In the *proceedings of the 1992 Conference on Logic in Computer Science*. IEEE Computer Society Press, 1992

The Type and Effect Discipline

Jean-Pierre Talpin Pierre Jouvelot

Centre de Recherche en Informatique
Ecole Nationale Supérieure des Mines de Paris
35 rue Saint Honoré, 77305 Fontainebleau Cedex, France
talpin@ensmp.fr jouvelot@ensmp.fr

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 variables that are not free in the type environment or in the observable effect.

1 Introduction

Type inference [12] 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 [20, 13, 8] and more recently [21].

Polymorphic effect systems [10, 11] allow 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 [7, 18].

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 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.

In this paper, section 2 presents the related work. Then, we describe the dynamic (section 3) and static (section 4) semantics of the language. We state that the static and dynamic semantics are consistent (section 5). The reconstruction algorithm is presented in section 6 and 7. Our algorithm is shown correct with respect to the static semantics (section 8). Before concluding, we give some examples (section 9) that show

that our approach surpasses previous techniques. The extended version of this paper [19] contains the detailed proofs which are skipped here for space reasons.

2 Related Work

Short of the ad-hoc techniques used in the first type inference systems, the imperative type discipline [20] 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 [1]. A different approach, suggested in [8], 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 on which side-effecting operations are performed.

All those approaches build conservative approximations of value types that may be accessible from the global store. Inferring and keeping track of the types of values in the store has for long been the subject of many investigations [3, 15, 21]. Effect inference allows us to approximate the store on regions and types and, as such, can be used to decide when to perform type generalization.

The FX system [5, 11] suggests a static semantics for polymorphic type and effect checking. In [7], 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 [18] an algorithm that computes the maximal type and the minimal effect of expressions, using subsumption on effects to overcome the previous problems. Contrarily to the integrated approach presented here, it has to resort to a much cruder syntatic criterium to perform type generalization.

In the *type and effect discipline*, we determine the principal type and the minimal observable effects of expressions. We use effect information to perform type generalization. Effect information alleviates the need to use syntactic criteria such as expansiveness [20, 18] to decide where type generalization can be performed. More specifically, by using effect information together with an observation criterium, 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 9) that our system improves over earlier type generalization policies for ML-like languages.

3 Dynamic Semantics

In the syntax *Exp* of our language, expressions $\mathtt{e}$ consist of identifiers $\mathtt{x}$ defined on *Id*, lexical bindings ($\mathtt{let}$ $(\mathtt{x}\ \mathtt{e})$ $\mathtt{e}'$), first-class functions ($\mathtt{lambda}$ $(\mathtt{x})$ $\mathtt{e}$) and applications ($\mathtt{e}$ $\mathtt{e}'$).

Computable values v are either the unit value u, reference values r or closures $\langle \mathtt{x}, \mathtt{e}, E\rangle$. Closures are composed of an identifier $\mathtt{x}$, the expression body $\mathtt{e}$ and an environment E. Environments bind identifiers to values. The initial environment binds the identifiers `new`, `get` and `set` to the imperative functions *new*, *get* and *set* that respectively initialize, dereference and assign reference values.

$$
\begin{aligned}
&r \in \mathit{Ref}\\
&s \in \mathit{Store} = \mathcal{P}(\mathit{Action})\\
&E \in \mathit{Env} = \mathit{Id} \stackrel{fin}{\rightarrow} \mathit{Value}\\
&c \in \mathit{Closure} = \mathit{Id} \times \mathit{Exp} \times \mathit{Env}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Value} = \{u\} + \mathit{Closure} + \{\mathit{new}, \mathit{get}, \mathit{set}\} + \mathit{Ref}\\
&\mathit{Action} = \mathit{init}(\mathit{Ref}, \mathit{Value}) + \mathit{read}(\mathit{Ref}) + \mathit{write}(\mathit{Ref})
\end{aligned}
$$

The dynamic semantics of the language is similar to Core-ML [14] extended with imperative constructs. It is presented, as in [20], by a set of rules that inductively defines the relation of evaluation $s, E \vdash \mathtt{e} \rightarrow v, s'$ on the structure of expressions. Given a store s and an environment E, we associate to each expression $\mathtt{e}$ a value v and a new store s'. We note $E_{\mathtt{x}}$ for E excluding the binding of $\mathtt{x}$.

$$s, E \cup \{\mathtt{x} \mapsto v\} \vdash \mathtt{x} \rightarrow v, s$$

$$s, E \vdash (\mathtt{lambda}\ (\mathtt{x})\ \mathtt{e}) \rightarrow \langle \mathtt{x}, \mathtt{e}, E_{\mathtt{x}}\rangle, s$$

$$\frac{\begin{array}{l} s, E \vdash \mathtt{e} \rightarrow v, s' \\ s', E_{\mathtt{x}} \cup \{\mathtt{x} \mapsto v\} \vdash \mathtt{e}' \rightarrow v', s'' \end{array}}{s, E \vdash (\mathtt{let}\ (\mathtt{x}\ \mathtt{e})\ \mathtt{e}') \rightarrow v', s''}$$

$$\frac{\begin{array}{l} s, E \vdash \mathtt{e} \rightarrow \langle \mathtt{x}, \mathtt{e}'', E'\rangle, s' \\ s', E \vdash \mathtt{e}' \rightarrow v', s'' \\ s'', E' \cup \{\mathtt{x} \mapsto v'\} \vdash \mathtt{e}'' \rightarrow v'', s''' \end{array}}{s, E \vdash (\mathtt{e}\ \mathtt{e}') \rightarrow v'', s'''}$$

The store s is represented as a set of memory allocation and side-effect events. It records the initialization of a location r to a value v by $\mathit{init}(r, v)$, and by $\mathit{read}(r)$ or $\mathit{write}(r)$ the side-effects performed on r during execution. We note $\mathit{Dom}(s) = \{r \mid \mathit{init}(r, v) \in s\}$ the domain of store s and s_r the store s excluding $\mathit{init}(r, v)$.

We say that a store s' extends s, noted $s \subseteq s'$, iff $Dom(s) \subseteq Dom(s')$.

$$\frac{s, E \vdash \mathtt{e} \rightarrow new, s' \quad s', E \vdash \mathtt{e}' \rightarrow v, s'' \quad r \notin Dom(s)}{s, E \vdash (\mathtt{e}\ \ \mathtt{e}') \rightarrow r, s'' \cup \{init(r, v)\}}$$

$$\frac{s, E \vdash \mathtt{e} \rightarrow get, s' \quad s', E \vdash \mathtt{e}' \rightarrow r, s'' \quad init(r, v) \in s''}{s, E \vdash (\mathtt{e}\ \ \mathtt{e}') \rightarrow v, s'' \cup \{read(r)\}}$$

$$\frac{s, E \vdash \mathtt{e} \rightarrow set, s' \quad s', E \vdash \mathtt{e}' \rightarrow r, s'' \quad s'', E \vdash \mathtt{e}'' \rightarrow v, s'''}{s, E \vdash ((\mathtt{e}\ \ \mathtt{e}')\ \ \mathtt{e}'') \rightarrow u, s'''_r \cup \{write(r), init(r, v)\}}$$

4 Static Semantics

The presentation of our static semantics requires the following formal definitions for regions, effects and types.

- The domain of regions ρ consists in 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 is 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 $init(\rho, \tau)$, $read(\rho)$ or $write(\rho)$ that approximate memory side-effects on their region argument ρ. $init(\rho, \tau)$ denotes the allocation and initialization of a mutable location in the region ρ with a value of type τ. 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.

  $$\sigma \ ::= \ \emptyset \mid \varsigma \mid init(\rho, \tau) \mid read(\rho) \mid write(\rho) \mid \sigma \cup \sigma$$

  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 $\sigma = \sigma' \cup \sigma''$.

  Every expression has an effect. It represents an abstraction of the store transformations the evaluation of this expression may perform at run time. Effects are specific to the region over which they range, except for allocation. Every allocation effect $init(\rho, \tau)$, for instance incurred when `new` is called, not only records the region ρ of the allocated and initialized reference, but also the type τ of the referenced value.

- The domain of types τ is composed of *finite terms* made 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.

  $$\tau \ ::= \ unit \mid \alpha \mid ref_\rho(\tau) \mid \tau \xrightarrow{\sigma} \tau'$$

- Type schemes $\forall \upsilon_{1..n}.\tau$ are defined as types τ quantified over a set of type, region and effect variables $\upsilon_{1..n}$. A type τ' is an instance of $\forall \upsilon_{1..n}.\tau$, noted $\tau' \prec \forall \upsilon_{1..n}.\tau$, if there exists a substitution θ defined over $\upsilon_{1..n}$ such that $\tau' = \theta\tau$. Substitutions θ map variables to types, regions and effects. We note $\theta\theta'$ the composition of the substitution θ with θ' and Id the identity.

Type environments $\mathcal{E}$ map identifiers to type schemes. The rules $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$ of the static semantics associate each expression $\mathtt{e}$ with its possible types τ and effects σ:

$$(var) : \frac{\tau \prec \mathcal{E}(\mathtt{x})}{\mathcal{E} \vdash \mathtt{x} : \tau, \emptyset}$$

$$(let) : \frac{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma \quad \mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto Gen(\sigma, \mathcal{E})(\tau)\} \vdash \mathtt{e}' : \tau', \sigma'}{\mathcal{E} \vdash (\mathtt{let}\ \ (\mathtt{x}\ \ \mathtt{e})\ \ \mathtt{e}') : \tau', \sigma \cup \sigma'}$$

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

$$(app) : \frac{\mathcal{E} \vdash \mathtt{e} : \tau \xrightarrow{\sigma} \tau', \sigma' \quad \mathcal{E} \vdash \mathtt{e}' : \tau, \sigma''}{\mathcal{E} \vdash (\mathtt{e}\ \ \mathtt{e}') : \tau', \sigma \cup \sigma' \cup \sigma''}$$

$$(sub) : \frac{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma \quad \sigma' \sqsupseteq Observe(\mathcal{E}, \tau)(\sigma)}{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma'}$$

A syntax directed inference system could be obtained by performing subsumption of effect in the conclusion of the *(abs)* rule and in applying the *Observe* function to the conclusion of the rules *(app)* and *(let)*.

Generalization

The generalization of a type τ is performed at `let` boundaries on some of the type, region and effect variables $v_{1..n}$ 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 observed effect σ. The first condition is standard in pure functional languages. As for the second, just as types are bound to identifiers in the environment, types are bound to regions in the reconstructed allocation effects. Thus, when these regions are observable from the context (in the environment $\mathcal{E}$ or the returned value of type τ), those types cannot be generalized.

$$\begin{array}{l} Gen(\sigma, \mathcal{E})(\tau) = \\ \quad let\ \{v_{1..n}\} = fv(\tau) \setminus (fv(\mathcal{E}) \cup fv(\sigma))\ in\ \forall v_{1..n}.\tau \end{array}$$

The function fv returns the set of free variables in type, effect and environment terms. We also define the function fr that returns the set of region constants and variables that occur in type and effect terms.

Observation

For type generalization, only side-effects that can affect the context of an expression are worth reporting; $Observe(\mathcal{E}, \tau)(\sigma)$ is the set of observable effects of σ. It consists in the side effects ς that occur free in τ or $\mathcal{E}$ and the side effects of the form $init(\rho, \tau')$, $read(\rho)$ and $write(\rho)$ such that ρ occur free in τ or $\mathcal{E}$:

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

The other effects of σ are non observable from the enclosing context. They do not have to be reported since they are only local; they are associated to references that are freshly created and not exported from the expression. This criterium is similar to the notion of effect masking presented in [5].

Initial Environment

In the static semantics, the initial environment $\overline{\mathcal{E}_0}$ binds appropriate type schemes to the three imperative constructs `new`, `get` and `set`.

$$\begin{array}{rcl} \overline{\mathcal{E}_0} = \{\texttt{set} & \mapsto & \forall \alpha\rho\varsigma\varsigma'.ref_\rho(\alpha) \xrightarrow{\varsigma} \alpha \xrightarrow{\varsigma' \cup write(\rho)} unit, \\ \texttt{get} & \mapsto & \forall \alpha\rho\varsigma.ref_\rho(\alpha) \xrightarrow{\varsigma \cup read(\rho)} \alpha, \\ \texttt{new} & \mapsto & \forall \alpha\rho\varsigma.\alpha \xrightarrow{\varsigma \cup init(\rho,\alpha)} ref_\rho(\alpha)\} \end{array}$$

5 Consistency Theorem

In this section, we state that the static semantics is consistent with respect to the dynamic semantics. We first define store models that relate dynamic stores to static effects; every memory location is associated to a unique static region.

Definition 1 (Store Models) *A* store model $\mathcal{S} \in$ *StoreModel is a function that maps dynamic reference values to static regions. We say that* $\mathcal{S}'$ extends $\mathcal{S}$, *noted* $\mathcal{S} \subseteq \mathcal{S}'$, *when* $\mathcal{S}(r) = \mathcal{S}'(r)$ *for every* $r \in Dom(\mathcal{S})$.

We now define a recursive structural relation $\models$ that relates dynamic values to their types. Such a recursive definition actually defines a collection of relations. As shown in [20], this definition can then be regarded as a fixed point equation, the relation between values and types we are interested in being its greatest fixed point:

Definition 2 (Consistent Types) *The consistency relation* $s : \sigma, \mathcal{S} \models v : \tau$ *between values* v *and types* τ *with respect to the store model* $\mathcal{S}$, *the store* s *and the effect* σ *recursively satisfies:*

- $s : \sigma, \mathcal{S} \models u : unit$
- $s : \sigma, \mathcal{S} \models r : ref_{\mathcal{S}(r)}(\tau)$ *iff* $init(\mathcal{S}(r), \tau) \in \sigma$ *and there exists* v *such that* $init(r, v) \in s$ *and* $s : \sigma, \mathcal{S} \models v : \tau$
- $s : \sigma, \mathcal{S} \models \langle \mathtt{x}, \mathtt{e}, E \rangle : \tau$ *iff there exists* $\mathcal{E}$ *such that* $\mathcal{E} \vdash (\texttt{lambda (x) e}) : \tau, \emptyset$ *and* $s : \sigma, \mathcal{S} \models E : \mathcal{E}$

We note $s : \sigma, \mathcal{S} \models E : \mathcal{E}$ when $Dom(E) = Dom(\mathcal{E})$ and $s : \sigma, \mathcal{S} \models E(\mathtt{x}) : \tau$ for every $\mathtt{x} \in Dom(E)$ and type τ such that $\tau \prec \mathcal{E}(\mathtt{x})$.

Let $\mathcal{U}$ be the set of all tuples $(s, \sigma, \mathcal{S}, v, \tau)$ and consider the complete lattice $(\mathcal{P}(\mathcal{U}), \subseteq)$. The relation $s : \sigma, \mathcal{S} \models v : \tau$ between values v and types τ is the maximal fixed point $\overline{\mathcal{F}} = \cup_{\mathcal{Q} \subseteq \mathcal{U}} \{\mathcal{Q} \mid \mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})\}$ of the operator $\mathcal{F}$ defined according to the property defined above. We say that a set $\mathcal{Q}$ in $\mathcal{U}$ is $\mathcal{F}$*-consistent* if and only if $\mathcal{Q} \in \overline{\mathcal{F}}$.

As types abstract values, effects abstract memory stores. The counterpart of the previous relation between types and values thus relates effects and stores in a given store model:

Definition 3 (Consistent Effects) *The store* s *and the effect* σ *are consistent with respect to the store model* $\mathcal{S}$, *noted* $\mathcal{S} \models s : \sigma$, *iff:*

$$\begin{array}{l} \forall\ read(r) \in s,\ read(\mathcal{S}(r)) \in \sigma \\ \forall\ write(r) \in s,\ write(\mathcal{S}(r)) \in \sigma \\ \forall\ init(r, v) \in s\ \exists \tau,\ init(\mathcal{S}(r), \tau) \in \sigma,\ s : \sigma, \mathcal{S} \models v : \tau \end{array}$$

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 pair $s : \mathcal{S}$, is preserved when the store s is properly expanded.

Lemma 1 (Side Effects) *Suppose that $\mathcal{S} \models s : \sigma$ and $s : \sigma, \mathcal{S} \models v : \tau$. Whenever $\mathcal{S} \subseteq \mathcal{S}'$, $s \subseteq s'$, $\sigma' \sqsupseteq \sigma$ and $\mathcal{S}' \models s' : \sigma'$ then $s' : \sigma', \mathcal{S}' \models v : \tau$.*

The substitution lemma states that any proof $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$ in the static semantics is stable under substitution.

Lemma 2 (Substitution) *If $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$ then $\theta\mathcal{E} \vdash \mathtt{e} : \theta\tau, \theta\sigma$ for any substitution θ.*

Now, we present, with the two following lemmas, the properties of typing judgments, of the form $s : \sigma, \mathcal{S} \models v : \tau$, under substitution.

Lemma 3 (Semantics Substitution) *If $s : \sigma, \mathcal{S} \models v : \tau$ then $s : \theta\sigma, \theta\mathcal{S} \models v : \theta\tau$ for every substitution θ.*

The next lemma states that a judgment $s : \sigma, \mathcal{S} \models v : \tau$ is stable under substitutions that do not range on variables that are observable. We note $Observe(\tau)(\sigma)$ for $Observe(\{\}, \tau)(\sigma)$.

Lemma 4 (Observable Substitutions) *If $s : \sigma, \mathcal{S} \models v : \tau$ and if θ is a substitution such that $Dom(\theta) \subseteq fv(\tau) \setminus fv(Observe(\tau)(\sigma))$ then we have $s : \sigma, \mathcal{S} \models v : \theta\tau$*

The consistency theorem between the dynamic and static semantics ensures that an expression and the value it evaluates to have the same type. Similarly an expression and the store modifications it performs have the same observable effects.

Theorem 1 (Consistency) *Assume that $\mathcal{S} \models s : \sigma$ and $s : \sigma, \mathcal{S} \models E : \mathcal{E}$. If $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma'$ and $s, E \vdash \mathtt{e} \rightarrow v, s'$, then there exist a store model $\mathcal{S}'$ extending $\mathcal{S}$ and an unobservable effect σ'', i.e. satisfying $Observe(\mathcal{E}, \tau)(\sigma'') = \emptyset$, such that:*

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

Proof sketch The proof is by induction on the length of the dynamic evaluation and is constructive, in that it inductively determines σ''.

Let us outline the case for the `let` construct. We assume that $\mathcal{S} \models s : \sigma$ and $s : \sigma, \mathcal{S} \models E : \mathcal{E}$. The rules for *(let)* state that:

$$\begin{array}{l} s, E \vdash (\mathtt{let}\ (\mathtt{x}\ \mathtt{e})\ \mathtt{e}') \rightarrow v', s'' \\ \mathcal{E} \vdash (\mathtt{let}\ (\mathtt{x}\ \mathtt{e})\ \mathtt{e}') : \tau', \sigma' \end{array}$$

By the definition of the rule *(let)* in the dynamic semantics, we must have the following:

$$s, E \vdash \mathtt{e} \rightarrow v, s' \text{ and } s', E_{\mathtt{x}} \cup \{\mathtt{x} \mapsto v\} \vdash \mathtt{e}' \rightarrow v', s''$$

By the rule *(let)* in the static semantics, there must exist a type τ_1 and two effects σ'_1 and σ'_2, such that $\sigma'_1 \cup \sigma'_2 = \sigma'$ verifying:

$$\begin{array}{l} \mathcal{E} \vdash \mathtt{e} : \tau_1, \sigma'_1 \\ \mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto Gen(\sigma'_1, \mathcal{E})(\tau_1)\} \vdash \mathtt{e}' : \tau', \sigma'_2 \end{array}$$

Let us note $Gen(\sigma'_1, \mathcal{E})(\tau_1) = \forall v'_{1..n}.\tau_1$. By the definition of Gen, we know that:

$$v'_{1..n} = fv(\tau_1) \setminus (fv(\sigma'_1) \cup fv(\mathcal{E}))$$

To use the lemma on observable substitutions on the value of $\mathtt{e}$, we also want to ensure that none of the $v'_{1..n}$ are observable in the initial store effect σ in order to get:

$$fv(\tau_1) \cap fv(Observe(\mathcal{E}, \tau_1)(\sigma)) \subseteq fv(\tau_1) \cap (fv(\sigma'_1) \cup fv(\mathcal{E}))$$

To this end, we rename the variables $v'_{1..n}$ occuring in $Observe(\mathcal{E}, \tau_1)(\sigma)$ with the substitution $\theta = \{v'_i \mapsto v_i,\ i = 1..n\}$ where $v_i = v'_i$ for every $v'_i \notin fv(Observe(\mathcal{E}, \tau_1)(\sigma))$ and otherwise v_i is new and not in $fv(\sigma \cup \sigma'_1 \cup \sigma''_1) \cup fv(\mathcal{E})$. Let us note $\tau = \theta\tau_1$, by the substitution lemma, we get:

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

By induction hypothesis on $\mathtt{e}$, there exist $\mathcal{S}'$ extending $\mathcal{S}$ and an unobservable effect σ''_1 such that $Observe(\mathcal{E}, \tau)(\sigma''_1) = \emptyset$ verifying:

$$\mathcal{S}' \models s' : \sigma \cup \sigma'_1 \cup \sigma''_1 \text{ and } s' : \sigma \cup \sigma'_1 \cup \sigma''_1, \mathcal{S}' \models v : \tau$$

Since $fv(\tau) \cap fv(Observe(\mathcal{E}, \tau)(\sigma'_1))$ and $fv(\tau) \cap fv(Observe(\mathcal{E}, \tau)(\sigma))$ are subsets of $fv(\tau) \cap (fv(\sigma'_1) \cup fv(\mathcal{E}))$, for any substitution θ' defined on the $v_{1..n}$, we get:

$$Dom(\theta') \subseteq fv(\tau) \setminus fv(Observe(\mathcal{E}, \tau)(\sigma \cup \sigma'_1 \cup \sigma''_1))$$

By the lemma on observable substitutions, we get $s' : \sigma \cup \sigma'_1 \cup \sigma''_1, \mathcal{S}' \models v : \theta'\tau$. By the lemma on side-effects, we also have $s' : \sigma \cup \sigma'_1 \cup \sigma''_1, \mathcal{S}' \models E : \mathcal{E}$. So, by the definition of $\models$, we have $s' : \sigma \cup \sigma'_1 \cup \sigma''_1, \mathcal{S}' \models E_{\mathtt{x}} \cup \{\mathtt{x} \mapsto v\} : \mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \forall v_{1..n}.\tau\}$.

By induction hypothesis on $\mathtt{e}'$, there exist $\mathcal{S}''$ extending $\mathcal{S}'$ and an unobservable effect σ''_2 such that $Observe(\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \forall v_{1..n}.\tau\}, \tau')(\sigma''_2) = \emptyset$ verifying:

$$\begin{array}{l} \mathcal{S}'' \models s'' : \sigma \cup \sigma'_1 \cup \sigma''_1 \cup \sigma'_2 \cup \sigma''_2 \\ s'' : \sigma \cup \sigma'_1 \cup \sigma''_1 \cup \sigma'_2 \cup \sigma''_2, \mathcal{S}'' \models v' : \tau' \end{array}$$

Let us note $\sigma'' = \sigma''_1 \cup \sigma''_2$. Then $\sigma'_1 \cup \sigma''_1 \cup \sigma'_2 \cup \sigma''_2 = \sigma' \cup \sigma''$ and $Observe(\mathcal{E}, \tau')(\sigma'') = \emptyset$. We have proved that there exist $\mathcal{S}''$ extending $\mathcal{S}$ and σ'' such that $Observe(\mathcal{E}, \tau')(\sigma'') = \emptyset$ verifying:

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

6 Inference Algorithm

We present the inference algorithm $\mathcal{I}$ that reconstructs the type and effect of an expression with respect to the static semantics. The reconstruction algorithm satisfies the criteria of type principality, with respect to substitution on variables, and effect minimality, with respect to the subsumption rule on effects. The algorithm $\mathcal{I}$ uses a double recursion scheme to separate the syntax-directed reconstruction of types and effects from the process of restricting effects with regard to the observation criterium.

Constraint Satisfaction

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. An important invariant of our method is that the latent effect of a function is always represented, in the algorithm, by an effect variable. This makes the problem of solving equations amenable to a Robinson-like unification algorithm [16] used on type, region and effect variables.

An effect inequation $\varsigma \sqsupseteq \sigma$ represents a constraint between an observed lower bound σ and an inferred effect variable ς. Such inequations are gathered in constraint systems κ and are built during the processing of `lambda` expressions, which is the place where effects are introduced into types.

When constraint sets κ are satisfiable (section 7 defines the notion of well-formed constraint sets), their solutions are substitutions θ that verify $\theta\varsigma \sqsupseteq \theta\sigma$ for every inequation $\varsigma \sqsupseteq \sigma$ in κ. This is noted $\theta \models \kappa$. For every substitution θ satisfying a constraint set κ, there exists a substitution θ' such that $\theta = \theta'\overline{\kappa}$, where $\overline{\kappa}$ is inductively defined by $\overline{\{\}} = Id$ and $\overline{\kappa \cup \{\varsigma \sqsupseteq \sigma\}} = \overline{\kappa}_\varsigma\{\varsigma \mapsto \overline{\kappa}(\varsigma \cup \sigma)\}$. Note that, by definition, the substitution $\overline{\kappa}$ is defined whether or not κ is satisfiable. A minimal substitution, with respect to the subsumption relation on effects, can be derived from $\overline{\kappa}$ by binding free effect variables to $\emptyset$, when κ is satisfiable.

Unification

Equations on types, that are built by the reconstruction algorithm, are solved with the straightforward Robinson-like unification algorithm $\mathcal{U}$ given below. It either fails or returns a substitution θ as the most general unifier of the two given type terms.

$$\begin{array}{l}
\mathcal{U}(\tau, \tau') = case\ (\tau, \tau')\ of \\
(unit, unit) \Rightarrow Id \\
(\alpha, \alpha') \Rightarrow \{\alpha \mapsto \alpha'\} \\
(\alpha, \tau) \mid (\tau, \alpha) \Rightarrow if\ \alpha \in fv(\tau)\ then\ fail\ else\ \{\alpha \mapsto \tau\} \\
(ref_\gamma(\tau), ref_{\gamma'}(\tau')) \Rightarrow let\ \theta = \{\gamma \mapsto \gamma'\}\ in\ \mathcal{U}(\theta\tau, \theta\tau')\theta \\
(\tau_i \xrightarrow{\varsigma} \tau_f, \tau'_i \xrightarrow{\varsigma'} \tau'_f) \Rightarrow \\
\quad let\ \theta = \{\varsigma \mapsto \varsigma'\}\ and\ \theta' = \mathcal{U}(\theta\tau_i, \theta\tau'_i) \\
\quad in\ \mathcal{U}(\theta'\theta\tau_f, \theta'\theta\tau'_f)\theta'\theta
\end{array}$$

Syntax-Directed Reconstruction

First phase of the reconstruction, the algorithm $\mathcal{I}'$ returns, for an expression $\mathtt{e}$ in an environment $\mathcal{E}$, its type τ, its effect σ, a substitution θ that ranges over the free variables of the environment $\mathcal{E}$ and a constraint system κ.

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

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

$$\begin{array}{l}
\mathcal{I}'(\mathcal{E}, (\mathtt{let\ (x\ e)\ e'})) = \\
\quad let\ \langle \theta, \tau, \sigma, \kappa\rangle = \mathcal{I}(\mathcal{E}, \mathtt{e}) \\
\qquad \mathcal{E}' = \theta\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto Gen_\kappa(\sigma, \theta\mathcal{E})(\tau)\} \\
\qquad \langle \theta', \tau', \sigma', \kappa'\rangle = \mathcal{I}(\mathcal{E}', \mathtt{e'}) \\
\quad in\ \langle \theta'\theta, \tau', \theta'\sigma \cup \sigma', \theta'\kappa \cup \kappa'\rangle
\end{array}$$

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

The type environment $\mathcal{E}$ binds value identifiers to constrained type schemes. A *constrained type scheme* $\forall \upsilon_{1..n}.(\tau, \kappa)$ is defined as a type τ and a constraint set κ quantified over a set of type, region and effect variables $\upsilon_{1..n}$. In the static semantics, type schemes consisted of a type quantified over a set of variables. In the algorithm, since effect variables occur in function types, constraint sets, which relate these effect variables to their reconstructed lower bounds, have to be kept within these constrained type schemes.

Generalization

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 effects σ.

$$Gen_\kappa(\sigma, \mathcal{E})(\tau) = \\ let\ \{\upsilon_{1..n}\} = fv_\kappa(\tau) \setminus (fv_\kappa(\mathcal{E}) \cup fv_\kappa(\sigma))\ in\ \forall \upsilon_{1..n}.(\tau, \kappa)$$

For any constraint set κ, the function fv_κ, defined as $fv_\kappa(\tau) = fv(\overline{\kappa}\tau)$, computes the set of free type, region and effect variables of its argument. It is similar to the definition of fv found in the static semantics, except that we add for each effect variable the set of all free variables that occur in its lower bounds present in κ.

It is extended to constrained type schemes by $fv_\kappa(\forall \upsilon_{1..n}.(\tau, \kappa')) = fv_{\kappa \cup \kappa'}(\tau) \setminus \{\upsilon_{1..n}\}$. The function fr_κ is structurally equivalent to fv_κ and collects free regions.

Observation

In its second phase, the algorithm $\mathcal{I}$ takes into account the observation criteria to restrict the effect σ computed by the algorithm $\mathcal{I}'$.

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

In σ, the effect variables ς that are not present in the context are not be observable, but their lower bounds, reconstructed in κ, may be. The observation function $Observe_\kappa$, defined as $Observe_\kappa(\mathcal{E}, \tau)(\sigma) = Observe(\overline{\kappa}\mathcal{E}, \overline{\kappa}\tau)(\overline{\kappa}\sigma)$, determines the effects that can be observed from the surrounding context.

Initial Environment

The type and effect reconstruction algorithm $\mathcal{I}$ uses the initial environment $\mathcal{E}_0$ to bind appropriate constrained type schemes to the imperative constructs `new`, `get` and `set`.

$$\mathtt{new}\ :\ \forall \alpha\rho\varsigma.(\alpha \xrightarrow{\varsigma} ref_\rho(\alpha), \{\varsigma \sqsupseteq init(\rho, \alpha)\})$$
$$\mathtt{get}\ :\ \forall \alpha\rho\varsigma.(ref_\rho(\alpha) \xrightarrow{\varsigma} \alpha, \{\varsigma \sqsupseteq read(\rho)\})$$
$$\mathtt{set}\ :\ \forall \alpha\rho\varsigma\varsigma'.(ref_\rho(\alpha) \xrightarrow{\varsigma} \alpha \xrightarrow{\varsigma'} unit, \{\varsigma' \sqsupseteq write(\rho)\})$$

7 Solving Constraint Sets

In our type system, effects σ occur in types $\tau \xrightarrow{\sigma} \tau'$ as well as types τ occur in effects $init(\rho, \tau)$. As a consequence, in the static semantics, some expressions may have to be rejected, in that they would require recursively defined types and effects: the effect σ of some expression may be required, by the rule of the static semantics, to contain $init(\rho, \tau \xrightarrow{\sigma} \tau')$ itself [9].

In the reconstruction algorithm, such requirements result in constraints $\kappa = \kappa' \cup \{\varsigma \sqsupseteq \sigma\}$ where $init(\rho, \tau \xrightarrow{\varsigma} \tau')$ occurs in $\overline{\kappa'}\sigma$. A model for ς would have to be defined by a fixed point inequation, requiring an infinite term to be substituted for $init(\rho, \tau \xrightarrow{\varsigma} \tau')$. The simplest known example producing such an ill-formed constraint set is:

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

The algorithm returns $(\alpha \xrightarrow{\varsigma} \alpha) \xrightarrow{\varsigma'} (\alpha \xrightarrow{\varsigma} \alpha)$ and $\{\varsigma \sqsupseteq write(\rho') \cup init(\rho, \alpha \xrightarrow{\varsigma} \alpha')\}$, though this expression can not be typed in the static semantics. Note that the incriminated effect $init(\rho, \alpha \xrightarrow{\varsigma} \alpha')$ must be observable for this situation to appear.

Formulation of the Static Semantics

In [8], the authors present a type system that also introduces a relation between function types and sets of types. To overcome the problem presented here, they specify a static semantics that closely parallels the reconstruction algorithm using indirections $\{\tau\} \rhd L$, via the notion of *labels* L, between function types $\tau \xrightarrow{L} \tau'$ and sets of *dangerous* types $\{\tau\}$. However, this representation seriously impedes the intuitive understanding of the type system, that would normally be profitable to users of such a system. For instance, their following typing:

$$(\mathtt{new}\ (\mathtt{lambda}\ (\mathtt{x})\ (+\ \mathtt{x}\ \mathtt{1}))) : \\ ref(int \xrightarrow{\varsigma} int)\ with\ \{int\} \rhd \varsigma$$

requires free effect variables such as ς. With such a representation, expressions cannot always be associated with a closed type term. This limitation becomes serious when integrating the language into a module system: only expressions having a *closed* type could be accepted in a module.

In [21], function types are associated with sets of type variables in order to break the similar kind of cycles that would otherwise occur. Putting free variables instead of type terms on the arrow of function types leads to a complicated notion of substitution, which also impedes the easy understanding of the type system. For the previous example, one would get:

$$(\mathtt{new}\ (\mathtt{lambda}\ (\mathtt{x})\ (+\ \mathtt{x}\ \mathtt{1}))) : ref(int \xrightarrow{\varsigma} int), \{\varsigma\}$$

In this system, one can replace the effect variable ς by $\emptyset$ to produce a correct type $ref(int \xrightarrow{\emptyset} int)$. However, the resulting effect $\{\emptyset\}$ does not reflect a proper inferred effect.

Our formulation of the static semantics has the advantage to allow every expression to have a closed type and effect. It could thus be easily integrated into existing module systems. For our example, we get:

$$\texttt{(new (lambda (x) (+ x 1)))} : \\ ref_\rho(int \xrightarrow{\emptyset} int), init(\rho, int \xrightarrow{\emptyset} int)$$

Effect inference thus provides easy to understand intuitions about the semantics of programs. It can be effectively implemented with the proviso that contraint sets must be checked for well-formedness, namely that no indirect cycles are introduced through *init* effects.

Well-Formed Constraint Sets

We specify the set of well-formed constraint sets that are acceptable to the algorithm $\mathcal{I}$ and that correspond to sound assignments of effect variables in the static semantics.

Definition 4 (Well Formed Constraint Set) *A constraint set κ is well formed, noted $wf(\kappa)$, if and only if, for every $\varsigma \sqsupseteq \sigma$ such that $\kappa = \kappa' \cup \{\varsigma \sqsupseteq \sigma\}$ we have:*

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

The notation $wf(\kappa)$ is extended to type schemes by $wf(\forall v_{1..n}.(\tau, \kappa))$ iff $wf(\kappa)$ and type environments by $wf(\mathcal{E})$ iff $wf(\mathcal{E}(\mathtt{x}))$ for every $\mathtt{x}$ in $Dom(\mathcal{E})$.

In the reconstruction algorithm $\mathcal{I}$, of course, instead of checking the well-formedness of the constructed constraint set after the expression is type-checked, we could implement an extended occurence check test, reporting the construction of ill-formed constraints at the point of unifying effect variables.

$$\mathcal{U}_\kappa(\varsigma, \varsigma') \ = \ \textit{if } wf(\{\varsigma \mapsto \varsigma'\}\kappa) \textit{ then } \{\varsigma \mapsto \varsigma'\} \textit{ else fail}$$

This solution would ensure that κ is well-formed for every $\langle \theta, \tau, \sigma, \kappa \rangle$ returned by the algorithm $\mathcal{I}(\mathcal{E}, \mathtt{e})$.

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, and that ill-formed constraint sets do not admit finite substitutions as models.

Lemma 5 (Constraint Sets) *For every constraint set κ, $wf(\kappa)$ if and only if $\overline{\kappa} \models \kappa$. If κ is not well-formed, then there does not exist a finite substitution modeling κ.*

8 Correctness Theorem

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. We note $\overline{\forall v_{1..n}.(\tau, \kappa)}$ for $\forall v_{1..n}.\overline{\kappa}\tau$ and define $\overline{\mathcal{E}}$ by extension.

Theorem 2 (Soundness) *Let $\mathcal{E}$ be a well-formed type environment. If $\mathcal{I}(\mathcal{E}, \mathtt{e}) = \langle \theta, \tau, \sigma, \kappa \rangle$ and $wf(\kappa)$, then $\theta'\theta\overline{\mathcal{E}} \vdash \mathtt{e} : \theta'\tau, \theta'\sigma$ for any $\theta' \models \kappa$.*

Proof sketch Assume that $\mathcal{I}(\mathcal{E}, \mathtt{e}) = \langle \theta, \tau, \sigma, \kappa \rangle$ and $wf(\kappa)$. Since $\theta' = \theta''\overline{\kappa}$ for any substitution θ' modeling κ, whenever we have $\overline{\kappa}\theta\overline{\mathcal{E}} \vdash \mathtt{e} : \overline{\kappa}\tau, \overline{\kappa}\sigma$ then $\theta'\theta\overline{\mathcal{E}} \vdash \mathtt{e} : \theta'\tau, \theta'\sigma$ follows by the substitution lemma. It is thus sufficient to prove, for $\mathcal{E}$ well-formed, that $\mathcal{I}(\mathcal{E}, \mathtt{e}) = \langle \theta, \tau, \sigma, \kappa \rangle$ and $wf(\kappa)$ imply $\overline{\kappa}\theta\overline{\mathcal{E}} \vdash \mathtt{e} : \overline{\kappa}\tau, \overline{\kappa}\sigma$.

We describe the proof for the case of the abstraction. Let $\mathcal{E}$ be well-formed and assume that $\mathcal{I}(\mathcal{E}, (\texttt{lambda (x) e})) = \langle \theta, \theta\alpha \xrightarrow{\varsigma} \tau, \emptyset, \kappa \cup \{\varsigma \sqsupseteq \sigma\} \rangle$ and $wf(\kappa \cup \{\varsigma \sqsupseteq \sigma\})$. By definition of the algorithm:

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

Since $wf(\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \alpha\})$ and $wf(\kappa)$, by induction hypothesis on $\mathtt{e}$, we get $\overline{\kappa}\theta(\overline{\mathcal{E}}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \alpha\}) \vdash \mathtt{e} : \overline{\kappa}\tau, \overline{\kappa}\sigma$. Since ς is new and $\overline{\kappa}\sigma \cup \varsigma = \overline{\kappa}(\sigma \cup \varsigma) \sqsupseteq \overline{\kappa}\sigma$, we have, by the rule *(sub)*:

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

Now, $\overline{\kappa}' = \overline{\kappa \cup \{\varsigma \sqsupseteq \sigma\}} = \overline{\kappa}\{\varsigma \mapsto \overline{\kappa}(\varsigma \cup \sigma)\}$. Since $\overline{\kappa}' \models \kappa$ and $\overline{\kappa}'\varsigma = \overline{\kappa}(\sigma \cup \varsigma)$, we have $\overline{\kappa}'\theta(\overline{\mathcal{E}}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \alpha\}) \vdash \mathtt{e} : \overline{\kappa}'\tau, \overline{\kappa}'\varsigma$. By the definition of the rule *(abs)*, we get:

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

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

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

Proof sketch The proof is by induction on the structure of expressions. We use the fact that $\theta'\theta\overline{\mathcal{E}} = \theta'\overline{\theta\mathcal{E}}$.

We outline the proof in the case of the application. The hypothesis is $\theta''\overline{\mathcal{E}} \vdash (\mathtt{e_1}\ \mathtt{e_2}) : \tau', \sigma'$. By the definition of the rule *(app)*, there exist $\sigma'_{1..3}$ such that

$\sigma' = \sigma'_1 \cup \sigma'_2 \cup \sigma'_3$ verifying:

$$\theta''\overline{\mathcal{E}} \vdash \mathtt{e_1} : \tau'_2 \xrightarrow{\sigma'_3} \tau', \sigma'_1 \ \ and \ \theta''\overline{\mathcal{E}} \vdash \mathtt{e_2} : \tau'_2, \sigma'_2$$

By induction hypothesis on $\mathtt{e_1}$, $\mathcal{I}(\mathcal{E}, \mathtt{e_1})$ returns $\langle \theta_1, \tau_1, \sigma_1, \kappa_1 \rangle$ and there exists θ'_1 modeling κ_1 such that:

$$\theta''\overline{\mathcal{E}} = \theta'_1\theta_1\overline{\mathcal{E}} \ \ and \ \tau'_2 \xrightarrow{\sigma'_3} \tau' = \theta'_1\tau_1 \ \ and \ \sigma'_1 \sqsupseteq \theta'_1\sigma_1$$

Since $\theta'_1\theta_1\overline{\mathcal{E}} = \theta'_1\overline{\theta_1\mathcal{E}}$ and $\theta_1\mathcal{E}$ is well formed, by induction hypothesis on $\mathtt{e_2}$, we get $\mathcal{I}(\theta_1\mathcal{E}, \mathtt{e_2}) = \langle \theta_2, \tau_2, \sigma_2, \kappa_2 \rangle$ and there exists θ'_2 modeling κ_2 such that:

$$\theta''\overline{\mathcal{E}} = \theta'_2\theta_2\overline{\theta_1\mathcal{E}} \ \ and \ \tau'_2 = \theta'_2\tau_2 \ \ and \ \sigma'_2 \sqsupseteq \theta'_2\sigma_2$$

Let V be the set of free variables in $\theta_2\overline{\theta_1\mathcal{E}}$, τ_2, σ_2 and κ_2. Take α and ς new and define θ'_3 as follows:

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

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

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

Now, for every υ in τ_1, σ_1 and κ_1, either υ is in $fv(\overline{\theta_1\mathcal{E}}) = fv(\theta_1\overline{\mathcal{E}})$ or υ is new, by definition of $\mathcal{I}$. Then, for every such υ in $fv(\theta_1\overline{\mathcal{E}})$, since $\theta'_3\theta_2\overline{\theta_1\mathcal{E}} = \theta'_2\theta_2\overline{\theta_1\mathcal{E}} = \theta'_1\theta_1\overline{\mathcal{E}}$, we have $\theta'_3\theta_2\upsilon = \theta'_2\theta_2\upsilon = \theta'_1\upsilon$. Otherwise, υ is new, and thus $\upsilon \notin Dom(\theta_2)$, so that we have $\theta'_3\theta_2\upsilon = \theta'_3\upsilon = \theta'_1\upsilon$. We thus get:

$$\tau'_2 \xrightarrow{\sigma'_3} \tau' = \theta'_3\theta_2\tau_1 \ \ and \ \theta'_1\sigma_1 = \theta'_3\theta_2\sigma_1 \ \ and \ \theta'_3 \models \theta_2\kappa_1$$

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

$$\theta = \theta_3\theta_2\theta_1 \ \ and \ \kappa = \theta_3(\theta_2\kappa_1 \cup \kappa_2)$$
$$\tau = \theta_3\alpha \ \ and \ \sigma_3 = \theta_3(\theta_2\sigma_1 \cup \sigma_2 \cup \varsigma)$$

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 $\theta'_3 = \theta'\theta_3$. Since $\theta'_3 \models \theta_2\kappa_1 \cup \kappa_2$, we get:

$$\theta' \models \kappa$$
$$\theta''\overline{\mathcal{E}} = \theta'\theta\overline{\mathcal{E}}$$
$$\tau' = \theta'\tau \ \ and \ \sigma'_1 \cup \sigma'_2 \cup \sigma'_3 \sqsupseteq \theta'\sigma_3$$

$\mathcal{I}(\mathcal{E}, (\mathtt{e_1}\ \ \mathtt{e_2}))$ returns $\langle \theta, \tau, \sigma, \kappa \rangle$ where σ is $Observe_\kappa(\theta\mathcal{E}, \tau)(\sigma_3)$. Since $Observe(\theta'\theta\overline{\mathcal{E}}, \theta'\tau)(\sigma') \sqsupseteq \theta' Observe_\kappa(\theta\mathcal{E}, \tau)(\sigma_3)$, we conclude that:

$$\theta' \models \kappa$$
$$\langle \theta, \tau, \sigma, \kappa \rangle = \mathcal{I}(\mathcal{E}, (\mathtt{e_1}\ \ \mathtt{e_2}))$$
$$\theta''\overline{\mathcal{E}} = \theta'\theta\overline{\mathcal{E}} \ \ and \ \tau' = \theta'\tau \ \ and \ \sigma' \sqsupseteq \theta'\sigma \ \square$$

9 Examples

To demonstrate the effectiveness of our algorithm for generalizing types as well as its capability of assigning similar typings to purely functional expressions and their related variants using locally defined mutable data structures, let us consider some well-known examples of list processing functions. To this end, we introduce the type constructor *list* of immutable lists with the following value and functions:

$$\begin{array}{rl} \mathtt{nil}: & \forall\alpha.list(\alpha) \\ \mathtt{cons}: & \forall\alpha\varsigma.\alpha \times list(\alpha) \xrightarrow{\varsigma} list(\alpha) \\ \mathtt{null?}: & \forall\alpha\varsigma.list(\alpha) \xrightarrow{\varsigma} boolean \\ \mathtt{car}: & \forall\alpha\varsigma.list(\alpha) \xrightarrow{\varsigma} \alpha \\ \mathtt{cdr}: & \forall\alpha\varsigma.list(\alpha) \xrightarrow{\varsigma} list(\alpha) \end{array}$$

This first example shows that defining the function `fold` using temporary locatives does not affect its typing. We extended our language syntax with multiple binding `let` and the looping `until` construct.

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

$$\forall\alpha\alpha'\varsigma\varsigma'\varsigma''.(\alpha \times \alpha' \xrightarrow{\varsigma} \alpha') \times \alpha' \xrightarrow{\varsigma'} (list(\alpha) \xrightarrow{\varsigma\cup\varsigma''} \alpha')$$

A nice consequence is that the function `reverse` can then be defined in terms of `fold` regardless of the details of its implementation:

```
(define reverse (fold cons nil))
```
$$\forall\alpha\varsigma.list(\alpha) \xrightarrow{\varsigma} list(\alpha)$$

In a very similar manner, we define the function `map`, using `reverse` and imperative constructs. It is given the very same type as its usual functional defi-

nition in ML:

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

$$\forall \alpha\alpha'\varsigma\varsigma'.(\alpha \xrightarrow{\varsigma} \alpha') \times list(\alpha) \xrightarrow{\varsigma\cup\varsigma'} list(\alpha')$$

So, for instance, the application of `map` to the identity function and the empty list has the polymorphic type of `nil` in `let` bindings.

$$\texttt{(map (lambda (x) x) nil)} \ : \ \forall\alpha.list(\alpha)$$

On the contrary, the application of `map` to `new` and `nil` has a monomorphic type accounting for the use of the function `new` on a region ρ with an observable effect.

$$\texttt{(map new nil)} \ : \ list(ref_\rho(\alpha)), init(\rho, \alpha)$$

Comparison with Related Work

The following example, quoted from [8], shows that the observation of effects is also 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 [8] or [21], which are not able to deal with the spurious `(new x)`.

```
(lambda (z)
  (let (id1 (lambda (x)
              (if true z (lambda (y)
                           (new x)
                           y))
              x))
    (id1 id1)))
```

The next table gives a detailled comparison using a series of examples adapted from a survey paper on this subject [15]. The figures for [7] are given for programs with explicit polymorphic types.

```
(define id2 (let (x (id 1)) rid))
(define id3 (lambda (y) ((rid id) y)))
(define id4 ((nop rid) id))
```

We note `rid` for the imperative identity and `id` for the usual functional identity . Also, we define the tricky function `nop`.

```
(define id (lambda (x) x))
(define rid (lambda (x) (get (new x))))
(define nop (lambda (f)
              (lambda (x)
                (let (g (lambda (y) (f x)))
                  x))))
```

The following benchmark suggests that our type and effect discipline, giving type $\forall\alpha\varsigma.\alpha \xrightarrow{\varsigma} \alpha$ to all the given examples, favorably competes with some earlier polymorphic type generalization policies.

Example	[20]	[3]	[1]	[7]	[21]	[8]
`id1`	yes	yes	yes	no	no	no
`id2`	no	yes	no	yes	yes	yes
`id3`	yes	yes	no	yes	yes	yes
`id4`	no	no	no	yes	yes	yes
`(id1 id1)`	no	no	no	no	no	no
`(id2 id2)`	no	no	no	no	no	yes
`(id3 id3)`	no	no	no	no	no	yes
`(id4 id4)`	no	no	no	yes	yes	yes

10 Extensions

The approach presented in this paper could easily be extended to deal with recursive function definitions in a way similar to [18]. It is also amenable to integration into existing module systems [1, 17].

Continuations

An appealing direction for further extensions is the treatment of exceptions and continuations, which shares very similar typing requirements with reference values.

Let us define the type $cont_\rho(\tau)$ for continuation values, as in [4], and the effects $from(\rho, \tau)$ and $goto(\rho)$ for allocating and invoking a continuation, as in [6]. The typing of continuations could then be handled in a manner very similar to reference values, after given the following appropriate types to the related primitive functions:

$$\texttt{callcc} \ : \ \forall\alpha\rho\varsigma\varsigma'.(cont_\rho(\alpha) \xrightarrow{\varsigma} \alpha) \xrightarrow{\varsigma'\cup\varsigma\cup from(\rho,\alpha)} \alpha$$

$$\texttt{throw} \ : \ \forall\alpha\alpha'\rho\varsigma\varsigma'.(cont_\rho(\alpha) \xrightarrow{\varsigma} \alpha) \xrightarrow{\varsigma'\cup\varsigma\cup goto(\rho)} \alpha'$$

The observation criterium helps here to delimit the scope of captured continuations. In a very similar way, the semantics of our language can be extended with

the treatment of exceptions, as in Standard ML [1]. However, since the related features cannot be represented by functions, this extension results in updating the syntax and static semantics of the language.

Recursive Effects

The example given in section 7 cannot be handled with our current type and effect system since it would require recursively defined effects in the static semantics to typecheck. Following the approach of [2] for recursive types, we could allow recursively defined effects $\mu\varsigma.\sigma$, with the added equivalence rule:

$$(equiv) : \frac{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma \quad \tau \simeq \tau' \quad \sigma \simeq \sigma'}{\mathcal{E} \vdash \mathtt{e} : \tau', \sigma'}$$

The addition of the rule *(equiv)* to the static semantics would allow equivalent types and effects for a given expression according to the relation $\simeq$. This relation would be structurally defined on type and effect terms with the additional unrolling axiom:

$$\mu\varsigma.\sigma \simeq \{\varsigma \mapsto \mu\varsigma.\sigma\}(\sigma)$$

The model mapping ς to $\mu\varsigma'.(init(\rho, \alpha \xrightarrow{\varsigma'} \alpha') \cup \varsigma \cup write(\rho'))$ would thus be a correct assignment for the example of section 7.

11 Conclusion

We have introduced the *type and effect discipline* as a new framework for reconstructing the principal type and the minimal observable effect of expressions in implicitly typed polymorphic functional languages that support imperative operations on references.

The initial design goal of polymorphic effect systems [10, 11] was to safely integrate functional and imperative constructs. Using effect masking, we showed how effect systems can be put to work for solving the type reconstruction problem in the presence of side-effects.

More specifically, by using effect information together with an observation criterium, 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.

We proved that the language static semantics is consistent with respect to its dynamic semantics and presented a reconstruction algorithm that computes the principal type and the minimal observable effects of expressions.

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

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

[2] Cardone, F., and Coppo, M. Type Inference with Recursive Types: Syntax and Semantics. In *Information and Computation*, Vol. 92, pages 48-80. Academic Press 1991.

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

[4] 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.

[5] 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.

[6] Jouvelot, P., and Gifford, D. K. Reasoning about Continuations with Control Effects. In *Proceedings of the 1989 ACM Conference on Programming Language Design and Implementation*. ACM, New-York, 1989.

[7] 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.

[8] 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.

[9] Leroy, X. Private communication. December 1991.

[10] 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.

[11] 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.

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

[13] Milner, R., Tofte, M., Harper, R. The definition of Standard ML. *The MIT Press*, Cambridge, 1990.

[14] 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.

[15] O'Toole, J. W. Type abstraction rules for references: a comparison of four which have achieved notoriety. *Technical Note of the PSRG Group*, MIT Laboratory for Computer Science, April 1989.

[16] 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.

[17] 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.

[18] Talpin, J. P., and Jouvelot, P. *Polymorphic Type, Region and Effect Inference*. To appear in the *Journal of Functional Programming*, Vol. 2, No. 2, Cambridge University Press, 1992. Also in Technical Report EMP-CRI-E150, Ecole Nationale Supérieure des Mines de Paris, February 1991 (revised December 1991).

[19] Talpin, J. P., and Jouvelot, P. *The Type and Effect Discipline*. Technical Report EMP-CRI-A206, Ecole Nationale Supérieure des Mines de Paris, March 1992.

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

[21] Wright, A. K. Polymorphic References for Mere Mortals. In *Proceedings of the 1992 European Symposium on Programming*. Springer Verlag, 1992.