v\}$ the extension of m to x.

$$(var) : \frac{\mathtt{x} \in Dom(()E)}{s, E \vdash \mathtt{x} \to E(\mathtt{x}), \emptyset, s} \qquad (abs) : \frac{}{s, E \vdash (\mathtt{lambda\ (x)\ e}) \to \langle \mathtt{x}, \mathtt{e}, E_{\mathtt{x}} \rangle, \emptyset, s}$$

$$(rec) : \frac{c = \langle \mathtt{x}, \mathtt{e}, E_{\mathtt{f},\mathtt{x}} \cup \{\mathtt{f} \mapsto c\} \rangle}{s, E \vdash (\mathtt{rec\ (f\ x)\ e}) \to c, \emptyset, s} \qquad (app) : \frac{\begin{array}{l} s_0, E \vdash \mathtt{e} \to \langle \mathtt{x}, \mathtt{e}'', E' \rangle, f, s \\ s, E \vdash \mathtt{e}' \to v', f', s' \\ s', E' \cup \{\mathtt{x} \mapsto v'\} \vdash \mathtt{e}'' \to v'', f'', s'' \end{array}}{s_0, E \vdash (\mathtt{e\ e}') \to v'', f \cup f' \cup f'', s''}$$

$$(let) : \frac{s_0, E \vdash \mathtt{e} \to v, f, s \qquad s, E_{\mathtt{x}} \cup \{\mathtt{x} \mapsto v\} \vdash \mathtt{e}' \to v', f', s'}{s_0, E \vdash (\mathtt{let\ (x\ e)\ e}') \to v', f \cup f', s'}$$

$$(new) : \frac{s_0, E \vdash \mathtt{e} \to v, f, s \qquad l \notin Dom(()s)}{s_0, E \vdash (\mathtt{new\ e}) \to l, f \cup \{init(l)\}, s \cup \{l \mapsto v\}}$$

$$(get) : \frac{s_0, E \vdash \mathtt{e} \to l, f, s}{s_0, E \vdash (\mathtt{get\ e}) \to s(l), f \cup \{read(l)\}, s}$$

$$(set) : \frac{s_0, E \vdash \mathtt{e} \to l, f, s \qquad s, E \vdash \mathtt{e}' \to v, f', s'}{s_0, E \vdash (\mathtt{set\ e\ e}') \to u, f \cup f' \cup \{write(l)\}, s'_l \cup \{l \mapsto v\}}$$

Dynamic Semantics

4 Static Semantics

We present the static semantics of our language. We begin by defining the algebra of types and effects, and specify the static semantics. There are three static domains: regions, effects and types.

$$\begin{array}{lll} r & \in RegConst & \\ \gamma & \in RegVar & \\ \rho & \in Region & = RegConst + RegVar \\ \sigma & \in Effect & \sigma ::= \emptyset \mid init(\rho) \mid read(\rho) \mid write(\rho) \mid \sigma \cup \sigma \mid \varsigma \\ \tau & \in Type & \tau ::= unit \mid \alpha \mid ref_\rho(\tau) \mid \tau \xrightarrow{\sigma} \tau \end{array}$$

Regions, Effects and Types

The domain of regions ρ is the disjoint union of a countable set of constants and variables γ. Every data structure corresponds to a given region in the static semantics; this region abstracts the memory locations in which it will be allocated at run time. Two values are in the same region if they may share some memory locations.

Basic effects σ can either be the constant $\emptyset$ that represents the absence of effects, effect variables ς, or store effects $init(\rho)$, $read(\rho)$ or $write(\rho)$ that approximate memory side-effects on their region argument ρ. $init(\rho)$ denotes the allocation and initialization of a mutable reference value in the region ρ. The effect $read(\rho)$ describes accesses to references in the region ρ, while $write(\rho)$ represents assignments of values to references in the region ρ.

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 $\sqsupseteq$ of subsumption on effects: $\sigma \sqsupseteq \sigma'$ 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* describing the type of commands, type variables α, reference types $ref_\rho(\tau)$ 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 is the effect incurred when the function is applied: it encapsulates the side-effects of its body.

4.1 Type and Effect Rules

The inference rules of the static semantics associate a type environment $\mathcal{E}$ and an expression `e` with its possible types τ and effects σ, noted $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$.

Generic types can be created for variables that are bound in `let` forms to referentially transparent expressions. One way to statically enforce that such expressions are pure would be to require their effects to be $\emptyset$. We did not adopt this policy here since it would have required a non-deterministic backtrack-based inference algorithm, which would have departed too much from existing syntax-directed type reconstruction algorithms. Among various syntactic type generalization policies [Tofte, Harper], we chose the simplest one, based on the expansiveness property of expressions; a non-expansive expression is syntactically guaranteed to never allocate references.

Variables and lambda-abstractions are non-*expansive* expressions [Tofte]. By extension, a `let` expression is non-expansive if and only if both its binding expression and its body are non-expansive. We define the boolean function *expansive* for expansive expressions by induction:

$$
\begin{array}{rcl}
expansive[\![\mathtt{e}]\!] & = & case\ \mathtt{e}\ of \\
\mathtt{x} \mid (\mathtt{lambda\ (x)\ e'}) \mid (\mathtt{rec\ (f\ x)\ e}) & \Rightarrow & false \\
(\mathtt{new\ e'}) \mid (\mathtt{get\ e'}) \mid (\mathtt{set\ e'\ e''}) \mid (\mathtt{e'\ e''}) & \Rightarrow & true \\
(\mathtt{let\ (x\ e')\ e''}) & \Rightarrow & expansive[\![\mathtt{e'}]\!] \vee expansive[\![\mathtt{e''}]\!]
\end{array}
$$

Expansive Expressions

Non-expansive `let` expressions, which can be generalized over, are handled by syntactic substitution of the binding for the variable in the body. This avoids the complication of introducing sophisticated type schemes inside the static semantics that would mimic the algebraic type schemes used in the algorithm. Indeed, this simple technique provides an equivalent way of expressing the property that non expansive expressions may admit multiple types. Even though the static semantics of `let` expressions uses explicit syntactic substitution, the reconstruction algorithm works very much like an ordinary Hindley-Milner type inferencer does when it handles `let`. Type environments $\mathcal{E}$ are finite maps from identifiers to types.

We write $\mathtt{e}'[\mathtt{e}/\mathtt{x}]$ for the textual substitution of $\mathtt{e}$ for $\mathtt{x}$ in $\mathtt{e}'$ with bound variables renamed as usual. Subeffecting is introduced by the *(does)* rule. Note that this rule can be used whenever a type or effect mismatch exists in the application rule *(app)* and the assignment rule *(set)*.

$$(var): \frac{\mathtt{x} \mapsto \tau \in \mathcal{E}}{\mathcal{E} \vdash \mathtt{x} : \tau, \emptyset} \qquad (rec): \frac{\mathcal{E}_{\mathtt{f},\mathtt{x}} \cup \{\mathtt{f} \mapsto \tau \xrightarrow{\sigma} \tau'\} \cup \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e} : \tau', \sigma}{\mathcal{E} \vdash (\mathtt{rec\ (f\ x)\ e}) : \tau \xrightarrow{\sigma} \tau', \emptyset}$$

$$(abs): \frac{\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e} : \tau', \sigma}{\mathcal{E} \vdash (\mathtt{lambda\ (x)\ e}) : \tau \xrightarrow{\sigma} \tau', \emptyset} \qquad (app): \frac{\mathcal{E} \vdash \mathtt{e} : \tau \xrightarrow{\sigma''} \tau', \sigma \quad \mathcal{E} \vdash \mathtt{e}' : \tau, \sigma'}{\mathcal{E} \vdash (\mathtt{e\ e}') : \tau', \sigma \cup \sigma' \cup \sigma''}$$

$$(let): \frac{\begin{array}{l}\neg expansive[\![\mathtt{e}]\!] \\ \mathcal{E} \vdash \mathtt{e} : \tau, \emptyset \\ \mathcal{E} \vdash \mathtt{e}'[\mathtt{e}/\mathtt{x}] : \tau', \sigma'\end{array}}{\mathcal{E} \vdash (\mathtt{let\ (x\ e)\ e}') : \tau', \sigma'} \qquad (ilet): \frac{\begin{array}{l}expansive[\![\mathtt{e}]\!] \\ \mathcal{E} \vdash \mathtt{e} : \tau, \sigma \\ \mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e}' : \tau', \sigma'\end{array}}{\mathcal{E} \vdash (\mathtt{let\ (x\ e)\ e}') : \tau', \sigma \cup \sigma'}$$

$$(does): \frac{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma \qquad \sigma' \sqsupseteq \sigma}{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma'} \qquad (new): \frac{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma}{\mathcal{E} \vdash (\mathtt{new\ e}) : ref_\rho(\tau), \sigma \cup init(\rho)}$$

$$(get): \frac{\mathcal{E} \vdash \mathtt{e} : ref_\rho(\tau), \sigma}{\mathcal{E} \vdash (\mathtt{get\ e}) : \tau, \sigma \cup read(\rho)} \qquad (set): \frac{\mathcal{E} \vdash \mathtt{e} : ref_\rho(\tau), \sigma \quad \mathcal{E} \vdash \mathtt{e}' : \tau, \sigma'}{\mathcal{E} \vdash (\mathtt{set\ e\ e}') : unit, \sigma \cup \sigma' \cup write(\rho)}$$

Static Semantics

5 Consistency of dynamic and static semantics

We use the proof method introduced in [Tofte] to show that the static and dynamic semantics are consistent with respect to a structural relation between values and types, defined as the maximal fixed point of a monotonic property.

We introduce store models $\mathcal{S}$ to tell which region ρ and type τ correspond to a reference value l:

$$\mathcal{S} \in StoreModel = Ref \xrightarrow{fin} Region \times Type$$

We note $\mathcal{S} \subseteq \mathcal{S}'$ if and only if $\forall l \in Dom(()\mathcal{S}),\ \mathcal{S}(l) = \mathcal{S}'(l)$.

Definition 1 (Effects consistency) *A dynamic trace of side effects $f \in Trace$ is consistent with the effect $\sigma \in Effect$ for the model $\mathcal{S} \in StoreModel$, noted $\mathcal{S} \models f : \sigma$, if and only if:*

$$\begin{array}{l}\forall\ init(l) \in f,\ \mathcal{S}(l) = (\rho, \tau)\ \wedge\ init(\rho) \in \sigma \\ \forall\ read(l) \in f,\ \mathcal{S}(l) = (\rho, \tau)\ \wedge\ read(\rho) \in \sigma \\ \forall\ write(l) \in f,\ \mathcal{S}(l) = (\rho, \tau)\ \wedge\ write(\rho) \in \sigma\end{array}$$

Note that, if $\mathcal{S} \subseteq \mathcal{S}'$ and $\mathcal{S} \models f : \sigma$, then $\mathcal{S}' \models f : \sigma$. Also, when $\mathcal{S} \models f : \sigma$ and $\mathcal{S} \models f' : \sigma'$, then $\mathcal{S} \models f \cup f' : \sigma \cup \sigma'$.

We define typed stores as models for describing the relation between values and types.

$$s : \mathcal{S} \in \mathit{TypedStore} = \mathit{Store} \times \mathit{StoreModel}$$

Definition 2 (Consistent values and types) *Given a typed store $s : \mathcal{S}$, the value v is* consistent *with the type τ, noted $s : \mathcal{S} \models v : \tau$, if and only if v and τ verify one of the following properties:*

$$\begin{array}{l} s : \mathcal{S} \models u : \mathit{unit} \\ s : \mathcal{S} \models l : \mathit{ref}_\rho(\tau) \Leftrightarrow \mathcal{S}(l) = (\rho, \tau) \text{ and } s : \mathcal{S} \models s(l) : \tau \\ s : \mathcal{S} \models \langle \mathtt{x}, \mathtt{e}, E \rangle : \tau \Leftrightarrow \text{there exists } \mathcal{E} \text{ and } s : \mathcal{S} \models E : \mathcal{E} \text{ and } \mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \tau, \emptyset \end{array}$$

We note $s : \mathcal{S} \models E : \mathcal{E}$ if and only if $Dom(()E) = Dom(()\mathcal{E})$ and $s : \mathcal{S} \models E(\mathtt{x}) : \mathcal{E}(\mathtt{x})$ for every $\mathtt{x} \in Dom(()E)$.

As shown in [Tofte], this structural property between values and types does not uniquely define a relation and must be regarded as a fixed point equation on the domain $\mathcal{R} = \mathit{TypedStore} \times \mathit{Value} \times \mathit{Type}$ of the relation. We define a function $\mathcal{F}$ on $\mathcal{P}_{\mathit{fin}}(\mathcal{R}) \rightarrow \mathcal{P}_{\mathit{fin}}(\mathcal{R})$; Its fixed points are the relations on $\mathcal{R}$ that verify the property defined above.

$$\begin{array}{l} \mathcal{F}(\mathcal{Q}) = \{(s, \mathcal{S}, v, \tau) \backslash \\ \quad \textit{if } v = u \textit{ then } \tau = \mathit{unit} \\ \quad \textit{if } v = l \textit{ then there exist } \rho \textit{ and } \tau' \textit{ such that} \\ \quad\quad \tau = \mathit{ref}_\rho(\tau') \textit{ and } \mathcal{S}(l) = (\rho, \tau') \textit{ and } (s, \mathcal{S}, s(v), \tau') \in \mathcal{Q} \\ \quad \textit{if } v = \langle \mathtt{x}, \mathtt{e}, E \rangle \textit{ then there exists } \mathcal{E} \textit{ such that} \\ \quad\quad s : \mathcal{S} \models E : \mathcal{E} \textit{ and } \mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \tau, \emptyset\} \end{array}$$

In order to guarantee the existence of fixed points for $\mathcal{F}$, it is sufficient to show that $\mathcal{F}$ is monotonic.

Lemma 1 (Monotony of $\mathcal{F}$) *If $\mathcal{Q} \subseteq \mathcal{Q}'$ then $\mathcal{F}(\mathcal{Q}) \subseteq \mathcal{F}(\mathcal{Q}')$.*

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

- If $v = u$, then $q \in \mathcal{F}(\mathcal{Q}')$ by definition.
- If $v \in \mathit{Ref}$, then there exist ρ and τ' such that $\tau = \mathit{ref}_\rho(\tau')$, $\mathcal{S}(v) = (\rho, \tau')$ and $(s, \mathcal{S}, s(v), \tau') \in \mathcal{Q}$. Since $\mathcal{Q} \subseteq \mathcal{Q}'$, we have $q \in \mathcal{F}(\mathcal{Q}')$.
- Finally, if $v \in \mathit{Closure}$, then $v = \langle \mathtt{x}, \mathtt{e}, E \rangle$ and there exists a type environment $\mathcal{E}$ such that $s : \mathcal{S} \models E : \mathcal{E}$, so that $q \in \mathcal{F}(\mathcal{Q}')$ □

Among the fixed points of $\mathcal{F}$, we choose the greatest fixed point $\mathit{gfp}(\mathcal{F})$ as our relation; $\mathit{gfp}(\mathcal{F})$ is defined by:

$$\mathit{gfp}(\mathcal{F}) = \cup\{\mathcal{Q} \subseteq \mathcal{R} \setminus\ \mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})\}$$

A set $\mathcal{Q}$ such that $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$ is called *$\mathcal{F}$-consistent.*

The relation between types and values is thus defined by:

$$s : \mathcal{S} \models v : \tau \;\Leftrightarrow\; (s, \mathcal{S}, v, \tau) \in \mathit{gfp}(\mathcal{F})$$

In order to use induction in the consistency proof, we need to check that the relation between a type and a value, whenever correct for some typed store $s : \mathcal{S}$, is preserved when the store is properly expanded. We note:

$$s : \mathcal{S} \sqsubseteq s' : \mathcal{S}' \;\Leftrightarrow\; \mathcal{S} \subseteq \mathcal{S}' \textit{ and, for all } v \textit{ and } \tau,\; s : \mathcal{S} \models v : \tau \Rightarrow s' : \mathcal{S}' \models v : \tau$$

Lemma 2 (Side Effects) *Assume $s : \mathcal{S} \models v : \tau$. If $\mathcal{S}(l) = (\rho, \tau)$, then $s : \mathcal{S} \sqsubseteq s_l \cup \{l \mapsto v\} : \mathcal{S}_l \cup \{l \mapsto (\rho, \tau)\}$. Otherwise, for every region ρ, $s : \mathcal{S} \sqsubseteq s \cup \{l \mapsto v\} : \mathcal{S} \cup \{l \mapsto (\rho, \tau)\}$.*

Proof We only consider here the first case to be non-trivial. The proof is by induction on the structure of typings and values. Define $s' = s_l \cup \{l \mapsto v\}$ and $\mathcal{S}' = \mathcal{S}_l \cup \{l \mapsto (\rho, \tau)\}$. We have to show that $s : \mathcal{S} \sqsubseteq s' : \mathcal{S}'$, i.e., $s' : \mathcal{S}' \models v' : \tau'$ from the hypothesis $s : \mathcal{S} \models v' : \tau'$, $s \subseteq s'$ and $\mathcal{S} \subseteq \mathcal{S}'$.

We consider the typed store $s : \mathcal{S}$, and $\mathcal{Q} \subseteq \mathcal{R}$ such that $\mathcal{Q} = \{(s', \mathcal{S}', v', \tau') \setminus s : \mathcal{S} \models v' : \tau'\}$. We show that $\mathcal{Q}$ is $\mathcal{F}$-consistent, i.e., that $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$. Let q be $(s', \mathcal{S}', v', \tau')$ in $\mathcal{Q}$:

- If $v' = u$, then $q \in \mathcal{F}(\mathcal{Q})$.
- If v' is a reference, by definition of $s : \mathcal{S} \models v' : \tau'$, there exist ρ' and τ'' such that $\tau' = \mathit{ref}_{\rho'}(\tau'')$, $\mathcal{S}(v') = (\rho', \tau'')$ and $s : \mathcal{S} \models s(v') : \tau''$. Since $s \subseteq s'$ and $\mathcal{S} \subseteq \mathcal{S}'$ then $\mathcal{S}'(v') = (\rho', \tau'')$ and $s : \mathcal{S} \models s'(v') : \tau''$, so that $(s', \mathcal{S}', s'(v'), \tau'') \in \mathcal{Q}$ and $q \in \mathcal{F}(\mathcal{Q})$.
- Finally, if $v' = \langle \mathtt{x}, \mathtt{e}, E \rangle$, then there exists a type environment $\mathcal{E}$ such that $s : \mathcal{S} \models E : \mathcal{E}$. This means that $s : \mathcal{S} \models E(\mathtt{x}) : \mathcal{E}(\mathtt{x})$ for every $\mathtt{x} \in \mathit{Dom}(()E)$. Thus, by definition of $\mathcal{Q}$, we have $(s', \mathcal{S}', E(\mathtt{x}), \mathcal{E}(\mathtt{x})) \in \mathcal{Q}$, so that $q \in \mathcal{F}(\mathcal{Q})$ □

Theorem 1 (Consistency of dynamic and static semantics) *Let E be an environment and $\mathcal{E}$ its type. Let $s : \mathcal{S}$ be a typed store such that $s : \mathcal{S} \models E : \mathcal{E}$. Provided that $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$ and $s, E \vdash \mathtt{e} \rightarrow v, f, s'$, there exists a store model $\mathcal{S}'$ such that $s : \mathcal{S} \sqsubseteq s' : \mathcal{S}'$ with:*

$$\mathcal{S}' \models f : \sigma \textit{ and } s' : \mathcal{S}' \models v : \tau$$

Proof The proof is by induction on the length of the dynamic evaluation, for each syntactic category of expressions.

Non-expansive expressions in `let`-bindings require a particular treatment. Assume that $\neg\mathit{expansive}[\![\mathtt{e}]\!]$ and $s, E \vdash \mathtt{e} \rightarrow v, f, s$ holds. Then, $s, E_{\mathtt{x}} \cup \{\mathtt{x} \mapsto v\} \vdash \mathtt{e}' \rightarrow v', f', s'$ holds if and only if there exists a proof of $s, E \vdash \mathtt{e}'[\mathtt{e}/\mathtt{x}] \rightarrow v', f', s'$. Thus, without loss of generality, we consider that non-expansive expressions in `let` bindings are explicitly substituted in the body of `let` constructs.

Case of (var) The hypothesis are:

$$s : \mathcal{S} \models E : \mathcal{E} \textit{ and } s, E \vdash \mathtt{x} \rightarrow E(\mathtt{x}), \emptyset, s \textit{ and } \mathcal{E} \vdash \mathtt{x} : \mathcal{E}(\mathtt{x}), \emptyset$$

We must have $\mathtt{x} \in \mathit{Dom}(()E)$ and $\mathtt{x} \in \mathit{Dom}(()\mathcal{E})$. From $s : \mathcal{S} \models E : \mathcal{E}$ and by taking $\mathcal{S}' = \mathcal{S}$, we conclude:

$$\mathcal{S} \models \emptyset : \emptyset \textit{ and } s : \mathcal{S} \models E(\mathtt{x}) : \mathcal{E}(\mathtt{x})$$

Case of (abs) The hypothesis are:

$$\begin{array}{l} s : \mathcal{S} \models E : \mathcal{E} \\ \mathcal{E} \vdash (\texttt{lambda (x) e}) : \tau \xrightarrow{\sigma} \tau', \emptyset \\ s, E \vdash (\texttt{lambda (x) e}) \rightarrow \langle \texttt{x}, \texttt{e}, E_{\texttt{x}} \rangle, \emptyset, s \end{array}$$

By the definition of the relation $\mathit{gfp}(\mathcal{F})$, taking $\mathcal{S}' = \mathcal{S}$, it follows that:

$$\mathcal{S} \models \emptyset : \emptyset \ \mathit{and}\ s : \mathcal{S} \models \langle \texttt{x}, \texttt{e}, E_{\texttt{x}} \rangle : \tau \xrightarrow{\sigma} \tau'$$

Case of (rec) The hypothesis are:

$$\begin{array}{l} s : \mathcal{S} \models E : \mathcal{E} \\ s, E \vdash (\texttt{rec (f x) e}) \rightarrow c, \emptyset, s \\ \mathcal{E} \vdash (\texttt{rec (f x) e}) : \tau \xrightarrow{\sigma} \tau', \emptyset \end{array}$$

This requires that:

$$\mathcal{E}_{\texttt{f},\texttt{x}} \cup \{\texttt{f} \mapsto \tau \xrightarrow{\sigma} \tau'\} \cup \{\texttt{x} \mapsto \tau\} \vdash \texttt{e} : \tau', \sigma \ \mathit{and}\ c = \langle \texttt{x}, \texttt{e}, E_{\texttt{f},\texttt{x}} \cup \{\texttt{f} \mapsto c\} \rangle$$

Let $\mathcal{E}' = \mathcal{E}_{\texttt{f}} \cup \{\texttt{f} \mapsto \tau \xrightarrow{\sigma} \tau'\}$, then $\mathcal{E}'_{\texttt{x}} \cup \{\texttt{x} \mapsto \tau\} \vdash \texttt{e} : \tau', \sigma$. By definition of the rule *(abs)*, we have:

$$\mathcal{E}' \vdash (\texttt{lambda (x) e}) : \tau \xrightarrow{\sigma} \tau', \emptyset$$

Let $E' = E_{\texttt{f},\texttt{x}} \cup \{\texttt{f} \mapsto c\}$. If we take $\mathcal{S}' = \mathcal{S}$, proving that $s' : \mathcal{S}' \models c : \tau \xrightarrow{\sigma} \tau'$ is equivalent to showing that $(s, \mathcal{S}, c, \tau \xrightarrow{\sigma} \tau') \in \mathit{gfp}(\mathcal{F})$. To this end, we define

$$\mathcal{Q} = \mathit{gfp}(\mathcal{F}) \cup \{(s, \mathcal{S}, c, \tau \xrightarrow{\sigma} \tau')\}$$

and show that $\mathcal{Q}$ is $\mathcal{F}$-consistent.

So, take $q \in \mathcal{Q}$. If $q \in \mathit{gfp}(\mathcal{F})$ then, since $\mathit{gfp}(\mathcal{F}) \subseteq \mathcal{Q}$ and $\mathcal{F}$ is monotonic, $q \in \mathcal{F}(\mathcal{Q})$. Otherwise, $q = (s, \mathcal{S}, c, \tau \xrightarrow{\sigma} \tau')$. Since $\mathcal{E}' \vdash (\texttt{lambda (x) e}) : \tau \xrightarrow{\sigma} \tau', \emptyset$, and $(s, \mathcal{S}, E(\texttt{y}), \mathcal{E}(\texttt{y})) \in \mathcal{Q}$ for every $\texttt{y} \in \mathit{Dom}(()E)$, and $(s, \mathcal{S}, c, \tau \xrightarrow{\sigma} \tau') \in \mathcal{Q}$, we get:

$$\mathit{for\ every}\ \texttt{y} \in \mathit{Dom}(()E),\ (s, \mathcal{S}, E'(\texttt{y}), \mathcal{E}'(\texttt{y})) \in \mathcal{Q}$$

and have proved that $\mathcal{Q}$ is $\mathcal{F}$-consistent. As a result:

$$\mathcal{S} \models \emptyset : \emptyset \ \mathit{and}\ s : \mathcal{S} \models c : \tau \xrightarrow{\sigma} \tau'$$

Case of (app) The hypothesis are:

$$\begin{array}{l} s : \mathcal{S} \models E : \mathcal{E} \\ \mathcal{E} \vdash (\texttt{e e}') : \tau', \sigma_0 \\ s, E \vdash (\texttt{e e}') \rightarrow v', f \cup f' \cup f'', s' \end{array}$$

By the definition of rule *(app)*, there exist τ, σ, σ' and σ'' such that $\sigma_0 = \sigma \cup \sigma' \cup \sigma''$ with

$$\mathcal{E} \vdash \texttt{e} : \tau \xrightarrow{\sigma''} \tau', \sigma \ \mathit{and}\ \mathcal{E} \vdash \texttt{e}' : \tau, \sigma'$$

By definition of the rule *(app)* in the dynamic semantics, we have:

$$\begin{array}{l} s, E \vdash \mathtt{e} \to \langle \mathtt{x}, \mathtt{e}'', E' \rangle, f, s_1 \\ s_1, E \vdash \mathtt{e}' \to v, f', s_2 \\ s_2, E' \cup \{\mathtt{x} \mapsto v\} \vdash \mathtt{e}'' \to v', f'', s' \end{array}$$

By induction on $\mathtt{e}$, there exists a store model $\mathcal{S}_1$ such that $s : \mathcal{S} \sqsubseteq s_1 : \mathcal{S}_1$ verifying:

$$s_1 : \mathcal{S}_1 \models \langle \mathtt{x}, \mathtt{e}'', E' \rangle : \tau \stackrel{\sigma''}{\to} \tau' \textit{ and } \mathcal{S}_1 \models f : \sigma$$

By the side-effects lemma, this implies that $s_1 : \mathcal{S}_1 \models E : \mathcal{E}$. By induction on $\mathtt{e}'$, there exists a store model $\mathcal{S}_2$ such that $s_1 : \mathcal{S}_1 \sqsubseteq s_2 : \mathcal{S}_2$ verifying:

$$s_2 : \mathcal{S}_2 \models v : \tau \textit{ and } \mathcal{S}_2 \models f' : \sigma'$$

We have $s_2 : \mathcal{S}_2 \models \langle \mathtt{x}, \mathtt{e}'', E' \rangle : \tau \stackrel{\sigma''}{\to} \tau'$ by the side-effects lemma. By definition of the $\models$ relation, there exists a type environment $\mathcal{E}'$ such that $s_2 : \mathcal{S}_2 \models E' : \mathcal{E}'$. By the side-effects lemma:

$$s_2 : \mathcal{S}_2 \models E' \cup \{\mathtt{x} \mapsto v\} : \mathcal{E}' \cup \{\mathtt{x} \mapsto \tau\}$$

By induction hypothesis on $\mathtt{e}''$, there exists a model $\mathcal{S}'$ such that $s_2 : \mathcal{S}_2 \sqsubseteq s' : \mathcal{S}'$ which verifies the theorem. Thus,

$$\mathcal{S}' \models f'' : \sigma'' \textit{ and } s' : \mathcal{S}' \models v' : \tau'$$

By transitivity of $\sqsubseteq$, this allows us to conclude that $\mathcal{S}'$ verifies $s : \mathcal{S} \sqsubseteq s' : \mathcal{S}'$ with:

$$s' : \mathcal{S}' \models v' : \tau' \textit{ and } \mathcal{S}' \models f \cup f' \cup f'' : \sigma \cup \sigma' \cup \sigma''$$

Case of (new) The hypothesis are:

$$\begin{array}{l} s : \mathcal{S} \models E : \mathcal{E} \\ \mathcal{E} \vdash (\mathtt{new\ e}) : \mathit{ref}_\rho(\tau), \sigma \cup \mathit{init}(\rho) \\ s, E \vdash (\mathtt{new\ e}) \to l, f \cup \{\mathit{init}(l)\}, s' \cup \{l \mapsto v\} \end{array}$$

By definition of the semantics, this requires that:

$$s, E \vdash \mathtt{e} \to v, f, s' \textit{ and } \mathcal{E} \vdash \mathtt{e} : \tau, \sigma$$

By induction on $\mathtt{e}$, there exists a store model $\mathcal{S}_1$ such that $s : \mathcal{S} \sqsubseteq s' : \mathcal{S}_1$ verifying:

$$\mathcal{S}_1 \models f : \sigma \textit{ and } s' : \mathcal{S}_1 \models v : \tau$$

By definition, we have $\{l \mapsto (\rho, \tau)\} \models \{\mathit{init}(l)\} : \mathit{init}(\rho)$. Since $l \notin \mathit{Dom}(()s')$, we define $\mathcal{S}' = \mathcal{S}_1 \cup \{l \mapsto (\rho, \tau)\}$; we have:

$$s' : \mathcal{S}_1 \sqsubseteq s' \cup \{l \mapsto v\} : \mathcal{S}'$$

By transitivity of $\sqsubseteq$, we conclude that $s : \mathcal{S} \sqsubseteq s' \cup \{l \mapsto v\} : \mathcal{S}'$ with:

$$\mathcal{S}' \models f \cup \{\mathit{init}(l)\} : \sigma \cup \mathit{init}(\rho) \textit{ and } s' \cup \{l \mapsto v\} : \mathcal{S}' \models l : \mathit{ref}_\rho(\tau)$$

Case of (get) The hypothesis are:

$$\begin{array}{l} s:\mathcal{S} \models E:\mathcal{E} \\ \mathcal{E} \vdash (\texttt{get e}) : \tau, \sigma \cup \mathit{read}(\rho) \\ s, E \vdash (\texttt{get e}) \rightarrow s'(l), f \cup \{\mathit{read}(l)\}, s' \end{array}$$

This requires that $s, E \vdash \texttt{e} \rightarrow l, f, s'$ and $\mathcal{E} \vdash \texttt{e} : \mathit{ref}_\rho(\tau), \sigma$. By induction hypothesis on $\texttt{e}$, there exists $\mathcal{S}'$ such that $s : \mathcal{S} \sqsubseteq s' : \mathcal{S}'$ verifying:

$$\mathcal{S}' \models f : \sigma \textit{ and } s' : \mathcal{S}' \models l : \mathit{ref}_\rho(\tau)$$

By definition, $\{l \mapsto (\rho, \tau)\} \models \{\mathit{read}(l)\} : \mathit{read}(\rho)$. Since $\{l \mapsto (\rho, \tau)\} \subseteq \mathcal{S}'$, we conclude that:

$$\mathcal{S}' \models f \cup \{\mathit{read}(l)\} : \sigma \cup \mathit{read}(\rho) \textit{ and } s' : \mathcal{S}' \models s'(l) : \tau$$

Case of (set) The hypothesis are:

$$\begin{array}{l} s:\mathcal{S} \models E:\mathcal{E} \\ \mathcal{E} \vdash (\texttt{set e e}') : \mathit{unit}, \sigma \cup \sigma' \cup \mathit{write}(\rho) \\ s, E \vdash (\texttt{set e e}') \rightarrow u, f \cup f' \cup \{\mathit{write}(l)\}, s''_l \cup \{l \mapsto v\} \end{array}$$

In the dynamic semantics, this requires that:

$$s, E \vdash \texttt{e} \rightarrow l, f, s \textit{ and } s', E \vdash \texttt{e}' \rightarrow v, f', s''$$

In the static semantics, we must have:

$$\mathcal{E} \vdash \texttt{e} : \mathit{ref}_\rho(\tau), \sigma \textit{ and } \mathcal{E} \vdash \texttt{e}' : \tau, \sigma'$$

By induction hypothesis on $\texttt{e}$, there exists a model $\mathcal{S}_1$ such that $s : \mathcal{S} \sqsubseteq s' : \mathcal{S}_1$ verifying:

$$\mathcal{S}_1 \models f : \sigma \textit{ and } s' : \mathcal{S}_1 \models l : \mathit{ref}_\rho(\tau)$$

Similarly, there exists $\mathcal{S}'$ such that $s' : \mathcal{S}_1 \sqsubseteq s'' : \mathcal{S}'$ with:

$$\mathcal{S}' \models f' : \sigma' \textit{ and } s'' : \mathcal{S}' \models v : \tau$$

Since $\mathcal{S}_1 \subseteq \mathcal{S}'$, we have $\{l \mapsto (\rho, \tau)\} \subseteq \mathcal{S}'$. Thus:

$$\mathcal{S}' \models \{\mathit{write}(l)\} : \mathit{write}(\rho)$$

We conclude that $s : \mathcal{S} \sqsubseteq s''_l \cup \{l \mapsto v\} : \mathcal{S}'$ with:

$$\mathcal{S}' \models f \cup f' \cup \{\mathit{write}(l)\} : \sigma \cup \sigma' \cup \mathit{write}(\rho) \textit{ and } s''_l \cup \{l \mapsto v\} : \mathcal{S}' \models u : \mathit{unit}$$

Case of (ilet) The hypothesis are:

$$\begin{array}{l} s : \mathcal{S} \models E : \mathcal{E} \\ \mathit{expansive}[\![\mathtt{e}]\!] \\ \mathcal{E} \vdash (\mathtt{let\ (x\ e)\ e'}) : \tau', \sigma \cup \sigma' \\ s, E \vdash (\mathtt{let\ (x\ e)\ e'}) \rightarrow v', f', s' \end{array}$$

By definition of the dynamic semantics, we have:

$$s, E \vdash \mathtt{e} \rightarrow v, f, s_1 \ \mathit{and}\ s_1, E_{\mathtt{X}} \cup \{\mathtt{x} \mapsto v\} \vdash \mathtt{e'} \rightarrow v', f', s'$$

In the static semantics, we must have:

$$\mathcal{E} \vdash \mathtt{e} : \tau, \sigma \ \mathit{and}\ \mathcal{E}_{\mathtt{X}} \cup \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e'} : \tau', \sigma'$$

By induction on $\mathtt{e}$, there exists a store model $\mathcal{S}_1$ such that $s : \mathcal{S} \sqsubseteq s_1 : \mathcal{S}_1$ verifying:

$$\mathcal{S}_1 \models f : \sigma \ \mathit{and}\ s_1 : \mathcal{S}_1 \models v : \tau$$

Moreover $s_1 : \mathcal{S}_1 \models E : \mathcal{E}$ implies that $s_1 : \mathcal{S}_1 \models E_{\mathtt{X}} \cup \{\mathtt{x} \mapsto v\} : \mathcal{E}_{\mathtt{X}} \cup \{\mathtt{x} \mapsto \tau\}$. By induction hypothesis on $\mathtt{e'}$, there exists $\mathcal{S}'$ such that $s_1 : \mathcal{S}_1 \sqsubseteq s' : \mathcal{S}'$ verifying:

$$\mathcal{S}' \models f' : \sigma' \ \mathit{and}\ s' : \mathcal{S}' \models v' : \tau'$$

We conclude that $s : \mathcal{S} \sqsubseteq s' : \mathcal{S}'$ with:

$$\mathcal{S}' \models f \cup f' : \sigma \cup \sigma' \ \mathit{and}\ s' : \mathcal{S}' \models v' : \tau' \ \square$$

6 Type, Region and Effect Reconstruction

We now present the algorithm for reconstructing the types, regions and effects of expressions. We discuss the central ideas of our approach, describe the unification process, give the reconstruction algorithm and discuss its properties.

6.1 Presentation

Given a type environment and an expression, the reconstruction algorithm determines a type and an effect consistent with all type and effect assignments of the static semantics. The reconstructed solution, if one exists, satisfies the criteria of maximality of the type with respect to substitution on variables, and minimality of the effect with respect to the subsumption on effects.

We view the reconstruction of types and effects of expressions as a constraint satisfaction problem. The algorithm computes equalities between types and regions, and inequalities between effects. For an expression to admit a type and an effect in the static semantics, this set of inequations must have at least one solution.

An important invariant of our method is that latent effects of functions are always represented by effect variables in the algorithm. The algorithm only deals with region variables; region constants only appear in the static semantics. This makes the problem of solving equations tractable by a simple extension to a unification algorithm on free algebras [Robinson] used on types, region variables and effect variables.

$$
\begin{array}{rcl}
\theta \in \mathit{Subst} & = & (\mathit{TyVar} \xrightarrow{fin} \mathit{Type}) + (\mathit{RegVar} \xrightarrow{fin} \mathit{Region}) + (\mathit{EfVar} \xrightarrow{fin} \mathit{Effect}) \\
\kappa \in \mathit{Constraint} & = & \mathcal{P}_{fin}(\mathit{EfVar} \times \mathit{Effect}) \\
\forall \upsilon_{1..n}.(\tau, \kappa) \in \mathit{TyScheme} & & \\
\mathcal{E} \in \mathit{TyEnv} & = & \mathit{Id} \xrightarrow{fin} \mathit{TyScheme}
\end{array}
$$

Substitutions and Constraint Sets

Constraints κ consist of sets of inequalities between effect variables and effect sets. The inequality $\varsigma \sqsupseteq \sigma$ in κ enforces a lower bound σ for the inferred effect variable ς, consistent with the static semantics. It is built during the processing of `lambda` and `rec` expressions which is the place where effects are introduced into types. By construction, constraint sets *always* admit at least one solution (see below).

In order to avoid recomputing the type of non-expansive binding expressions in `let` constructs as would a naive implementation of the syntactic substitution in the (*let*) rule, we use algebraic type schemes to generically represent their types and associated constraints. *Algebraic type schemes* $\forall \upsilon_{1..n}.(\tau, \kappa)$ are composed of a type τ and a set of inequalities κ universally quantified over type, effect and region variables $\upsilon_{1..n}$. Algebraic type schemes are used to implement the textual substitution specified in the (*let*) binding rule for non-expansive expressions `e`. The type and constraint set associated with `e` only depend on the free variables of `e` and, thereby, on the type environment $\mathcal{E}$. An algebraic type scheme *caches* the effect constraint that would have to be recomputed each time `e` appeared in the substituted body. Constrained type environments $\mathcal{E}$ map value identifiers to algebraic type schemes.

Equations on types, effect variables and regions are solved by a Robinson-like unification algorithm [Robinson] operating on the free algebra of types handled by the reconstruction algorithm. It returns a substitution θ which is the most general unifier of two type terms. Substitutions θ are defined on variables and extended on types and environments in the obvious way. We note *Id* the identity substitution.

6.2 The reconstruction algorithm

Given a type environment $\mathcal{E}$ and an expression $\texttt{e}$, the reconstruction algorithm $\mathcal{I}$ computes a substitution θ ranging over the free type, effect and region variables of the type environment $\mathcal{E}$, a type τ, an effect σ and an inequality system κ containing the inequalities that need to be satisfied by effect variables in order to preserve the static semantics.

$$
\begin{array}{l}
\mathcal{I}(\mathcal{E}, \texttt{x}) \Rightarrow \\
\quad \textit{if}\ \texttt{x} \mapsto \forall \upsilon_{1..n}.(\tau, \kappa) \in \mathcal{E}\ \textit{then} \\
\quad\quad \textit{let}\ \{\upsilon'_{1..n}\}\ \textit{new} \\
\quad\quad\quad \theta = \cup_{i=1}^{n} \{\upsilon_i \mapsto \upsilon'_i\} \\
\quad\quad \textit{in}\ \langle \mathit{Id}, \theta\tau, \emptyset, \theta\kappa \rangle \\
\quad \textit{else fail}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{I}(\mathcal{E}, (\texttt{let (x e) e}')) \Rightarrow \\
\quad \textit{let}\ \langle \theta, \tau, \sigma, \kappa \rangle = \mathcal{I}(\mathcal{E}, \texttt{e})\ \textit{in} \\
\quad \textit{if}\ \neg \textit{expansive}[\![\texttt{e}]\!]\ \textit{then} \\
\quad\quad \textit{let}\ \upsilon_{1..n} = (\mathit{fv}(\tau) \cup \mathit{fv}(\kappa)) \setminus \mathit{fv}(\mathcal{E}) \\
\quad\quad\quad \mathcal{E}' = \theta\mathcal{E}_{\texttt{x}} \cup \{\texttt{x} \mapsto \forall \upsilon_{1..n}.(\tau, \kappa)\} \\
\quad\quad\quad \langle \theta', \tau', \sigma', \kappa' \rangle = \mathcal{I}(\mathcal{E}', \texttt{e}') \\
\quad\quad \textit{in}\ \langle \theta'\theta, \tau', \sigma', \kappa' \rangle \\
\quad \textit{else let}\ \mathcal{E}' = \theta\mathcal{E}_{\texttt{x}} \cup \{\texttt{x} \mapsto \tau\} \\
\quad\quad\quad\quad \langle \theta', \tau', \sigma', \kappa' \rangle = \mathcal{I}(\mathcal{E}', \texttt{e}') \\
\quad\quad\quad \textit{in}\ \langle \theta'\theta, \tau', \theta'\sigma \cup \sigma', \theta'\kappa \cup \kappa' \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{I}(\mathcal{E}, (\texttt{lambda (x) e})) \Rightarrow \\
\quad \textit{let}\ \alpha, \varsigma\ \textit{new} \\
\quad\quad \langle \theta, \tau, \sigma, \kappa \rangle = \mathcal{I}(\mathcal{E}_{\texttt{x}} \cup \{\texttt{x} \mapsto \alpha\}, \texttt{e}) \\
\quad \textit{in}\ \langle \theta, \theta\alpha \xrightarrow{\varsigma} \tau, \emptyset, \kappa \cup \{\varsigma \sqsupseteq \sigma\} \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{I}(\mathcal{E}, (\texttt{rec (f x) e})) \Rightarrow \\
\quad \textit{let}\ \alpha, \alpha', \varsigma\ \textit{new} \\
\quad\quad \mathcal{E}' = \mathcal{E}_{\texttt{f},\texttt{x}} \cup \{\texttt{f} \mapsto \alpha \xrightarrow{\varsigma} \alpha'\} \cup \{\texttt{x} \mapsto \alpha\} \\
\quad\quad \langle \theta, \tau, \sigma, \kappa \rangle = \mathcal{I}(\mathcal{E}', \texttt{e}) \\
\quad\quad \theta' = \mathcal{U}(\theta\alpha', \tau) \\
\quad \textit{in}\ \langle \theta'\theta, \theta'\theta(\alpha \xrightarrow{\varsigma} \alpha'), \emptyset, \theta'(\kappa \cup \{\theta\varsigma \sqsupseteq \sigma\}) \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{I}(\mathcal{E}, (\texttt{e e}')) \Rightarrow \\
\quad \textit{let}\ \langle \theta, \tau, \sigma, \kappa \rangle = \mathcal{I}(\mathcal{E}, \texttt{e}) \\
\quad\quad \langle \theta', \tau', \sigma', \kappa' \rangle = \mathcal{I}(\theta\mathcal{E}, \texttt{e}') \\
\quad\quad \alpha, \varsigma\ \textit{new} \\
\quad\quad \theta'' = \mathcal{U}(\theta'\tau, \tau' \xrightarrow{\varsigma} \alpha) \\
\quad\quad \sigma'' = \theta''(\theta'\sigma \cup \sigma' \cup \varsigma) \\
\quad \textit{in}\ \langle \theta''\theta'\theta, \theta''\alpha, \sigma'', \theta''(\theta'\kappa \cup \kappa') \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{I}(\mathcal{E}, (\texttt{new e})) \Rightarrow \\
\quad \textit{let}\ \gamma\ \textit{new} \\
\quad\quad \langle \theta, \tau, \sigma, \kappa \rangle = \mathcal{I}(\mathcal{E}, \texttt{e}) \\
\quad \textit{in}\ \langle \theta, \mathit{ref}_{\gamma}(\tau), \sigma \cup \mathit{init}(\gamma), \kappa \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{I}(\mathcal{E}, (\texttt{get e})) \Rightarrow \\
\quad \textit{let}\ \langle \theta, \tau, \sigma, \kappa \rangle = \mathcal{I}(\mathcal{E}, \texttt{e}) \\
\quad\quad \alpha, \gamma\ \textit{new} \\
\quad\quad \theta' = \mathcal{U}(\mathit{ref}_{\gamma}(\alpha), \tau) \\
\quad \textit{in}\ \langle \theta'\theta, \theta'\alpha, \sigma \cup \mathit{read}(\theta'\gamma), \theta'\kappa \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{I}(\mathcal{E}, (\texttt{set e e}')) \Rightarrow \\
\quad \textit{let}\ \langle \theta, \tau, \sigma, \kappa \rangle = \mathcal{I}(\mathcal{E}, \texttt{e}) \\
\quad\quad \langle \theta', \tau', \sigma', \kappa' \rangle = \mathcal{I}(\theta\mathcal{E}, \texttt{e}') \\
\quad\quad \gamma\ \textit{new} \\
\quad\quad \theta'' = \mathcal{U}(\mathit{ref}_{\gamma}(\tau'), \theta'\tau) \\
\quad\quad \sigma'' = \theta''(\theta'\sigma \cup \sigma' \cup \mathit{write}(\gamma)) \\
\quad \textit{in}\ \langle \theta''\theta'\theta, \mathit{unit}, \sigma'', \theta''(\theta'\kappa \cup \kappa') \rangle
\end{array}
$$

Reconstruction Algorithm

Note that a consequence of the unification of effect variables (induced by type unification) ς and ς' is that, in the constraint set, the inequalities $\{\varsigma \sqsupseteq \sigma,\ \varsigma' \sqsupseteq \sigma'\}$ are replaced by $\{\varsigma \sqsupseteq \sigma, \varsigma \sqsupseteq \sigma'\}$, which is equivalent to $\{\varsigma \sqsupseteq \sigma \cup \sigma'\}$.

6.3 Unification

The algorithm $\mathcal{U}$ below solves the equations on types, region and effect variables that are built by the reconstruction algorithm. It returns a substitution θ as the most general unifier of two terms, or fails. Note that the reconstruction algorithm only needs to unify region and effect expressions that are variables.

Lemma 3 (Correctness of $\mathcal{U}$ [Robinson]) *Let τ and τ' be two type terms in the domain of $\mathcal{U}$. If $\mathcal{U}(\tau, \tau') \rightarrow \theta$, then $\theta\tau = \theta\tau'$ and, whenever $\theta'\tau = \theta'\tau'$, there exists a substitution θ'' such that $\theta' = \theta''\theta$.*

Proof $\mathcal{U}$ unifies terms over a free algebra, and is thus complete following [Robinson] $\Box$

$$
\begin{array}{rcl}
\mathcal{U}(\tau, \tau') & = & \textit{case } (\tau, \tau') \textit{ of} \\
(\textit{unit}, \textit{unit}) & \Rightarrow & \textit{Id} \\
(\alpha, \alpha') & \Rightarrow & \{\alpha \mapsto \alpha'\} \\
(\alpha, \tau) | (\tau, \alpha) & \Rightarrow & \textit{if } \alpha \in \textit{fv}(\tau) \textit{ then fail else } \{\alpha \mapsto \tau\} \\
(\tau_i \xrightarrow{\varsigma} \tau_f, \tau_i' \xrightarrow{\varsigma'} \tau_f') & \Rightarrow & \textit{let } \theta = \{\varsigma \mapsto \varsigma'\} \textit{ and } \theta' = \mathcal{U}(\theta\tau_i, \theta\tau_i') \textit{ in } \mathcal{U}(\theta'\theta\tau_f, \theta'\theta\tau_f')\theta'\theta \\
(\textit{ref}_\gamma(\tau), \textit{ref}_{\gamma'}(\tau')) & \Rightarrow & \textit{let } \theta = \{\gamma \mapsto \gamma'\} \textit{ in } \mathcal{U}(\theta\tau, \theta\tau')\theta \\
(\_, \_) & \Rightarrow & \textit{fail}
\end{array}
$$

Unification Algorithm

6.4 Constraint Satisfaction

An expression e is type and effect safe if and only if $\mathcal{I}$ applied to e does not fail and returns a constraint set κ that admits at least one solution.

Definition 3 (Effect Model) *A substitution μ from EfVar to Effect is a* model *of a constraint set κ, noted $\mu \models \kappa$, if and only if, for each inequality $\varsigma \sqsupseteq \sigma \in \kappa$, $\mu\varsigma \sqsupseteq \mu\sigma$.*

Theorem 2 (Satisfaction) *Every constraint set κ admits at least one model.*

Proof Let $\kappa_n = \{\varsigma_i \sqsupseteq \sigma_i,\ i = 1..n\}$ be a constraint system and consider, for all i, $\sigma_i' = \cup_{i=1}^n \sigma_i \setminus \cup_{i=1}^n \varsigma_i$. Then $\{\varsigma_i \mapsto \sigma_i'\}$ is a model of κ_n $\Box$

An important result is that the constraint systems of the reconstruction algorithm always admit a unique minimal model with respect to the subsumption relation $\sqsupseteq$ on effects. The relation $\sqsupseteq$ is straightforwardly extended by extension to models.

Theorem 3 (Minimality) *Any constraint set κ admits a unique minimal model $Min(\kappa)$ such that, for any model μ of κ, we have $\mu \sqsupseteq Min(\kappa)$.*

We assume here that the effect variables on the left hand sides of the inequations are distinct, following upon our remark in the section 6.2.

$$\textit{Min}(\emptyset) \Rightarrow \textit{Id and } \textit{Min}(\{\varsigma \sqsupseteq \sigma\} \cup \kappa') \Rightarrow \textit{let } \mu = \textit{Min}(\kappa') \textit{ in } \{\varsigma \mapsto \mu\sigma \setminus \varsigma\}\mu$$

The algorithm *Min* recursively computes the minimal model of κ by composing the model μ of the constraint subset κ' with the substitution of ς. Note that the solution is independent of the order with which constraints are selected.

Proof The proof is by induction on κ $\square$

7 Correctness of the Reconstruction Algorithm

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

Proof The proof is straightforward by induction on the structure of expressions $\square$

Theorem 4 (Termination) *On all inputs $(\mathcal{E}, \mathtt{e})$, the algorithm $\mathcal{I}$ either fails or terminates.*

Proof $\mathcal{I}$ works by induction on the structure of expressions of finite height $\square$

Algebraic type schemes are used to implement the textual substitution specified in the (*let*) binding rule for non-expansive expressions $\mathtt{e}$. Without loss of generality, we assume in the correctness proofs that, in programs to be typechecked, non-expansive let-bound expressions are explicitly substituted in the body; type environments thus simply map identifiers to types.

Theorem 5 (Soundness) *Let $\mathcal{E}$ be the reconstruction environment and $\mathtt{e}$ an expression. If $\mathcal{I}(\mathcal{E}, \mathtt{e}) = \langle \theta, \tau, \sigma, \kappa \rangle$ and $\mu \models \kappa$ for some model μ, then $\mu\theta\mathcal{E} \vdash \mathtt{e} : \mu\tau, \mu\sigma$.*

The soundness result states that the application of any model of the reconstructed inequality system to the reconstructed type and effect is a solution of the static semantics.

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

Case of (var) In the case of identifiers, note that whenever $\mathcal{I}(\mathcal{E}, \mathtt{x}) = \langle Id, \tau, \emptyset, \emptyset \rangle$ then $\mathtt{x} \mapsto \tau \in \mathcal{E}$. By definition of the rule (*var*), we have:

$$\mathcal{E} \vdash \mathtt{x} : \tau, \emptyset$$

Case of (abs) By hypothesis, we have $\mathcal{I}(\mathcal{E}, (\mathtt{lambda\ (x)\ e})) = \langle \theta, \theta\alpha \xrightarrow{\varsigma} \tau, \emptyset, \kappa \cup \{\varsigma \sqsupseteq \sigma\} \rangle$ and consider any model μ of $\kappa \cup \{\varsigma \sqsupseteq \sigma\}$.

By definition of the algorithm, we have $\mathcal{I}(\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \alpha\}, \mathtt{e}) = \langle \theta, \tau, \sigma, \kappa \rangle$. Moreover, μ is a model of κ, so that, by induction hypothesis on $\mathtt{e}$, we have:

$$\mu\theta(\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \alpha\}) \vdash \mathtt{e} : \mu\tau, \mu\sigma$$

Since μ models $\{\varsigma \sqsupseteq \sigma\}$, we have $\mu\varsigma \sqsupseteq \mu\sigma$ by definition. By the rule *(does)*, this requires that $\mu\theta(\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \alpha\}) \vdash \mathtt{e} : \mu\tau, \mu\varsigma$. By definition of the rule (*abs*), we can conclude that:

$$\mu\theta\mathcal{E} \vdash (\mathtt{lambda\ (x)\ e}) : \mu(\theta\alpha \xrightarrow{\varsigma} \tau), \emptyset$$

Case of (rec) The assumption is that:

$$\mathcal{I}(\mathcal{E}, (\texttt{rec (f x) e})) = \langle \theta'\theta, \theta'\theta(\alpha \xrightarrow{\varsigma} \alpha'), \emptyset, \theta'(\kappa \cup \{\theta\varsigma \sqsupseteq \sigma\})\rangle$$

Let us consider any model μ of $\theta'(\kappa \cup \{\theta\varsigma \sqsupseteq \sigma\})$. By definition of our algorithm, we have:

$$\theta' = \mathcal{U}(\theta\alpha', \tau) \textit{ and } \mathcal{I}(\mathcal{E}_{\texttt{f},\texttt{x}} \cup \{\texttt{f} \mapsto \alpha \xrightarrow{\varsigma} \alpha'\} \cup \{\texttt{x} \mapsto \alpha\}, \texttt{e}) = \langle \theta, \tau, \sigma, \kappa\rangle$$

Note also that $\mu\theta'$ is a model of κ, so that by induction hypothesis on $\texttt{e}$, we get:

$$\mu\theta'\theta(\mathcal{E}_{\texttt{f},\texttt{x}} \cup \{\texttt{f} \mapsto \alpha \xrightarrow{\varsigma} \alpha'\} \cup \{\texttt{x} \mapsto \alpha\}) \vdash \texttt{e} : \mu\theta'\tau, \mu\theta'\sigma$$

Since $\mu\theta'$ models $\{\theta\varsigma \sqsupseteq \sigma\}$, we have $\mu\theta'\theta\varsigma \sqsupseteq \mu\theta'\sigma$ by definition. By the rule *(does)*, this requires that:

$$\mu\theta'\theta(\mathcal{E}_{\texttt{f},\texttt{x}} \cup \{\texttt{f} \mapsto \alpha \xrightarrow{\varsigma} \alpha'\} \cup \{\texttt{x} \mapsto \alpha\}) \vdash \texttt{e} : \mu\theta'\tau, \mu\theta'\theta\varsigma$$

By unification, $\mu\theta'\tau = \mu\theta'\theta\alpha'$. By the definition of the rule *(rec)*, we get:

$$\mu\theta'\theta\mathcal{E} \vdash (\texttt{rec (f x) e}) : \mu\theta'\theta(\alpha \xrightarrow{\varsigma} \alpha'), \emptyset$$

Case of (app) In the case of the application construct, we assume that

$$\mathcal{I}(\mathcal{E}, (\texttt{e e}')) = \langle \theta''\theta'\theta, \theta''\alpha, \theta''(\theta'\sigma \cup \sigma' \cup \varsigma), \theta''(\theta'\kappa \cup \kappa')\rangle$$

We suppose that μ is a model of $\theta''(\theta'\kappa \cup \kappa')$. By the definition of our algorithm, we must have $\theta'' = \mathcal{U}(\theta'\tau, \tau' \xrightarrow{\varsigma} \alpha)$ for fresh variables α and ς, and also:

$$\mathcal{I}(\mathcal{E}, \texttt{e}) = \langle \theta, \tau, \sigma, \kappa\rangle \textit{ and } \mathcal{I}(\theta\mathcal{E}, \texttt{e}') = \langle \theta', \tau', \sigma'\kappa'\rangle$$

Since μ is a model of $\theta''(\theta'\kappa \cup \kappa')$, we have also $\mu\theta''\theta' \models \kappa$ and $\mu\theta'' \models \kappa'$, so that by induction hypothesis, we get:

$$\mu\theta''\theta'\theta\mathcal{E} \vdash \texttt{e} : \mu\theta''\theta'\tau, \mu\theta''\theta'\sigma \textit{ and } \mu\theta''\theta'\theta\mathcal{E} \vdash \texttt{e}' : \mu\theta''\tau', \mu\theta''\sigma'$$

By unification, we have $\mu\theta''\theta'\tau = \mu\theta''(\tau' \xrightarrow{\varsigma} \alpha)$. By the definition of the rule (*app*), we conclude:

$$\mu\theta''\theta'\theta\mathcal{E} \vdash (\texttt{e e}') : \mu\theta''\alpha, \mu\theta''(\theta'\sigma \cup \sigma' \cup \varsigma)$$

Case of (ilet) We assume that $\mathcal{I}(\mathcal{E}, (\texttt{let (x e) e}')) = \langle \theta'\theta, \tau', \theta'\sigma \cup \sigma', \theta'\kappa \cup \kappa'\rangle$ and suppose that μ is a model of $\theta'\kappa \cup \kappa'$. By definition of the algorithm $\mathcal{I}$, we have:

$$\mathcal{I}(\mathcal{E}, \texttt{e}) = \langle \theta, \tau, \sigma, \kappa\rangle \textit{ and } \mathcal{I}(\theta\mathcal{E}_{\texttt{x}} \cup \{\texttt{x} \mapsto \tau\}, \texttt{e}') = \langle \theta', \tau', \sigma', \kappa'\rangle$$

Since μ is a model of $\theta'\kappa \cup \kappa'$, we have $\mu\theta' \models \kappa$, so that by induction hypothesis on $\texttt{e}$, we get:

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

Now, since we also have $\mu \models \kappa'$, we get by induction hypothesis on $\texttt{e}'$:

$$\mu\theta'(\theta\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \tau\}) \vdash \mathtt{e'} : \mu\tau', \mu\sigma'$$

By the definition of rule (*ilet*), we conclude that:

$$\mu\theta'\theta\mathcal{E} \vdash (\mathtt{let\ (x\ e)\ e'}) : \mu\tau', \mu(\theta'\sigma \cup \sigma')$$

Case of (new) We suppose that $I(\mathcal{E}, (\mathtt{new\ e})) = \langle \theta, \mathit{ref}_\gamma(\tau), \sigma \cup \mathit{init}(\gamma), \kappa\rangle$ and that $\mu \models \kappa$. We must have $I(\mathcal{E}, \mathtt{e}) = \langle \theta, \tau, \sigma, \kappa\rangle$. By induction hypothesis on $\mathtt{e}$, we get:

$$\mu\theta\mathcal{E} \vdash \mathtt{e} : \mu\tau, \mu\sigma$$

By the definition of the rule (*new*), we conclude that:

$$\mu\theta\mathcal{E} \vdash (\mathtt{new\ e}) : \mu(\mathit{ref}_\gamma(\tau)), \mu(\sigma \cup \mathit{init}(\gamma))$$

Case of (get) We suppose that $I(\mathcal{E}, (\mathtt{get\ e})) = \langle \theta'\theta, \theta'\alpha, \sigma \cup \mathit{read}(\theta'\gamma), \theta'\kappa\rangle$ and that $\mu \models \theta'\kappa$. For some α, we must have:

$$I(\mathcal{E}, \mathtt{e}) = \langle \theta, \tau, \sigma, \kappa\rangle \textit{ and } \theta' = \mathcal{U}(\tau, \mathit{ref}_\gamma(\alpha))$$

By induction hypothesis on $\mathtt{e}$, since $\mu\theta'$ is a model of κ, we get:

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

By the rule (*get*), and since $\theta'\tau = \theta'\mathit{ref}_\gamma(\alpha)$ by unification, we conclude that:

$$\mu\theta'\theta\mathcal{E} \vdash (\mathtt{get\ e}) : \mu\theta'\alpha, \mu\theta'(\sigma \cup \mathit{read}(\gamma))$$

Case of (set) Assume that $\mathcal{I}(\mathcal{E}, (\mathtt{set\ e\ e'})) = \langle \theta''\theta'\theta, \mathit{unit}, \theta''(\theta'\sigma \cup \sigma' \cup \mathit{write}(\gamma)), \theta''(\theta'\kappa \cup \kappa')\rangle$ and that μ is a model of $\theta''(\theta'\kappa \cup \kappa')$. By the definition of our algorithm, we must have:

$$\begin{array}{l} \theta'' = \mathcal{U}(\mathit{ref}_\gamma(\tau'), \theta'\tau) \\ \mathcal{I}(\mathcal{E}, \mathtt{e}) = \langle \theta, \tau, \sigma, \kappa\rangle \\ \mathcal{I}(\theta\mathcal{E}, \mathtt{e'}) = \langle \theta', \tau', \sigma', \kappa'\rangle \end{array}$$

Since $\mu\theta''\theta' \models \kappa$ and by induction hypothesis on $\mathtt{e}$, we have:

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

Since $\mu\theta'' \models \kappa'$ and by induction hypothesis on $\mathtt{e'}$, we get:

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

By unification, we have $\theta''\theta'\tau = \theta''\mathit{ref}_\gamma(\tau')$. So, by the rule *(set)*, we conclude that:

$$\mu\theta''\theta'\theta\mathcal{E} \vdash (\mathtt{set\ e\ e'}) : \mathit{unit}, \mu\theta''(\theta'\sigma \cup \sigma' \cup \mathit{write}(\gamma)) \ \Box$$

The completeness theorem states that the reconstructed type τ' and effect σ' are maximal, with respect to any inferred type τ and effect σ, for some substitution θ'' that verifies the computed constraints κ'.

Theorem 6 (Completeness) *If* $\theta\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$, *then* $\mathcal{I}(\mathcal{E}, \mathtt{e}) = \langle \theta', \tau', \sigma', \kappa' \rangle$ *and there exists a substitution* θ'' *modeling* κ' *such that:*

$$\theta\mathcal{E} = \theta''\theta'\mathcal{E} \text{ and } \tau = \theta''\tau' \text{ and } \sigma \sqsupseteq \theta''\sigma'$$

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

Case of (var) We assume that $\theta\mathcal{E} \vdash \mathtt{x} : \tau, \sigma$. By the definition of the rule *(var)*, this requires that $\theta\mathcal{E} \vdash \mathtt{x} : \tau, \emptyset$. As a consequence, there exists τ' such that $\tau = \theta\tau'$ and $\mathcal{E}(\mathtt{x}) = \tau'$. By definition of the algorithm:

$$\mathcal{I}(\mathcal{E}, \mathtt{x}) = \langle Id, \tau', \emptyset, \emptyset \rangle$$

The theorem is satisfied with $\theta'' = \theta$.

Case of (abs) Assume that $\theta\mathcal{E} \vdash (\mathtt{lambda\ (x)\ e}) : \tau \xrightarrow{\sigma} \tau'', \emptyset$. By the definition of the rule *(abs)*, we have:

$$\theta\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e} : \tau'', \sigma$$

This is equivalent to $(\theta \cup \{\alpha \mapsto \tau\})(\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \alpha\}) \vdash \mathtt{e} : \tau'', \sigma$ for some type variable α. By induction hypothesis on $\mathtt{e}$, we have:

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

and there exists a substitution θ''_1 modeling κ' and verifying:

$$(\theta \cup \{\alpha \mapsto \tau\})(\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \alpha\}) = \theta''_1\theta'(\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \alpha\}) \text{ and } \tau'' = \theta''_1\tau' \text{ and } \sigma \sqsupseteq \theta''_1\sigma'$$

By the definition of the algorithm, for some ς, we have:

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

Since ς is fresh in algorithm $\mathcal{I}$, the substitution:

$$\theta'' = \theta''_1 \cup \{\varsigma \mapsto \sigma\}$$

is a model of both κ' and $\{\varsigma \sqsupseteq \sigma'\}$. Thus, we can conclude that:

$$\theta\mathcal{E} = \theta''\theta'\mathcal{E} \text{ and } \tau \xrightarrow{\sigma} \tau'' = \theta''(\theta'\alpha \xrightarrow{\varsigma} \tau')$$

Case of (rec) We suppose that $\theta\mathcal{E} \vdash (\mathtt{rec\ (f\ x)\ e}) : \tau \xrightarrow{\sigma} \tau'', \emptyset$. By the rule *(rec)*, this requires that:

$$\theta(\mathcal{E}_{\mathtt{f,x}} \cup \{\mathtt{f} \mapsto \tau \xrightarrow{\sigma} \tau''\} \cup \{\mathtt{x} \mapsto \tau\}) \vdash \mathtt{e} : \tau'', \sigma$$

For fresh α, α' and ς, this can be rewritten as:

$$(\theta \cup \{\alpha \mapsto \tau\} \cup \{\alpha' \mapsto \tau''\} \cup \{\varsigma \mapsto \sigma\})(\mathcal{E}_{\mathtt{f,x}} \cup \{\mathtt{f} \mapsto \alpha \xrightarrow{\varsigma} \alpha'\} \cup \{\mathtt{x} \mapsto \alpha\}) \vdash \mathtt{e} : \tau'', \sigma$$

Now, let us note $\mathcal{E}' = \mathcal{E}_{\mathtt{f},\mathtt{x}} \cup \{\mathtt{f} \mapsto \alpha \xrightarrow{\varsigma} \alpha'\} \cup \{\mathtt{x} \mapsto \alpha\}$. By induction hypothesis on $\mathtt{e}$, we get:

$$\mathcal{I}(\mathcal{E}', \mathtt{e}) = \langle \theta_1', \tau', \sigma', \kappa' \rangle$$

and there exists a model θ_1'' of κ' such that:

$$(\theta \cup \{\alpha \mapsto \tau\} \cup \{\alpha' \mapsto \tau''\} \cup \{\varsigma \mapsto \sigma\})\mathcal{E}' = \theta_1''\theta_1'\mathcal{E}' \textit{ and } \tau'' = \theta_1''\tau' \textit{ and } \sigma \sqsupseteq \theta_1''\sigma'$$

By unification, since $\tau'' = \theta_1''\theta_1'\alpha' = \theta_1''\tau'$, there exists θ_2' such that $\theta_2' = \mathcal{U}(\theta_1'\alpha', \tau')$. Thus, by the definition of the algorithm $\mathcal{I}$, we get:

$$\mathcal{I}(\mathcal{E}, (\mathtt{rec\ (f\ x)\ e})) = \langle \theta_2'\theta_1', \theta_2'\theta_1'(\alpha \xrightarrow{\varsigma} \alpha'), \emptyset, \theta_2'(\kappa' \cup \{\theta_1'\varsigma \sqsupseteq \sigma\}) \rangle$$

Since unification is complete, there exists θ'' such that $\theta_1'' = \theta''\theta_2'$. Since $\sigma = \theta_1''\theta_1'\varsigma$ and $\sigma \sqsupseteq \theta_1''\sigma'$, then $\theta'' \models \theta_2'\{\theta_1'\varsigma \sqsubseteq \sigma'\}$. Moreover, since $\theta_1'' \models \kappa'$, then $\theta'' \models \theta_2'\kappa'$. We conclude that θ'' is a model of $\theta_2'(\kappa' \cup \{\theta_1'\varsigma \sqsupseteq \sigma'\})$ such that:

$$\theta\mathcal{E} = \theta''\theta_2'\theta_1'\mathcal{E} \textit{ and } \tau \xrightarrow{\sigma} \tau'' = \theta''\theta_2'\theta_1'(\alpha \xrightarrow{\varsigma} \alpha')$$

Case of (app) We assume that $\theta\mathcal{E} \vdash (\mathtt{e}_1\ \mathtt{e}_2) : \tau', \sigma'$. By definition of rule *(app)*, there exist σ, σ_1 and σ_2 such that $\sigma' = \sigma_1 \cup \sigma_2 \cup \sigma$ verifying:

$$\theta\mathcal{E} \vdash \mathtt{e}_1 : \tau \xrightarrow{\sigma} \tau', \sigma_1 \textit{ and } \theta\mathcal{E} \vdash \mathtt{e}_2 : \tau, \sigma_2$$

By induction hypothesis on $\mathtt{e}_1$, we have:

$$\mathcal{I}(\mathcal{E}, \mathtt{e}_1) = \langle \theta_1', \tau_1', \sigma_1', \kappa_1' \rangle$$

and there exists a substitution θ_1'' modeling κ_1' such that:

$$\theta\mathcal{E} = \theta_1''\theta_1'\mathcal{E} \textit{ and } \tau \xrightarrow{\sigma} \tau' = \theta_1''\tau_1' \textit{ and } \sigma_1 \sqsupseteq \theta_1''\sigma_1'$$

Since $\theta\mathcal{E} = \theta_1''\theta_1'\mathcal{E}$, then $\theta_1''\theta_1'\mathcal{E} \vdash \mathtt{e}_2 : \tau, \sigma_2$. So by induction hypothesis on $\mathtt{e}_2$, we have:

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

There exists a substitution θ_2'' modeling κ_2' such that:

$$\theta_1''\theta_1'\mathcal{E} = \theta_2''\theta_2'\theta_1'\mathcal{E} \textit{ and } \tau = \theta_2''\tau_2' \textit{ and } \sigma_2 \sqsupseteq \theta_2''\sigma_2'$$

First note that:

$$\theta\mathcal{E} = \theta_1''\theta_1'\mathcal{E} = \theta_2''\theta_2'\theta_1'\mathcal{E}$$

Take α and ς new. Let V be the set of the free variables of $\theta_2'\theta_1'\mathcal{E}$, τ_2', σ_2' and κ_2' and define θ_3'' as follows:

$$\theta_3''v = \begin{cases} \theta_2''v, & v \in V \\ \tau', & v = \alpha \\ \sigma, & v = \varsigma \\ \theta_1''v, & \textit{otherwise} \end{cases}$$

By this definition, we get:

$$\theta\mathcal{E} = \theta''_3\theta'_2\theta'_1\mathcal{E} \textit{ and } \tau \xrightarrow{\sigma} \tau' = \theta''_3(\tau'_2 \xrightarrow{\varsigma} \alpha) \textit{ and } \theta''_2\sigma'_2 = \theta''_3\sigma'_2$$

Now, for every v in τ'_1, σ'_1 and κ'_1, either v is in $fv(\theta'_1\mathcal{E})$ or v is new, by definition of $\mathcal{I}$. Then, for every such v in $fv(\theta'_1\mathcal{E})$, since $\theta''_3\theta'_2(\theta'_1\mathcal{E}) = \theta''_2\theta'_2(\theta'_1\mathcal{E}) = \theta''_1(\theta'_1\mathcal{E})$, we have:

$$\theta''_3\theta'_2 v = \theta''_2\theta'_2 v = \theta''_1 v$$

Otherwise, v is new, and thus $\theta'_2 v = v$, so that we have:

$$\theta''_3\theta'_2 v = \theta''_3 v = \theta''_1 v$$

We get:

$$\tau \xrightarrow{\sigma} \tau' = \theta''_3\theta'_2\tau'_1 \textit{ and } \theta''_1\sigma'_1 = \theta''_3\theta'_2\sigma'_1 \textit{ and } \theta''_3\theta'_2 \models \kappa'_1$$

It follows that:

$$\theta''_3 \models \theta'_2\kappa'_1 \cup \kappa'_2$$

Since $\theta''_3\theta'_2\tau'_1 = \theta''_3(\tau'_2 \xrightarrow{\varsigma} \alpha)$ and by the correctness of unification, there exists a substitution θ'_3 such that $\theta'_3 = \mathcal{U}(\theta'_2\tau'_1, \tau'_2 \xrightarrow{\varsigma} \alpha)$ verifying:

$$\theta'_3\theta'_2\tau'_1 = \theta'_3(\tau'_2 \xrightarrow{\varsigma} \alpha)$$

By the definition of the algorithm, we get:

$$\mathcal{I}(\mathcal{E}, (\texttt{e}_1\ \ \texttt{e}_2)) = \langle \theta'_3\theta'_2\theta'_1, \theta'_3\alpha, \theta'_3(\theta'_2\sigma'_1 \cup \sigma'_2 \cup \varsigma), \theta'_3(\theta'_2\kappa'_1 \cup \kappa'_2)\rangle$$

Now, since θ'_3 is the most general unifier of $\theta'_2\tau'_1$ and $\tau'_2 \xrightarrow{\varsigma} \alpha$, there exists a substitution θ'' such that

$$\theta''_3 = \theta''\theta'_3$$

We have proved that θ'' models $\theta'_3(\theta'_2\kappa'_1 \cup \kappa'_2)$ and verifies:

$$\theta\mathcal{E} = \theta''\theta'_3\theta'_2\theta'_1\mathcal{E} \textit{ and } \tau' = \theta''\theta'_3\alpha \textit{ and } \sigma' \sqsupseteq \theta''\theta'_3(\theta'_2\sigma'_1 \cup \sigma'_2 \cup \varsigma)$$

Case of (ilet) We assume that $\theta\mathcal{E} \vdash (\texttt{let (x e}_1\texttt{) e}_2) : \tau_2, \sigma$. By the rule *(let)*, this requires that there exist σ_1 and σ_2 such that $\sigma = \sigma_1 \cup \sigma_2$ verifying:

$$\theta\mathcal{E} \vdash \texttt{e}_1 : \tau_1, \sigma_1 \textit{ and } \theta\mathcal{E}_\texttt{x} \cup \{\texttt{x} \mapsto \tau_1\} \vdash \texttt{e}_2 : \tau_2, \sigma_2$$

By induction hypothesis on $\texttt{e}_1$, we have:

$$\mathcal{I}(\mathcal{E}, \texttt{e}_1) = \langle \theta'_1, \tau'_1, \sigma'_1, \kappa'_1\rangle$$

There exists a substitution θ''_1 modeling κ'_1 such that:

$$\theta\mathcal{E} = \theta''_1\theta'_1\mathcal{E} \textit{ and } \tau_1 = \theta''_1\tau'_1 \textit{ and } \sigma_1 \sqsupseteq \theta''_1\sigma'_1$$

We also have $\theta\mathcal{E}_\texttt{x} \cup \{\texttt{x} \mapsto \tau_1\} \vdash \texttt{e}_2 : \tau_2, \sigma_2$, which is equivalent to:

$$\theta_1''(\theta_1'\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \tau_1'\}) \vdash \mathtt{e}_2 : \tau_2, \sigma_2$$

By induction hypothesis on $\mathtt{e}_2$, this implies that:

$$\mathcal{I}(\theta_1'\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \tau_1'\}, \mathtt{e}_2) = \langle \theta_2', \tau_2', \sigma_2', \kappa_2' \rangle$$

and that there exists θ_2'' modeling κ_2' such that:

$$\theta_1''(\theta_1'\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \tau_1'\}) = \theta_2''\theta_2'(\theta_1'\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \tau_1'\}) \ \mathit{and}\ \tau_2 = \theta_2''\tau_2' \ \mathit{and}\ \sigma_2 \sqsupseteq \theta_2''\sigma_2'$$

By the definition of the algorithm, we get:

$$\mathcal{I}(\mathcal{E}, (\mathtt{let}\ (\mathtt{x}\ \mathtt{e}_1)\ \mathtt{e}_2)) = \langle \theta_2'\theta_1', \tau_2', \theta_2'\sigma_1' \cup \sigma_2', \theta_2'\kappa_1' \cup \kappa_2' \rangle$$

Note that:

$$\theta\mathcal{E} = \theta_1''\theta_1'\mathcal{E} = \theta_2''\theta_2'\theta_1'\mathcal{E}$$

As for application, let V be the set of the free variables of $\theta_2'\theta_1'\mathcal{E}$, τ_2', σ_2' and κ_2' and define θ'' as follows:

$$\theta''\upsilon = \begin{cases} \theta_2''\upsilon, & \upsilon \in V \\ \theta_1''\upsilon, & \mathit{otherwise} \end{cases}$$

Thus θ'' is a model of $\theta_2'\kappa_1' \cup \kappa_2'$, and as for application, it satisfies:

$$\theta\mathcal{E} = \theta''\theta_2'\theta_1'\mathcal{E} \ \mathit{and}\ \tau_2 = \theta''\tau_2' \ \mathit{and}\ \sigma \sqsupseteq \theta''(\theta_2'\sigma_1' \cup \sigma_2')$$

Case of (new) We suppose that $\theta\mathcal{E} \vdash (\mathtt{new}\ \mathtt{e}) : \mathit{ref}_\rho(\tau), \sigma \cup \mathit{init}(\rho)$. By the rule *(new)*, this requires that $\theta\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$. By induction hypothesis on $\mathtt{e}$, we have:

$$\mathcal{I}(\mathcal{E}, \mathtt{e}) = \langle \theta', \tau', \sigma', \kappa' \rangle$$

and there exists θ_1'' modeling κ' such that:

$$\theta\mathcal{E} = \theta_1''\theta'\mathcal{E} \ \mathit{and}\ \tau = \theta_1''\tau' \ \mathit{and}\ \sigma \sqsupseteq \theta_1''\sigma'$$

By the definition of the algorithm, we get for some new γ:

$$\mathcal{I}(\mathcal{E}, (\mathtt{new}\ \mathtt{e})) = \langle \theta', \mathit{ref}_\gamma(\tau'), \sigma' \cup \mathit{init}(\gamma), \kappa' \rangle$$

Considering $\theta'' = \theta_1'' \cup \{\gamma \mapsto \rho\}$, we can conclude that:

$$\theta\mathcal{E} = \theta''\theta'\mathcal{E} \ \mathit{and}\ \mathit{ref}_\rho(\tau) = \theta''\mathit{ref}_\gamma(\tau') \ \mathit{and}\ \sigma \cup \mathit{init}(\rho) \sqsupseteq \theta''(\sigma' \cup \mathit{init}(\gamma))$$

Case of (get) We suppose that $\theta\mathcal{E} \vdash (\texttt{get e}) : \tau, \sigma \cup \mathit{read}(\rho)$. By the rule *(get)*, this requires that $\theta\mathcal{E} \vdash \texttt{e} : \mathit{ref}_\rho(\tau), \sigma$. By induction hypothesis on $\texttt{e}$, we have:

$$\mathcal{I}(\mathcal{E}, \texttt{e}) = \langle \theta_1', \tau', \sigma', \kappa' \rangle$$

and there exists a substitution θ_1'' modeling κ' such that:

$$\theta\mathcal{E} = \theta_1''\theta_1'\mathcal{E} \textit{ and } \mathit{ref}_\rho(\tau) = \theta_1''\tau' \textit{ and } \sigma \sqsupseteq \theta_1''\sigma'$$

Let $\theta_2'' = \theta_1'' \cup \{\gamma \mapsto \rho\} \cup \{\alpha \mapsto \tau\}$ where γ and α are new. We have $\theta_2''(\mathit{ref}_\gamma(\alpha)) = \theta_2''\tau'$. Thus, $\mathit{ref}_\gamma(\alpha)$ and τ' unify. Let θ_2' be such that:

$$\theta_2' = \mathcal{U}(\mathit{ref}_\gamma(\alpha), \tau')$$

By completeness of $\mathcal{U}$, there exists θ'' such that $\theta_2'' = \theta''\theta_2'$. By the definition of the algorithm, we then get:

$$\mathcal{I}(\mathcal{E}, (\texttt{get e})) = \langle \theta_2'\theta_1', \mathit{ref}_\gamma(\tau'), \sigma' \cup \mathit{read}(\theta_2'\gamma), \theta_2'\kappa' \rangle$$

So that θ'', which models $\theta_2'\kappa'$, satisfies the theorem:

$$\theta\mathcal{E} = \theta''\theta_2'\theta_1'\mathcal{E} \textit{ and } \tau = \theta''\theta_2'\alpha \textit{ and } \sigma \cup \mathit{read}(\rho) \sqsupseteq \theta''(\sigma' \cup \mathit{read}(\theta_2'\gamma))$$

Case of (set) We suppose that $\theta\mathcal{E} \vdash (\texttt{set e e'}) : \mathit{unit}, \sigma \cup \sigma' \cup \mathit{write}(\rho)$. By the rule *(set)*, this requires that:

$$\theta\mathcal{E} \vdash \texttt{e} : \mathit{ref}_\rho(\tau), \sigma \textit{ and } \theta\mathcal{E} \vdash \texttt{e'} : \tau, \sigma'$$

By induction hypothesis on $\texttt{e}$, we have:

$$\mathcal{I}(\mathcal{E}, \texttt{e}) = \langle \theta_1', \tau_1', \sigma_1', \kappa_1', \rangle$$

and there exists θ_1'' modeling κ_1' such that:

$$\theta\mathcal{E} = \theta_1''\theta_1'\mathcal{E} \textit{ and } \mathit{ref}_\rho(\tau) = \theta_1''\tau_1' \textit{ and } \sigma \sqsupseteq \theta_1''\sigma_1'$$

Since $\theta\mathcal{E} = \theta_1''\theta_1'\mathcal{E}$ and $\theta\mathcal{E} \vdash \texttt{e'} : \tau, \sigma'$, we get:

$$\mathcal{I}(\theta_1'\mathcal{E}, \texttt{e'}) = \langle \theta_2', \tau_2', \sigma_2', \kappa_2' \rangle$$

By induction hypothesis on $\texttt{e'}$, and there exists θ_2'' modeling κ_2' such that:

$$\theta_1''\theta_1'\mathcal{E} = \theta_2''\theta_2'\theta_1'\mathcal{E} \textit{ and } \tau = \theta_2''\tau_2' \textit{ and } \sigma' \sqsupseteq \theta_2''\sigma_2'$$

Take γ new. Let V be the set of the free variables of $\theta_2'\theta_1'\mathcal{E}$, τ_2', σ_2' and κ_2' and define θ_3'' as follows:

$$\theta_3''v = \begin{cases} \theta_2''v, & v \in V \\ \rho, & v = \gamma \\ \theta_1''v, & \textit{otherwise} \end{cases}$$

As for application, there exists a substitution $\theta'_3 = \mathcal{U}(ref_\gamma(\tau'_2), \theta'_2\tau'_1)$. By definition of the algorithm, we get:

$$\mathcal{I}(\mathcal{E}, (\texttt{set e e'})) = \langle \theta'_3\theta'_2\theta'_1, unit, \theta'_3(\theta'_2\sigma'_1 \cup \sigma'_2 \cup write(\gamma)), \theta'_3(\theta'_2\kappa'_1 \cup \kappa'_2)\rangle$$

Since unification is complete, there exists θ'' such that $\theta''_3 = \theta''\theta'_3$ which models $\theta'_3(\theta'_2\kappa'_1 \cup \kappa'_2)$ and satisfies:

$$\theta\mathcal{E} = \theta''\theta'_3\theta'_2\theta'_1\mathcal{E} \ and \ \sigma \cup \sigma' \cup write(\rho) \sqsupseteq \theta''\theta'_3(\theta'_2\sigma'_1 \cup \sigma'_2 \cup write(\gamma)) \ \Box$$

8 Examples

We consider two examples that demonstrate the effectiveness of our algorithm to infer effects of programs as well as to interpret and use effect information to perform code optimizations. All of the additional language constructs we use in this section can be easily integrated in the framework defined in this paper.

Program Documentation

This first example illustrates the effectiveness of program documentation provided by the use of our system. The expression below creates an integer reference value `counter` and initializes it to the value `initial`. The counter is then used in the `gensym`-like closure returned by the expression.

```
(lambda (initial)
  (let (counter (new initial))
    (lambda (inc)
      (begin (set counter (+ (get counter) inc))
             (get counter)))))
```

In the algorithm, the identifier `counter` is assigned the type $ref_\gamma(integer)$. Then, the type and effect of the body of the returned `lambda` expression:

```
(begin (set counter (+ (get counter) inc)) (get counter))
```

are computed. We get *integer* as type and $read(\gamma) \cup write(\gamma)$ as effect. As a consequence, the whole expression is assigned the following type and related constraint set:

$$integer \xrightarrow{\varsigma} (integer \xrightarrow{\varsigma'} integer), \ \{\varsigma \sqsupseteq init(\gamma), \ \varsigma' \sqsupseteq read(\gamma) \cup write(\gamma)\}$$

In the static semantics, this corresponds to the type:

$$integer \xrightarrow{init(\gamma)} (integer \xrightarrow{read(\gamma) \cup write(\gamma)} integer)$$

Parallel Code Generation

The second example illustrates the use of our type and effect system to perform sophisticated code optimizations such as stack allocation and parallelization of global operations on vectors, which have recently been implemented into a prototype of the related FX compiler [Talpin II], generating *Lisp [*Lisp] code and targeted towards the Connection Machine architecture [Hillis].

Contrarily to other work related to the topic of *compile-time garbage collection* or *reference escape analysis* ([Hudak], [Hughes] and [Neirynck]), type and effect inference effectively deals with higher-order functions, reference values and imperative constructs. The use of other methods such as abstract interpretation or interprocedural analysis may give more precise information than regions, but they are generally limited to simpler languages.

Regions denote abstractions of sets of memory locations. Effects are expressed in terms of these regions and approximate the observational imperative behavior of the evaluation of expressions. Nonetheless, if these effects are related to values that are locally allocated, the effects do not need to be reported. This can be detected by looking at the typing environment and the free variables of every expression [Gifford]. If a region appears in some effect but not in the type of the free variables or the return type of the expression, then such an effect is not observable from the outside. Any data structure allocated in such a region can be safely stack allocated, thus avoiding a superfluous and costly heap allocation.

In the following program:

```
(let (v (identity 10))
  (let (f (lambda (x) (* a (+ b x))))
    (vector_map f v)))
```

`(identity 10)` initializes a vector to the integers of 1 to 10, which is then bound to `v`. We define an affine function `f` which is then mapped over every element of `v`. Provided that we give to `v` and `f` the following types:

$$\texttt{v} \;:\; \mathit{vector}_{\gamma}(\mathit{integer}) \;\mathit{and}\; \texttt{f} \;:\; \mathit{integer} \xrightarrow{\emptyset} \mathit{integer}$$

the type and effect of this program are:

$$\mathit{vector}_{\gamma'}(\mathit{int})\;,\;\; \mathit{init}(\gamma) \cup \mathit{read}(\gamma) \cup \mathit{init}(\gamma')$$

Note that the region γ, in which the vector `v` was allocated, is absent both from the context of the program and its value type. As a result, the vector `v` is isolated once the execution of this program terminates, and it can thus be stack allocated.

As far as parallel code generation is concerned, we can easily detect that the function `f` only handles basic data types (*integer*) and does not produce any side effect; its mapping on `v` can thus be performed in parallel:

```
(*let ((v (*with_vp_set (vp_set_of_size 10) (enumerate!!))))
  (labels ((f!! x!) (*!! (!! a) (+!! (!! b) x!)))
     (*with_vp_set (pvar_vp_set v)
       (f!! v))))
```

The *Lisp code that is generated for this example program can be analyzed as follows. The construct `*let` performs stack allocation of the vector `v` as a specific *Lisp data structure:

a *pvar*. Each element of `v` is distributed over the processing elements of the Connection Machine. We define a parallel version `f!!` of the function `f`; it is then applied to the pvar `v` to perform the parallel mapping of `f` on `v`.

9 Conclusion

We have presented a type, region and effect inference algorithm for an implicitly typed functional language extended with imperative constructs. We have shown that this algorithm is consistent with its static semantics. It computes the maximal type and effect of expressions with respect to substitution on variables and the minimal effect with respect to the rule of subsumption on effects.

A number of standard program optimizations can take advantage of the program properties that type and effect inference computes. Stack allocation and parallel code generation have been discussed in this paper. This framework provides the basis for sophisticated program verification and transformation techniques in the presence of side-effects and higher-order functions. In order to assess the practicality of our approach, our inference algorithm has been implemented into a prototype of the FX compiler targeted towards the Connection Machine architecture [Hillis] at the Ecole des Mines de Paris [Talpin II].

Instead of resorting to a syntactic criterion for managing `let` polymorphism, we are working on extending this framework to handle more gracefully type generalization by using type schemes in a way reminiscent of Standard ML [Talpin I]. 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.

References

[*Lisp] *Lisp Reference Manual. Thinking Machines Corporation, 1987.

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

[Cousot] Cousot, P., and Cousot, R. Abstract Interpretation, a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings of the 1977 ACM Conference on Principles of Programming Languages.* ACM, New-York, 1977.

[Deutsch] Deutsch A. On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher-Order Functional Specifications. In *Proceedings of the 1990 ACM Conference on Principles of Programming Languages.* ACM, New-York, 1990.

[Gifford] 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.

[Gordon] Gordon, M. C. J., and Milner, R. Edinburgh LCF. In *Lecture Notes in Computer Science*, vol. 78. Springer Verlag, 1979.

[Hammel] Hammel, R. T., and Gifford, D. K. FX-87 Performance Measurements: Dataflow Implementation. *MIT/LCS/TR-421*, MIT Laboratory for Computer Science, November 1988.

[Harper] Harper, R., Milner, R., and Tofte, M. The definition of Standard ML. *Edinburgh LFCS Report 88-62*, University of Edinburgh, 1988.

[Harrison] Harrison, W. L. The Interprocedural Analysis and Automatic Parallelization of Scheme Programs. In *Lisp and Symbolic Computation, an Internal Journal*, 2 (3). 1989.

[Hillis] Hillis, W. D. The Connection Machine. The MIT Press, Cambridge, 1985.

[Hudak] Hudak, P. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 ACM Conference on Programming Language Design and Implementation.* ACM, New-York, August 1986.

[Hughes] Hughes J. Backward Analysis of Functional Programs. In *Proceedings of the Workshop on Partial Evaluation and Mixed Computation.* North Holland, October 1987.

[Jouvelot] 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.

[Larus] Larus, J. R., and Hilfinger, P. N. Detecting conflicts between structure accesses. In *Proceedings of the 1988 ACM Conference on Programming Language Design and Implementation.* ACM, New-York, 1988.

[Leroy] 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.

[Lucassen] 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.

[Milner] Milner, R. A Theory for type polymorphism in programming. In *Journal of Computer and Systems Sciences*, Vol. 17, pages 348-375. 1978.

[Mitchell] Mitchell, J. C., and Harper, R. The Essence of ML. In *Proceedings of the 1988 ACM Conference on Principles of Programming Languages.* ACM, New-York, 1988.

[Morris] Morris, J. H. Lambda Calculus Models of Programming Languages. *MAC-TR-57.* Massachusetts Institute of Technology, 1968.

[Neirynck] Neirynck, A., Panangaden, P., and Demers, A. Effect analysis of higher order languages. In *International Journal of Parallel Programming*, Vol. 18, No. 119. 1989.

[Plotkin] Plotkin, G. A structural approach to operational semantics. *Technical report DAIMI-FN-19.* Aarhus University, 1981.

[Robinson] 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.

[Rosen] Rosen, B. Data Flow Analysis for Procedural Languages. In *Journal of the ACM*, Vol. 26(2), pages 322-344. ACM, New-York, April 1979.

[Scheme] Rees, J., and Clinger W., Editors. Fourth Report on the Algorithmic Language Scheme. September 1988.

[Sheldon] Sheldon, A. M., and Gifford, D. K. Static Dependent Types for First Class Modules. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming.* ACM, New-York, 1990.

[Talpin I] Talpin, J. P., and Jouvelot, P. The Type and Effect Discipline. *Research Report EMP-CRI-A206* (revised version). Ecole Nationale Supérieure des Mines de Paris, November 1991.

[Talpin II] Talpin, J. P., and Jouvelot, P. The FX/CM Compiler Backend, or Taming Massive Parallelism with an Effect System. *Research Report EMP-CRI-A208.* Ecole Nationale Supérieure des Mines de Paris, November 1991.

[Tofte] Tofte, M. Operational semantics and polymorphic type inference. PhD Thesis, University of Edinburgh, 1987.