Algebraic Reconstruction of Types and Effects

Pierre Jouvelot[1,2]
David K. Gifford[2]

Ecole des Mines de Paris[1]
MIT Laboratory for Computer Science[2]

Abstract

We present the first algorithm for reconstructing the types and effects of expressions in the presence of first class procedures in a polymorphic typed language. Effects are static descriptions of the dynamic behavior of expressions. Just as a type describes what an expression computes, an effect describes how an expression computes. Types are more complicated to reconstruct in the presence of effects because the algebra of effects induces complex constraints on both effects and types. In this paper we show how to perform reconstruction in the presence of such constraints with a new algorithm called *algebraic reconstruction*, prove that it is sound and complete, and discuss its practical import.

This research was supported by DARPA under ONR Contract N00014-89-J-1988.

1 Introduction

Type reconstruction systems compute the type declarations that a programmer has omitted from a program, and thus provide a programmer with the performance and safety benefits of static typing without the burden of writing declarations. Type reconstruction may also aid software reuse because it always guarantees to find the most general type of a program, while most programmers simply use declarations that suffice for the immediate application at hand. Because of their utility, type reconstruction systems are now finding their way into modern programming languages [M78] and are the subject of much theoretical and practical interest.

We examine in this paper how declaration reconstruction can be extended to effect systems. An effect system [LG88] is a dual to a type system. Just as types describe what expressions compute, effects describe how expressions compute. In a language with an effect system, every procedure type includes a *latent effect*. A latent effect is used to communicate the side-effects of a procedure body from the point of its definition (via lambda abstraction) to the points of its use (via application). Because types include effects, type reconstruction and effect reconstruction are necessarily dependent on one another.

We are interested in effect systems because they can be used to compute a wide variety of useful properties about a program. To date we have used them to compute store effects [GJLS87] (`read`, `write`, and `initialize` regions of the store), communication effects [JG89a] (`in` and `out` effects on channels), and control effects [JG89b] (`goto` and `comefrom`). Store effects can be used in direct support of parallel computing because they determine an expression dependency graph that can be used to automatically schedule expressions for parallel execution [HG88].

Morris [M68] recognized that type reconstruction is a constraint satisfaction problem. All that is necessary to reconstruct the types of expressions in a program is to gather together all of the constraints placed on the variables by expressions, and then compute the most general type that could be safely assigned to the variables subject to the constraints.

Contemporary type reconstruction systems use structural constraints, and thus we will refer to them as *structural reconstruction* systems. Structural reconstruction systems use Robinson's algorithm [R65] to solve constraints based upon structure matching in a set of simultaneous equations. For example, ML [M78] uses a structural reconstruction system that is based upon Robinson's algorithm.

Effect reconstruction is also a constraint satisfaction problem, but the algebra of effects is richer than most type algebras. In specific, effects form a commutative monoid with idempotence under the effect union operator, `maxeff`. Thus when a program undergoes type and effect reconstruction a system of simultaneous algebraic equations is generated. A program is well-typed if and only if the system of algebraic equations induced by a program has a solution.

Structural reconstruction can only handle free algebras, and thus it can not be used to reconstruct types in the presence of effects. There are two underlying causes to this limitation. First, Robinson's algorithm is limited to the syntactic equivalence of expressions, and thus can not deal with complex algebraic axioms. Second, structural reconstruction is based upon the idea that every expression has a single most general type that can be specialized by simple syntactic substitutions.

We have developed a new *algebraic reconstruction* system for types and effects that uses algebraic constraints to overcome the problems of structural reconstruction. In our algebraic reconstruction system, an expression is *well-typed* iff a given constraint system of algebraic equations is *satisfiable*. Algebraic reconstruction is based upon: (1) a unification algorithm that solves sets of algebraic equations, and (2) a generalization of principal type schemes beyond syntactic substitution to include constraints.

As we discuss at the end of this paper, algebraic reconstruction can also be applied to complex type algebras. For example, computing the field names of a record is a simple instance of a constraint-based problem that is not handled in a natural way by Robinson's unification algorithm.

In the remainder of this paper we discuss previous work (Section 2), a static semantics for a simple language with types and effects (Section 3), our reconstruction algorithm (Section 4), proofs of soundness and completeness (Section 5), extension to ML-style generic polymorphism (section 6) and a view towards the import and application of our result (Section 7).

2 Related Work

The closest related work to algebraic reconstruction is recent research that extends structural reconstruction to cope with the algebraic properties of record types. Record types are a bit more complex than other types because: (1) the field names that must appear in a record type are discovered incrementally, and (2) the order of field names in a record type often does not matter. For example, Jategaonkar [J89] shows how to add the algebraic properties of record types to a structural reconstruction system and proves her resulting system sound and complete. Wand [W89] extends [W87] by considering concatenation of records as a way of formalizing multiple inheritance. Wand uses disjunctions of constraints on record types to generate multiple possible types for an expression.

Any kind of structural reconstruction depends upon the existence of syntactic principal types. A *syntactic* principal type describes all of the possible types of an expression via permissible syntactic substitutions for certain type variables. Syntactic principal types can often be preserved when one wants to extend the language with constructs such as records [C84, W87], although one has to be careful not to introduce too much flexibility [W88]. In the latter case, syntactic principal typing is lost because a simple syntactic substitution is not powerful enough to describe the set of types an expression might have.

The advantage of algebraic reconstruction is that it can cope with complex algebraic constraints on types that can not be handled by extended structural reconstruction. For example, structural reconstruction can not be extended to deal with the algebra of effects because expressions no longer have a syntactic principal type. The algebra of effects, introduced in [LG88], has a wealth of properties (e.g., commutativity and idempotence) that makes principal typing modulo substitution impossible.

The ACUI-unification (Associative, Commutative, Unitary, Idempotent) procedure [LC89] is also related to our work because we will use it to show the decidability of type and effect reconstruction. Although ACUI-unification is NP-complete [KN86], we expect most cases to be tractable, in the same manner ML's inference system has proved to be useful while being DEXPTIME-complete [M90]. In fact, since we don't have to deal with free function symbols, the full power of ACUI-unification is not needed and we show how a simpler polynomial algorithm [MB90] can be used instead to solve constraints. We will discuss this issue in more detail later in the paper.

Other related work seeks to understand the ultimate semantic limits of structural reconstruction. Although ML doesn't provide the full power of the second-order polymorphic lambda-calculus [FLO83], [GR88] shows how this constraint-based approach can be extended to the full-fledged calculus. Partial type reconstruction for the full language is known to be undecidable [B85, P88], even though the complete problem is still open. [OG89] proposes an interesting way to mix together ML's generic type schemes with explicit polymorphic types while preserving type reconstruction decidability on non-polymorphic types.

3 The Type and Effect System

In this section we present the type and effect system that we will use throughout the paper. For pedagogical purposes, we will study effect reconstruction in the context of KFX, a simplified version of the FX-87 language. FX-87 [GJLS87] is a kinded polymorphic typed language that allows side-effects and first-class procedures and uses an effect system. The language KFX has the following Kind, Description (Effect and Type) and Expression domains:

K	::=	`effect` | `type`	
D	::=	`F` | `T`	
F	::=	`I`	
		(`maxeff` $\mathtt{F}_0$ $\mathtt{F}_1$)	combination of effects
T	::=	`M`	
		(`proc` `F` ($\mathtt{T}_0$) $\mathtt{T}_1$)	procedure
		(`poly` (`I` `K`) `T`)	polymorphic
M	::=	`I`	
		(`proc` `F` ($\mathtt{M}_0$) $\mathtt{M}_1$)	procedure
E	::=	`I`	
		(`lambda` (`I` `T`) `E`)	typed lambda
		(`lambda` (`I`) `E`)	implicitly typed lambda
		(`plambda` (`I` `K`) `E`)	polymorphic abstraction
		(`proj` `E` `D`)	projection
		($\mathtt{E}_0$ $\mathtt{E}_1$)	application
I	::=	`N`	constant
		`V`	user variable
		ν	unification variable

`M` is the class of types that will be inferred by our algorithm. It consists of variables and procedures that do not include polymorphic types in them. Note that procedure types include a latent effect, and that inferred procedure types are not required to have `pure` latent effects. Unification variables cannot appear in user programs.

KFX descriptions have a much richer algebra than normal type expressions. The conversion equivalence relation $\sim$ between descriptions, beside the usual α-renaming of bound variables in polymorphic types, supports structural equivalence ($[y/x]$ is the substitution that maps x to y and is the identity elsewhere and $FV(\mathtt{T})$ denotes the free variables of $\mathtt{T}$) :

$$\frac{\begin{array}{rcl}\mathtt{T}_0 & \sim & \mathtt{T'}_0\\ \mathtt{T}_1 & \sim & \mathtt{T'}_1\\ \mathtt{F} & \sim & \mathtt{F'}\end{array}}{(\mathtt{proc}\ \mathtt{F}\ (\mathtt{T}_0)\ \mathtt{T}_1) \sim (\mathtt{proc}\ \mathtt{F'}\ (\mathtt{T'}_0)\ \mathtt{T'}_1)}$$

$$\frac{\mathtt{I}' \notin FV(\mathtt{T})}{(\mathtt{poly}\ (\mathtt{I}\ \mathtt{K})\ \mathtt{T}) \sim (\mathtt{poly}\ (\mathtt{I}'\ \mathtt{K})\ [\mathtt{I}'/\mathtt{I}]\mathtt{T})}$$

but also includes the whole algebra of effects induced by the `maxeff` construct, namely Associativity, Commutativity, Unitary and Idempotence:

$$\begin{array}{c}(\mathtt{maxeff}\ \mathtt{F}_0\ (\mathtt{maxeff}\ \mathtt{F}_1\ \mathtt{F}_2)) \sim (\mathtt{maxeff}\ (\mathtt{maxeff}\ \mathtt{F}_0\ \mathtt{F}_1)\ \mathtt{F}_2) \\ (\mathtt{maxeff}\ \mathtt{F}_0\ \mathtt{F}_1) \sim (\mathtt{maxeff}\ \mathtt{F}_1\ \mathtt{F}_0) \\ (\mathtt{maxeff}\ \mathtt{F}\ \mathtt{pure}) \sim \mathtt{F} \\ (\mathtt{maxeff}\ \mathtt{F}\ \mathtt{F}) \sim \mathtt{F}\end{array}$$

Because of the algebraic properties of effects, the equivalence on procedure types requires reasoning over the effect algebra rules.

The kind rules for effects and types follow. The relation "has kind", noted "::", is used to denote the kind of any description in a given assignment. A is the kind assignment function that maps variables to their kind:

$$\frac{\mathtt{I} :: \mathtt{K} \in A}{A \vdash \mathtt{I} :: \mathtt{K}}$$

$$\frac{\begin{array}{c}A \vdash \mathtt{F} :: \mathtt{effect} \\ A \vdash \mathtt{T}_0 :: \mathtt{type} \\ A \vdash \mathtt{T}_1 :: \mathtt{type}\end{array}}{A \vdash (\mathtt{proc}\ \mathtt{F}\ (\mathtt{T}_0)\ \mathtt{T}_1) :: \mathtt{type}}$$

$$\frac{A[\mathtt{I} :: \mathtt{K}] \vdash \mathtt{T} :: \mathtt{type}}{A \vdash (\mathtt{poly}\ (\mathtt{I}\ \mathtt{K})\ \mathtt{T}) :: \mathtt{type}}$$

The type and effect rules for variable, lambda abstraction (with and without explicit type), application, polymorphic abstraction, projection and description equivalence follow. Just as ":" is used to denote the "has type" relation, "!" is used to denote the "has effect" relation. Effect and type information flow back and forth between the "has type" and "has effect" relations as can be seen by looking at the ($\rightarrow$IT) and ($\rightarrow$E) rules where the latent effect of procedures is respectively computed and used. A is extended to map variables to types:

$$(\text{var})\ \frac{\mathtt{I} : \mathtt{T} \in A}{A \vdash \mathtt{I} : \mathtt{T}\ !\ \mathtt{pure}}$$

$$(\rightarrow\text{IT})\ \frac{A[\mathtt{I} : \mathtt{T}] \vdash \mathtt{E} : \mathtt{T}'\ !\ \mathtt{F}'}{A \vdash (\mathtt{lambda}\ (\mathtt{I}\ \mathtt{T})\ \mathtt{E}) : (\mathtt{proc}\ \mathtt{F}'\ (\mathtt{T})\ \mathtt{T}')\ !\ \mathtt{pure}}$$

$$(\rightarrow\text{IM})\ \frac{A[\mathtt{I} : \mathtt{M}] \vdash \mathtt{E} : \mathtt{T}\ !\ \mathtt{F}}{A \vdash (\mathtt{lambda}\ (\mathtt{I})\ \mathtt{E}) : (\mathtt{proc}\ \mathtt{F}\ (\mathtt{M})\ \mathtt{T})\ !\ \mathtt{pure}}$$

$$(\rightarrow\text{E})\ \frac{\begin{array}{c}A \vdash \mathtt{E}_0 : (\mathtt{proc}\ \mathtt{F}\ (\mathtt{T}_0)\ \mathtt{T})\ !\ \mathtt{F}_0 \\ A \vdash \mathtt{E}_1 : \mathtt{T}_0\ !\ \mathtt{F}_1\end{array}}{A \vdash (\mathtt{E}_0\ \mathtt{E}_1) : \mathtt{T}\ !\ (\mathtt{maxeff}\ \mathtt{F}_0\ (\mathtt{maxeff}\ \mathtt{F}_1\ \mathtt{F}))}$$

$$(\text{polyI})\ \frac{A[\mathtt{I} :: \mathtt{K}] \vdash \mathtt{E} : \mathtt{T}\ !\ \mathtt{pure}}{A \vdash (\mathtt{plambda}\ (\mathtt{I}\ \mathtt{K})\ \mathtt{E}) : (\mathtt{poly}\ (\mathtt{I}\ \mathtt{K})\ \mathtt{T})\ !\ \mathtt{pure}}$$

$$(\text{polyE})\ \frac{\begin{array}{c}A \vdash \mathtt{E} : (\mathtt{poly}\ (\mathtt{I}\ \mathtt{K})\ \mathtt{T})\ !\ \mathtt{F} \\ A \vdash \mathtt{D} :: \mathtt{K}\end{array}}{A \vdash (\mathtt{proj}\ \mathtt{E}\ \mathtt{D}) : [\mathtt{D}/\mathtt{I}]\mathtt{T}\ !\ \mathtt{F}}$$

$$(\sim)\ \frac{\begin{array}{c}A \vdash \mathtt{E} : \mathtt{T}\ !\ \mathtt{F} \\ \mathtt{T} \sim \mathtt{T}' \\ \mathtt{F} \sim \mathtt{F}'\end{array}}{A \vdash \mathtt{E} : \mathtt{T}'\ !\ \mathtt{F}'}$$

4 Type and Effect Reconstruction

In this section we will describe how to reconstruct the types and effects of expressions in the language we have just presented. Before we launch into the technical details of reconstruction, we will first consider an example that will motivate a major design decision. Assume for a moment that the following effect equation describes the latent effect e of a procedure: `write` $\sim$ (`maxeff` e `write`). Now e can be either be `pure` or `write`. We can now see that a type that contains an effect can have multiple possible forms, and there is no simple syntactic way to describe this diversity. Here is a procedure whose argument in fact has latent effect e and thus has two different types[1]:

```
(lambda (f)
  (lambda (g (proc write (bool) bool))
    (if #t
        g
        (lambda (x) (begin (f x) (g x))))))
```

Thus in order to handle the algebraic properties of KFX we must generalize the idea of *principal typing* [DM82] beyond syntactic substitution on *type schemes* to include substitution that observes algebraic constraints. This is because syntactic type schemes cannot describe the most general type of an expression in our system.

4.1 Algebraic Unification

We will now describe a unification algorithm that can handle algebraic constraints. Robinson's unification algorithm is used in syntactic reconstruction to enforce type constraints between type expressions by computing a substitution that maps one type to the other. It also ensures that the most general type (modulo substitution) is obtained after type reconstruction. However, the effect constraints that arise in an effect system cannot be enforced by this simple unification technique.

The unification algorithm `U` that follows computes both a *substitution* and a *constraint set*. We use a substitution to describe constraints on types, while we use a constraint set to describe constraints on effects. A substitution maps unification variables to types in `M`, and a constraint set contains pairs of effects, where the left and right element of each pair must be the same effect. When `U` is applied to two types $\mathtt{T}_1$ and $\mathtt{T}_2$, it returns a pair (S, C) where S is a substitution and C a constraint set. In our description ϕ is the empty set, ν is a unification variable, `N` is a constant, and [] is the identity substitution.

```
U( T1, T2 ) =
T1 = ν =>
        if T2 ∈ M then ([T2/T1], φ) else fail
T2 = ν =>
        if T1 ∈ M then ([T1/T2], φ) else fail
T1 = I =>
        if T2 = I then ([], φ) else fail
T1 = (proc F1 (T11) T12) =>
        if T2 = (proc F2 (T21) T22) then
                let (S1, C1) = U( T11, T21 )
                let (S2, C2) = U( S1T12, S1T22 )
                (S2S1, C1 ∪ C2 ∪ {(F1, F2)})
        else fail
```

[1] begin and if are the usual sequence and alternative special forms.

```
T1 = (poly (I1 K) T'1) =>
        if T2 = (poly (I2 K) T'2) then
                U( [N/I1]T'1, [N/I2]T'2 )
                        where N is fresh
        else fail
else fail
```

In U two polymorphic types unify iff they are abstracted over the same kind and their bodies unify when the abstraction variable has been substituted by the same newly created *constant* description. This definition corresponds to an extensional view of equality of polymorphic types:

$$(\texttt{poly}\ (\texttt{I}_1\ \texttt{K})\ \texttt{T}_1) \sim (\texttt{poly}\ (\texttt{I}_2\ \texttt{K})\ \texttt{T}_2) \text{ iff } \forall \texttt{D}.\ [\texttt{D}/\texttt{I}_1]\texttt{T}_1 \sim [\texttt{D}/\texttt{I}_2]\texttt{T}_2$$

Using a new constant is a way to ensure that the two polymorphic types are the same, whatever they might be projected upon later.

Two type expressions unify if and only if the constraint set returned by U is satisfiable. Satisfiability of a constraint set can be checked, e.g. by using an ACUI-unification procedure. When effect constraints are checked is a trade-off between providing users with early notification of errors (eager checking) vs. optimum performance of the reconstruction system (lazy checking).

Definition 1 (Satisfiability) *A constraint set* $C = \{(\texttt{F}_i, \texttt{F}'_i)\}$ *is* satisfied *under the model* m *(that maps unification variables of* C *onto* F*), written* $m \models C$*, iff for all* i*,* $m\texttt{F}_i \sim m\texttt{F}'_i$.

Models, being ground substitutions on effects, are straightforwardly extended to types by induction.

Theorem 1 (Correctness) *Let (S, C) be* U($\texttt{T}_1$, $\texttt{T}_2$) *and* m *be a model:*

$$(m \models C) \Longrightarrow m(S\texttt{T}_1) \sim m(S\texttt{T}_2)$$

Proof. By induction on type expressions, using the fact that $m\texttt{T} \sim m\texttt{T}' \Longrightarrow m(S\texttt{T}) \sim m(S\texttt{T}')$. □

4.2 The Algebraic Reconstruction Algorithm

Our algebraic reconstruction algorithm R uses the unification procedure U to compute the type and effect of an expression E. The two input parameters to the reconstruction algorithm R are a type environment A that binds identifiers to types or kinds, and an expression E. We assume that the expression E has been alpha-renamed so there are no identifier name conflicts. The reconstruction algorithm R returns a quadruple $(\texttt{T}, \texttt{F}, C, S)$ where T is the type of E in the environment A and F its effect. Both T and F are subject to the constraint set C.

```
R( A, E ) = case E in
I =>
        if [I : T] ∈ A then (T, pure, φ, []) else fail
(lambda (I T) E) =>
        let (T',F',C,S) = R( A[I : T], E )
        ((proc F' (T) T'), pure, C, S)
(lambda (I) E)
        let (T,F,C,S) = R( A[I : ν], E )
                where ν is fresh
        ((proc F (Sν) T), pure, C, S)
(E0 E1) =>
        let (T0,F0,C0,S0) = R( A, E0 )
        let (T1,F1,C1,S1) = R( S0A, E1 )
        let (S,C) = U( S1T0, (proc ν1 (T1) ν2) )
                where νi are fresh
        let F' = (maxeff F0 (maxeff F1 ν1))
        (Sν2, F', C0 ∪ C1 ∪ C, SS1S0)
(plambda (I K) E) =>
        let (T,F,C,S) = R( A[I :: K], E )
        let C' = (C ∪ {(F, pure)})
        let {Ii} = FV(C') − FV(S(A[I :: K]))
        ((poly (I K) T), pure, [N/I][νi/Ii]C' ∪ C', S)
                where N, νi are fresh
(proj E D) =>
        let (T',F,C,S) = R( A, E )
        if T' = (poly (I K) T) then
                ([D/I]T, F, [D/I]C, S)
        else fail
else fail
```

where FV is extended to constraint sets and environments, with $FV[\texttt{I} : \texttt{T}] = FV(\texttt{T})$ and $FV[\texttt{I} :: \texttt{K}] = \{\texttt{I}\}$.

In the case of the polymorphic abstraction operator plambda, a compound constraint set is constructed. The second part C' of the constraint set propagates the current effect bindings. The purpose of the first part is twofold. $[\texttt{N}/\texttt{I}][\nu_i/\texttt{I}_i]C$ ensures that the constraint set is actually polymorphic over the abstraction variable, using the same idea as in the unification procedure (i.e., by simulating a projection upon a new *constant* description N). $[\texttt{N}/\texttt{I}][\nu_i/\texttt{I}_i]\{(\texttt{F}, \texttt{pure})\}$ arranges for the body of the expression to always pure. New unification variables ν_i are introduced to decouple the check for polymorphism from the propagation of constraint bindings.

5 Correctness Theorems

The following theorems show the termination, soundness and completeness of the type reconstruction algorithm with respect to the typing rules of KFX and study the complexity of R.

5.1 Termination

Theorem 2 (Termination) *R(* $A, \texttt{E}$ *) terminates.*

Proof. R works by induction on the structure of expressions, which are of finite height. □

5.2 Soundness

Theorem 3 (Soundness) *Let* $(\texttt{T}, \texttt{F}, C, S) = \texttt{R}(A, \texttt{E})$ *and* m *be a model:*

$$(m \models C) \Longrightarrow mSA \vdash \texttt{E} : m\texttt{T} \mathbin{!} m\texttt{F}$$

Proof. By case analysis of the typing rules and induction on the structure of expressions. We describe the interesting cases of proj(which takes advantage of the compound constraint created by plambda) and application (which uses the unification theorem).

For proj, suppose that:

$$([\texttt{D}/\texttt{I}]\texttt{T}, \texttt{F}, [\texttt{D}/\texttt{I}]C, S) = \texttt{R}(A, (\texttt{proj E D}))$$

and $m \models [\mathtt{D}/\mathtt{I}]C$. We want to prove:

$$mSA \vdash (\mathtt{proj\ E\ D}) \ : \ m([\mathtt{D}/\mathtt{I}]\mathtt{T}) \ ! \ m\mathtt{F} \qquad (1)$$

Since m is defined only on unification variables: $m([\mathtt{D}/\mathtt{I}]\mathtt{T}) = [\mathtt{D}/\mathtt{I}](m\mathtt{T})$. Thus, for (1) to be true, we need to prove, according to the typing rule (polyE):

$$mSA \vdash \mathtt{E} \ : \ (\mathtt{poly}\ (\mathtt{I}\ \mathtt{K})\ m\mathtt{T}) \ ! \ m\mathtt{F}$$

Since, by definition of $\mathtt{R}$,

$$((\mathtt{poly}\ (\mathtt{I}\ \mathtt{K})\ \mathtt{T}), \mathtt{F}, C, S) = \mathtt{R}(A, \mathtt{E})$$

then by induction:

$$(m' \models C) \Longrightarrow m'SA \vdash \mathtt{E} \ : \ m'(\mathtt{poly}\ (\mathtt{I}\ \mathtt{K})\ \mathtt{T}) \ ! \ m'\mathtt{F}$$

Since, as before,

$$m'(\mathtt{poly}\ (\mathtt{I}\ \mathtt{K})\ \mathtt{T}) = (\mathtt{poly}\ (\mathtt{I}\ \mathtt{K})\ m'\mathtt{T})$$

we just have to show that $m \models C$. But, by hypothesis, $m \models [\mathtt{D}/\mathtt{I}]C$. Since C is the constraint associated to a polymorphic type, it comes from either 1) an explicit `plambda`, in which case we took care to check that C is satisfiable for any $\mathtt{I}$, or 2) from a formal argument that has a polymorphic type, in which case C is equivalent to ϕ. Thus, $m \models C$.

For application, suppose that $\mathtt{R}(A, (\mathtt{E}_0\ \mathtt{E}_1))$ is equal to

$$(S\nu_2, (\mathtt{maxeff}\ \mathtt{F}_0\ (\mathtt{maxeff}\ \mathtt{F}_1\ \nu_1)), C_0 \cup C_1 \cup C, SS_1S_0)$$

and $m \models C_0 \cup C_1 \cup C$. We want to prove that in the environment mSS_1S_0A:

$$(\mathtt{E}_0\ \mathtt{E}_1) \ : \ m(S\nu_2) \ ! \ m(\mathtt{maxeff}\ \mathtt{F}_0\ (\mathtt{maxeff}\ \mathtt{F}_1\ \nu_1)) \qquad (2)$$

However, by using structural induction, we can apply SS_1 to $mS_0A \vdash \mathtt{E}_0 \ : \ m\mathtt{T}_0 \ ! \ m\mathtt{F}_0$ and apply S to $mS_1S_0A \vdash \mathtt{E}_1 \ : \ m\mathtt{T}_1 \ ! \ m\mathtt{F}_1$, since $m \models C_0$ and $m \models C_1$. By the unification lemma, since $m \models C$:

$$mSS_1\mathtt{T}_0 \sim (\mathtt{proc}\ m\nu_1\ (mS\mathtt{T}_1)\ mS\nu_2)$$

To complete the proof of (2), we thus only have to use the typing rule ($\rightarrow$E). $\square$

5.3 Completeness

Theorem 4 (Completeness) *If $mSA \vdash \mathtt{E} \ : \ \mathtt{T} \ ! \ \mathtt{F}$, there exist $(\mathtt{T}', \mathtt{F}', C', S')$, a model m' and a substitution P' such that:*

$$\begin{cases} (\mathtt{T}', \mathtt{F}', C', S') = \mathtt{R}(A, \mathtt{E}) \\ m'P'S'A = mSA, \text{ on } FV(\mathtt{E}) \\ m' \models C' \\ \mathtt{T} \sim m'P'\mathtt{T}' \\ \mathtt{F} \sim m'\mathtt{F}' \end{cases}$$

Proof. As before, by induction on the typing derivation and induction on the size of terms. We will prove the case of lambda abstraction on $\mathtt{M}$ types, which shows how to deal with unification variables, and polymorphic abstraction, which checks for pureness of expressions.

For `lambda` on $\mathtt{M}$ types, assume that

$$mSA \vdash (\mathtt{lambda}\ (\mathtt{I})\ \mathtt{E}) \ : \ (\mathtt{proc}\ \mathtt{F}\ (\mathtt{M})\ \mathtt{T}) \ ! \ \mathtt{pure}$$

By ($\rightarrow$IM), this requires

$$mSA[\mathtt{I} : \mathtt{M}] \vdash \mathtt{E} \ : \ \mathtt{T} \ ! \ \mathtt{F}$$

which is equivalent to

$$m(S[\mathtt{M}/\nu])(A[\mathtt{I} : \nu]) \vdash \mathtt{E} \ : \ \mathtt{T} \ ! \ \mathtt{F}$$

where ν is fresh. By induction, there exist $(\mathtt{T}'', \mathtt{F}'', C'', S'')$, m'' and P'' that verify the theorem for $\mathtt{E}$ in the environment $m(S[\mathtt{M}/\nu])(A[\mathtt{I} : \nu])$. In particular:

$$m''P''S''(A[\mathtt{I} : \nu]) = m(S[\mathtt{M}/\nu])(A[\mathtt{I} : \nu])$$

To prove the theorem, pick:

$$\begin{array}{l} \mathtt{T}' = (\mathtt{proc}\ \mathtt{F}''\ (S''\nu)\ \mathtt{T}'') \\ \mathtt{F}' = \mathtt{pure} \\ C' = C'' \\ S' = S'' \\ m' = m'' \\ P' = P'' \end{array}$$

The theorem is verified since $m'P'S'A = m''P''S''A$ on the free variables of the lambda expression, $m' \models C' = m'' \models C''$ and

$$\begin{array}{rcl} m'P'\mathtt{T}' & = & (\mathtt{proc}\ m''\mathtt{F}''\ (m''P''S''\nu)\ m''P''\mathtt{T}'') \\ & = & (\mathtt{proc}\ \mathtt{F}\ (\mathtt{M})\ \mathtt{T}) \end{array}$$

For `plambda`, assume that

$$mSA \vdash (\mathtt{plambda}\ (\mathtt{I}\ \mathtt{K})\ \mathtt{E}) \ : \ (\mathtt{poly}\ (\mathtt{I}\ \mathtt{K})\ \mathtt{T}) \ ! \ \mathtt{pure}$$

By (polyI), this requires

$$(mSA)[\mathtt{I} :: \mathtt{K}] \vdash \mathtt{E} \ : \ \mathtt{T} \ ! \ \mathtt{pure}$$

which is equivalent to

$$mS(A[\mathtt{I} :: \mathtt{K}]) \vdash \mathtt{E} \ : \ \mathtt{T} \ ! \ \mathtt{pure}$$

By induction, there exist $(\mathtt{T}_0, \mathtt{F}_0, C_0, S_0)$, m_0 and P_0 that verify the theorem for $\mathtt{E}$ in the environment $mSA[\mathtt{I} :: \mathtt{K}]$. Since we know that the polymorphic `plambda` expression has a type, it also has one when projected upon any description. Thus, for any constant $\mathtt{N}$:

$$mSA \vdash [\mathtt{N}/\mathtt{I}]\mathtt{E} \ : \ [\mathtt{N}/\mathtt{I}]\mathtt{T} \ ! \ \mathtt{pure}$$

By induction, there exist $\mathtt{T}_1, \mathtt{F}_1, C_1, S_1$, m_1 and P_1 that verify the theorem for $[\mathtt{N}/\mathtt{I}]\mathtt{E}$ in the environment mSA. To prove the theorem, pick:

$$\begin{array}{l} \mathtt{T}' = (\mathtt{poly}\ (\mathtt{I}\ \mathtt{K})\ \mathtt{T}_0) \\ \mathtt{F}' = \mathtt{pure} \\ C' = [\mathtt{N}/\mathtt{I}][\nu_i/\mathtt{I}_i](C_0 \cup \{(\mathtt{F}_0, \mathtt{pure})\}) \cup (C_0 \cup \{(\mathtt{F}_0, \mathtt{pure})\}) \\ S' = S_0 \\ m' = [m_1\mathtt{I}_i/\nu_i]m_0 \\ P' = P_0 \end{array}$$

where $\{\mathtt{I}_i\} = (FV(\mathtt{F}_0) \cup FV(C_0)) - FV(S_0(A[\mathtt{I} :: \mathtt{K}]))$ and ν_i are fresh. The theorem is trivially verified, except for the verification of constraints. Since $m_0 \models C_0$ and $m_0\mathtt{F}_0 \sim \mathtt{pure}$, then $m' \models C_0 \cup \{(\mathtt{F}_0, \mathtt{pure})\}$. For the first term C'_1 of C' defined by

$$C'_1 = [\mathtt{N}/\mathtt{I}][\nu_i/\mathtt{I}_i](C_0 \cup \{(\mathtt{F}_0, \mathtt{pure})\})$$

$m' \models C'_1$ can be successively rewritten as:

$$\begin{array}{l} m' \models C'_1 = \\ [m_1 \mathtt{I}_i/\nu_i]m_0 \models [\mathtt{N}/\mathtt{I}][\nu_i/\mathtt{I}_i](C_0 \cup \{(\mathtt{F}_0, \mathtt{pure})\}) = \\ m_0 \models [\mathtt{N}/\mathtt{I}][m_1 \mathtt{I}_i/\mathtt{I}_i](C_0 \cup \{(\mathtt{F}_0, \mathtt{pure})\}) = \\ (m_1 \models C_1) \wedge (m_1 \mathtt{F}_1 \sim \mathtt{pure}) \end{array}$$

since the set of free variables $\{\mathtt{I}_i\}$ of $\mathtt{F}_0$ and C_0 is the same (more precisely, is isomorphic to) the one of $\mathtt{F}_1$ and C_1. □

5.4 Complexity

Type checking a program now amounts to running `R` in an initial environment that binds predefined variables (such as `cons`, `set!` or `bool`) to their types or kinds. Deciding whether the program is type-safe is equivalent to finding a model m that satisfies the constraint set C.

Theorem 5 (Decidability) *The type and effect reconstruction problem for KFX is decidable.*

Proof. KFX type reconstruction depends on the decidability of the satisfiability problem for constraint sets. Since `maxeff` is an ACUI operator, constraint satisfiability can be decided by an ACUI-unification algorithm [LC89]. □

ACUI-unification is an NP-complete problem [KN86]. This untractability is rooted into the presence of free function symbols in terms. The definition of KFX does not support free effect functions, and thus constraint set satisfaction is a function-free ACUI unification problem. The following theorem, due to McAllester and Blair, implies that the satisfiability problem for constraint set satisfiability is polynomial time decidable.

Definition 2 (Function-free ACUI Unification) *An ACUI unification problem will be called* function-free *if the only function symbol in the problem is the ACUI operator.*

Theorem 6 (Complexity [MB90]) *Function-free ACUI unification is polynomial time decidable.*

Proof. Each equation in a function-free ACUI unification problem has the form:

$$(\mathtt{maxeff}\ X_1\ ...\ X_m) \sim (\mathtt{maxeff}\ Y_1\ ...\ Y_n)$$

where `maxeff` is the ACUI operator and X_i and Y_j are either variables or constants. Without loss of generality, we will consider the special case of effect equations, and we have extended the `maxeff` constructor to allow a variable number of arguments in which duplicates and `pure` are eliminated.

If we consider the universe of N effect constants E_1 to E_N drawn from the constraint set, we can rewrite an equation as the following set of N double implications:

$$\begin{array}{l} (E_1 \in X_1 \vee ... \vee E_1 \in X_m) \Leftrightarrow (E_1 \in Y_1 \vee ... \vee E_1 \in Y_n) \\ ... \\ (E_N \in X_1 \vee ... \vee E_N \in X_m) \Leftrightarrow (E_N \in Y_1 \vee ... \vee E_N \in Y_n) \end{array}$$

This set of double implications is identical to the original equation because they will be true if and only if the set of effects on both sides of the original equation were equal. We now rewrite each double implication, say for E_i, into the following set of single implications:

$$\begin{array}{l} E_i \in X_1 \Rightarrow (E_i \in Y_1 \vee ... \vee E_i \in Y_n) \\ ... \\ E_i \in X_m \Rightarrow (E_i \in Y_1 \vee ... \vee E_i \in Y_n) \\ E_i \in Y_1 \Rightarrow (E_i \in X_1 \vee ... \vee E_i \in X_m) \\ ... \\ E_i \in Y_n \Rightarrow (E_i \in X_1 \vee ... \vee E_i \in X_m) \end{array}$$

This rewriting transforms a single constraint into $(m+n)N$ implications. Taking the contrapositive of each implication, we have:

$$E_i \notin X_j \Leftarrow (E_i \notin Y_1 \wedge ... \wedge E_i \notin Y_n)$$

and similarly for Ys. Thus if we start with k constraints with sides having at most n terms, we will have $O(knN)$ single sided implications. These implications are propositional Horn clauses and thus their satisfiability can be determined in linear time [DG84] starting with the following polynomial number of axioms:

$$E_i \notin E_j \text{ if } i \text{ is different from } j$$

□

KFX type reconstruction is not necessarily polynomial in the size of the input program, even though the satisfiability of constraint sets can be decided in polynomial time. The reason is that the constraint set built by `R` doubles in size each time a `plambda` construct is encountered. Thus constraint set size is exponential in the nesting level of `plambda` constructs. We expect this not to be a major problem in practice since `plambda` nesting is usually shallow.

6 Extension to Generic Types

It is possible to add to our system the notion of *generic* types used in structural reconstruction systems for type checking `let` constructs.

6.1 Definitions

The language $\mathrm{KFX}_{\mathtt{let}}$ adds the `let` binding construct to KFX:

$$\mathtt{E} \ += \ (\mathtt{let}\ (\mathtt{I}\ \mathtt{E}_0)\ \mathtt{E}_1) \quad \text{local binding}$$

We will create generic types for a `let` variable that is bound to a pure expression. Following [T87], we will only create a generic type for a variable that is bound to a non-expansive expression:

Definition 3 (Expansive) *An expression* `E` *is* expansive *iff* `E` *is not a variable, a lambda expression or a plambda expression.*

The astute reader might question why we do not use our own effect information to determine which bindings to generalize. The reason is that there is no guarantee that we can solve the effect equations that describe the effect of binding expressions at the time we examine a `let`. An implementation could use backtracking to solve this problem.

Our typing rule for `let` does not compute a type scheme in order to avoid the difficulty of describing in the rule system how to adapt type schemes to include effect constraints.

However, our reconstruction algorithm for `let` in the next section does in fact compute an algebraic type scheme. We later prove the consistency of this approach.

The typing rule system for $\mathrm{KFX}_{\mathtt{let}}$ includes all the rules used in KFX, plus the (Glet)[2] and (let) rules:

$$\text{(Glet)}\quad \frac{\begin{array}{c} A \vdash \mathtt{E}_0 : \mathtt{T}_0 \,!\, \mathtt{pure} \\ A \vdash [\mathtt{E}_0/\mathtt{I}]\mathtt{E}_1 : \mathtt{T}_1 \,!\, \mathtt{F}_1 \\ \mathtt{E}_0 \text{ is not expansive} \end{array}}{A \vdash (\mathtt{let}\ (\mathtt{I}\ \mathtt{E}_0)\ \mathtt{E}_1) : \mathtt{T}_1 \,!\, \mathtt{F}_1}$$

$$\text{(let)}\quad \frac{\begin{array}{c} A \vdash \mathtt{E}_0 : \mathtt{T}_0 \,!\, \mathtt{F}_0 \\ A[\mathtt{I} : \mathtt{T}_0] \vdash \mathtt{E}_1 : \mathtt{T}_1 \,!\, \mathtt{F}_1 \\ \mathtt{E}_0 \text{ is expansive} \end{array}}{A \vdash (\mathtt{let}\ (\mathtt{I}\ \mathtt{E}_0)\ \mathtt{E}_1) : \mathtt{T}_1 \,!\, (\mathtt{maxeff}\ \mathtt{F}_0\ \mathtt{F}_1)}$$

6.2 Generic Reconstruction

We describe the type of a let bound variable in our reconstruction algorithm with an *algebraic type scheme.* An algebraic type scheme packages together a syntactic type scheme and a constraint set. As we have already seen, we can not use syntactic type schemes to represent the most general type of an expression in our system. This is because our reconstruction algorithm returns a type that is subject to an effect constraint set. Thus there is no way to describe the possible types of an expression via syntactic substitutions.

Definition 4 (Algebraic Type Schemes) *An algebraic type scheme is noted* $(\forall\ \{\mathtt{I}_i\}\ (\mathtt{T}, C))$. *By convention,* $\mathtt{T} \sim (\forall\ \phi\ (\mathtt{T}, \phi))$.

The A environment maps identifiers to algebraic type schemes. The previous definition of `R` can then be expanded in the following way:

```
R( A, E ) = ...
I =>
        if [I : (∀ {I_i} (T, C))] ∈ A then
                ([ν_i/I_i]T, pure, [ν_i/I_i]C, [])
                        where ν_i are fresh
        else fail
(let (I E_0) E_1) and E_0 expansive =>
        let (T_0, F_0, C_0, S_0) = R( A, E_0 )
        let (T_1, F_1, C_1, S_1) = R( S_0(A[I : T_0]), E_1 )
        (T_1, (maxeff F_0 F_1), C_0 ∪ C_1, S_1S_0)
(let (I E_0) E_1) and E_0 not expansive =>
        let (T_0, F_0, C_0, S_0) = R( A, E_0 )
        let {I_i} = (FV(T_0) ∪ FV(C_0)) − FV(S_0A)
        let A' = A[I : (∀ {I_i} (T_0, C_0))]
        let (T_1, F_1, C_1, S_1) = R( S_0A', E_1 )
        (T_1, F_1, C_0 ∪ C_1, S_1S_0)
```

The most interesting part deals with the non-expansive case of a `let` expression. The `let` body is reconstructed in an environment that binds `I` to an algebraic type scheme packaged by collecting $\mathtt{T}_0$ and C_0. This type scheme is abstracted over all the free variables that do not occur in A. The type scheme is instantiated when `I` is used, thus mimicking the syntactic substitution of $\mathtt{E}_0$ inside $\mathtt{E}_1$.

[2] One can show [M89] that, in ML, this is equivalent to the usual type-scheme based approach.

6.3 Correctness Theorems

Theorem 7 ($\mathrm{KFX}_{\mathtt{let}}$) *All the previous theorems about KFX stated in Section 5 extend trivially to* $\mathrm{KFX}_{\mathtt{let}}$.

Proof. The soundness and completeness properties of `R` w.r.t. the expansive (let) rule can be shown by a straightforward structural induction.

The reconstruction algorithm for the non-expansive `let` implements the textual substitution required by the (Glet) rule. This is easily seen by noticing that the type and constraint set of a non-expansive expression `E` only depend on the free variables of `E`, i.e. of the environment A. Algebraic type schemes simply *cache* the type and effect constraint that would have to be recomputed each time $\mathtt{E}_0$ appeared in the substituted body. □

7 Conclusion

We have presented the first algorithm for reconstructing the types and effects of expressions in the presence of first class procedures in a polymorphic typed language. Effects are more complicated to reconstruct than types because the algebra of effects induces complex constraints on both effects and types that must be solved during reconstruction. Algebraic reconstruction was introduced to deal with these constraints. We proved that it is sound and complete and we studied its complexity.

It is likely that algebraic reconstruction will find application in systems without effects. For example, algebraic reconstruction could prove useful for type systems with algebraic properties; a simple example is the type algebra of records [W87].

Algebraic reconstruction is being implemented inside the FX compiler under development at MIT in order to assess the practicality of our approach.

Acknowledgments

We thank David McAllester and Michael Blair, who discovered the polynomial reduction of constraint set satisfiability to satisfiability of propositional Horn clauses. We also thank Hans Boehm, Vincent Dornic, James O'Toole, Mark Sheldon and Jean-Pierre Talpin for their insightful comments.

References

[B85] Boehm, H. J. Partial Polymorphic Type Inference is Undecidable. In *Proceedings of the 26th FOCS Symposium.* IEEE, 1985.

[C84] Cardelli, L. A Semantics of Multiple Inheritance. In *Proceedings of the Inter. Symp. on Semantics of Data Types*, LNCS 173, Springer-Verlag, 1984.

[DG84] Dowling, W. F., and Gallier, J. H. Linear-time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *J. Logic Programming (3)*, 1984.

[DM82] Damas, L., and Milner, R. Principal Type Schemes for Functional Programs. In *Proceedings of the 9th Annual ACM Conference on Principles of Programming Languages.* ACM, New York, 1982.

[FLO83] Fortune, S., Leivant, D., and O'Donnell, M. Simple and Second-Order Type Structures. *JACM*, Jan. 1983.

[GJLS87] Gifford, D. K., Jouvelot, P., Lucassen, J. M., and Sheldon, M. A. *The FX-87 Reference Manual.* Massachusetts Institute of Technology, LCS/TR-407, 1987.

[GR88] Giannini, P., and Ronchi Della Rocca, S. Characterization of Typings in Polymorphic Type Discipline. In *Proceedings of the IEEE LICS Symposium.* IEEE, 1988.

[HG88] Hammel, R. T., and Gifford, D. K. *FX-87 Performance Measurements: Dataflow Implementation.* Massachusetts Institute of Technology, LCS/TR-421, 1988.

[J89] Jategaonkar, L. A., *ML With Extended Pattern Matching and Subtypes.* Massachusetts Institute of Technology, LCS/TR-468, August 1989.

[JG89a] Jouvelot, P., and Gifford, D. K. *Communication Effects for Message-Based Concurrency.* Massachusetts Institute of Technology, LCS/TM-386, February 1989.

[JG89b] Jouvelot, P., and Gifford, D. K. Reasoning about Continuations with Control Effects. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, New York, 1989.

[KN86] Kapur, D., and Narendran, P. NP-Completeness of the Set Unification and Matching Problems. In *Proceedings of the 8th Inter. Conference on Automated Deduction.* LNCS 230, Springer-Verlag, 1986.

[LC89] Lincoln, P., and Christian, J. Adventures in Associative-Commutative Unification. *J. of Symbolic Computation (8)*, 1989.

[LG88] Lucassen, J. M., and Gifford, D. K. Polymorphic Effect Systems. In *Proceedings of the 15th Annual ACM Conference on Principles of Programming Languages.* ACM, New York, 1988, pp. 47-57.

[M68] Morris, J. H. *Lambda-Calculus Models of Programming Languages*, Massachusetts Institute of Technology, MAC-TR-57, 1968.

[M78] Milner, R. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, vol. 17, 1978, pp. 349-375.

[M89] Mitchell, J. *Type Systems for Programming Languages.* Stanford Rep. CS-89-1277, 1989.

[M90] Mairson, H. G. Deciding ML Typability is Complete for Deterministic Exponential Time. In *Proceedings of the ACM Conference on Principles of Programming Languages.* ACM, New York, 1990.

[MB90] McAllester, D., and Blair, M. Private communication. July 1990.

[OG89] O'Toole, J., and Gifford, D. K. Polymorphic Type Reconstruction. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, New York, 1989.

[P88] Pfenning, F. Partial Polymorphic Type Inference and Higher-Order Unification. In *Proceedings of the ACM Conference on Lisp and Functional Programming.* ACM, New York, 1988.

[R65] Robinson, J. A. A machine-oriented logic based on the resolution principle. *JACM* 12,1 (1965), pp. 23-41.

[T87] Tofte, M. *Operational Semantics and Polymorphic Type Inference.* Ph. D. Thesis, Edinburgh University, 1987.

[W87] Wand, M. Complete Type Inference for Simple Objects. In *Proceedings of the IEEE LICS Symposium.* IEEE, 1987.

[W88] Wand, M. Corrigendum: Complete Type Inference for Simple Objects. In *Proceedings of the IEEE LICS Symposium.* IEEE, 1988.

[W89] Wand, M. Type Inference for Record Concatenation and Multiple Inheritance. In *Proceedings of the IEEE LICS Symposium.* IEEE, 1989.