Algebraic Reconstruction of Types and Effects

Pierre Jouvelot\textsuperscript{1,2}
David K. Gifford\textsuperscript{2}
Ecole des Mines de Paris\textsuperscript{1}
MIT Laboratory for Computer Science\textsuperscript{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, \texttt{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 cannot 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 i f f 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 that cannot 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 [LGS88], 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 [K86], we expect most cases to be tractable, in the same manner ML’s inference system has proved to be useful while being EXPTIME-complete [MB90]. 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:

\[
K ::= \text{effect} \mid \text{type}
\]

\[
D ::= F \mid T
\]

\[
F ::= \text{maxeff } F_0 F_1
\]

(combination of effects)

\[
T ::= M
\]

(procedure)

\[
M ::= I
\]

(poly (I) (K) T)

(polymorphic lambda)

\[
M ::= I
\]

(pro I M)

(projection)

\[
E ::= I
\]

(\lambda (I) T) E

(typed lambda)

\[
I ::= N
\]

(constant)

\[
V ::= \nu
\]

(user variable)

\[
\nu ::= \nu
\]

(unification variable)

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

KFX descriptions have a much richer algebra than normal type expressions. The conversion equivalence relation \( \sim \) between descriptions, beside the usual \( \alpha \)-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 \( FV(T) \) denotes the free variables of \( T \):

\[
\begin{align*}
T_0 & \sim T_0' \\
T_1 & \sim T_1' \\
F & \sim F'
\end{align*}
\]

\[
(\text{proc } F (T_0) T_1) \sim (\text{proc } F' (T_0) T_1)
\]
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 ≈ (maxeff e write). Now e can be either 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:

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

Thus also 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 T₁ and T₂, 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 [I] is the identity substitution.

\[
\begin{align*}
\text{U( }& T₁, T₂ \text{ ) = } \\
& T₁ = ν \iff \\
& \text{if } T₂ \in M \text{ then } ([T₂/T₁], φ) \text{ else fail } \\
& \text{if } T₁ \in M \text{ then } ([T₁/T₂], φ) \text{ else fail } \\
& T₁ = I \iff \\
& \text{if } T₂ = I \text{ then } ([I], φ) \text{ else fail } \\
& T₁ = (\text{proc } F₁ (T₁₁) T₁₂) \iff \\
& \text{if } T₂ = (\text{proc } F₂ (T₂₁) T₂₂) \text{ then } \\
& \text{let } (S₁, C₁) = U(T₁₁, T₂₁) \\
& \text{let } (S₂, C₂) = U(S₁T₁₂, S₁T₂₂) \\
& (S₂S₁, C₁ ∪ C₂ ∪ \{(F₁, F₂)\}) \text{ else fail }
\end{align*}
\]

1. begin and if are the usual sequence and alternative special forms.
\[ T_1 = \text{(poly } \langle I_1 \ K \rangle \ T'_1 \rangle \Rightarrow \]
\[ \text{if } T_2 = \text{(poly } \langle I_2 \ K \rangle \ T'_2 \rangle \text{ then } \]
\[ U( \langle N/I_1 \rangle T'_1, \langle N/I_2 \rangle T'_2 ) \]
\[ \text{where } N \text{ is fresh} \]
\[ \text{else fail} \]
\[ \text{else fail} \]

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:

\[ \text{(poly } \langle I_1 \ K \rangle T_1 \rangle \sim \text{(poly } \langle I_2 \ K \rangle T_2 \rangle \text{ iff } \forall D \langle D/I_1 \rangle T_1 \sim \langle D/I_2 \rangle T_2 \]

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 ACU-unification procedure. When effect constraints are checked the 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 \( C = \{ (F_i, F'_i) \} \) is satisfied under the model \( m \) (that maps unification variables of \( C \) onto \( F \)), written \( m \models C \), iff for all \( i \), \( m F_i \sim m F'_i \).

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

**Theorem 1 (Correctness)** Let \( (S, C) \) be \( U( T_1, T_2 ) \)
and \( m \) be a model:

\[ (m \models C) \Rightarrow m(ST_1) \sim m(ST_2) \]

**Proof.** By induction on type expressions, using the fact that \( mT \sim mT' \Rightarrow m(ST) \sim m(ST') \). \( \square \)

### 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 \( (T, F, C, S) \) 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 \( C \).

\[
 R( A, E ) = \text{case } E \text{ in}
 I \Rightarrow
 (\lambda \text{ if } [I : T] \in A \text{ then } (T, \text{pure, } \phi, []) \text{ else fail})
 (\lambda \text{ let } (T', F', C, S) = R( A[I : T], E )
 (\langle \text{proc } F' (T') T \rangle, \text{pure, C, S}))
 (\lambda \text{ let } (T, F, C, S) = R( A[I : \nu], E )
 \text{where } \nu \text{ is fresh}
 (\langle \text{proc } F (S\nu) T \rangle, \text{pure, C, S}))
 (E_0, E_1) \Rightarrow
 let (T_0, F_0, C_0, S_0) = R( A, E_0 )
 let (T_1, F_1, C_1, S_1) = R( A, E_1 )
 let (S, C) = U( S_1 T_0, \langle \text{proc } \nu_1 (T_1) \nu_2 \rangle )
 \text{where } \nu_i \text{ are fresh}
 let F' = \text{(maxeff F_0 (maxeff F_1 \nu_1))}
 (S_{\nu_2} F', C_0 \cup C_1 \cup C, S S_0)
 (\lambda \text{ (lambda } (I K) E \Rightarrow
 let (T, F, C, S) = R( A[I : K], E )
 let C' = (C \cup \{(F, \text{pure})\})
 let [I_1] = F'V(C') - F'V(\langle S(A[I :: K]) \rangle)
 (\langle \text{poly } (I K) T \rangle, \text{pure, } [N/I]\nu_1, [I_1][C' \cup C', S])
 \text{where } N, \nu_i \text{ are fresh}
 (\lambda \text{ proj } E \text{ D}) \Rightarrow
 let (T', F, C, S) = R( A, E )
 if T' = \text{(poly } (I K) T \rangle \text{ then}
 (D[I]T, F, [D/I]C, S)
 \text{else fail})
\]

where \( F' \) is extended to constraint sets and environments, with \( F'V[I : T] = F'V(T) \) and \( F'V[I : K] = \{ I \} \).

In the case of the polymorphic abstraction operator \( \lambda \text{ (lambda } \text{) \text{, a compound constraint set is constructed. The second part } C' \text{ of the constraint set propagates the current effect bindings. The purpose of the first part is twofold. } [N/I]\nu_1, [I_1][\{(F, \text{pure})\} \text{ 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 \}. [N/I]\nu_1, [I_1][\{(F, \text{pure})\} \text{ arranges for the body of the expression to always pure. New unification variables } \nu_i \text{ 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)** \( R( A, E ) \text{ terminates.} \)

**Proof.** \( R \) works by induction on the structure of expressions, which are of finite height. \( \square \)

#### 5.2 Soundness

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

\[
 (m \models C) \Rightarrow mSA \vdash E : mT \vdash mF
\]

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

For \( \text{proj} \), suppose that:

\[
 (\langle D/I \rangle T, F, [D/I]C, S) = R( A, (\text{proj } E D))
\]
and \(m = [B/I]C\). We want to prove:

\[
mSA \vdash (\text{proj E F}) : m([B/I]T) \Downarrow mF
\]

(1)

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

\[
mSA \vdash E : (\text{poly} (I K) mT) \Downarrow mF
\]

Since, by definition of \(\mathcal{R}\),

\[
((\text{poly} (I K) T), F, C, S) = \mathcal{R}(A, E)
\]

then by induction:

\[
(m' \models C) \implies m'sA \vdash E : m'(\text{poly} (I K) T) \Downarrow m'F
\]

Since, as before,

\[
m'(\text{poly} (I K) T) = (\text{poly} (I K) m'T)
\]

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

For application, suppose that \(\mathcal{R}(A, (E_0, E_1))\) is equal to

\[
(Sv_2, (\text{maxeff} F_0 (\text{maxeff} F_1 \nu_1)), C_0 \cup C_1 \cup C, SS_1 S_0)
\]

and \(m \models C_0 \cup C_1 \cup C\). We want to prove that in the environment \(mS S_1 S_0 A\):

\[
(E_0, E_1) \vdash (Sv_2) ! m(\text{maxeff} F_0 (\text{maxeff} F_1 \nu_1)) \quad (2)
\]

However, by using structural induction, we can apply \(S S_1\) to \(mS_0 A \vdash E_0 : mT_0 \downarrow mF_0\) and apply \(S\) to \(mS_1 S_0 A \vdash E_1 : mT_1 \downarrow mF_1\), since \(m \models C_0\) and \(m \models C_1\). By the unification lemma, since \(m \models C\):

\[
mS S_1 S_0 T_0 \sim (\text{proc} m\nu_1 (mS T_1) mS v_2)
\]

To complete the proof of (2), we thus only have to use the typing rule (→E). □

### 5.3 Completeness

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

\[
\begin{align*}
(T', F', C', S') &= \mathcal{R}(A, E) \\
m'P'S'A &= mSA \text{ on } FV(E) \\
m' &\models C' \\
T &\sim m'P'T' \\
F &\sim m'F'
\end{align*}
\]

**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 \(M\) types, which shows how to deal with unification variables, and polymorphic abstraction, which checks for pureness of expressions.

For \(\text{lambda}\) on \(M\) types, assume that

\[
mSA \vdash (\text{lambda} (I) E) : (\text{proc} F (M) T) \Downarrow \text{pure}
\]

By \((→\text{M})\), this requires

\[
mSA[I : M] \vdash E : T \Downarrow F
\]

which is equivalent to

\[
m[S[M/\nu]](A[I : \nu]) \vdash E : T \Downarrow F
\]

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

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

To prove the theorem, pick:

\[
T' = (\text{proc} F'' (S''\nu) T'')
\]

\[
F' = \text{pure}
\]

\[
C' = C''
\]

\[
S' = S''
\]

\[
m' = m''
\]

\[
P' = P''
\]

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

\[
m'P'T' = (\text{proc} m''P'' (m''P''S''\nu) m''P''T'')
\]

\[
= (\text{proc} F (M) T)
\]

For \(\text{plambda}\), assume that

\[
mSA \vdash (\text{plambda} (I K) E) : (\text{poly} (I K) T) \Downarrow \text{pure}
\]

By \((\text{polyI})\), this requires

\[
(mSA)[I :: K] \vdash E : T \Downarrow \text{pure}
\]

which is equivalent to

\[
mS(A[I :: K]) \vdash E : T \Downarrow \text{pure}
\]

By induction, there exist \((T_0, F_0, C_0, S_0)\), \(m_0\) and \(P_0\) that verify the theorem for \([N/I]E\) in the environment \(mSA\). To prove the theorem, pick:

\[
T' = (\text{poly} (I K) T_0)
\]

\[
F' = \text{pure}
\]

\[
C' = [N/I][\nu_1 / I](C_0 \cup \{F_0, \text{pure}\}) \cup (C_0 \cup \{F_0, \text{pure}\})
\]

\[
S' = S_0
\]

\[
m' = [m_1 I / \nu_1]m_0
\]

\[
P' = P_0
\]

where \(\{L\} = (FV(F_0) \cup FV(C_0)) - FV(S_0(A[I :: K]))\) and \(\nu_1\) are fresh. The theorem is trivially verified, except for the verification of constraints. Since \(m_0 \models C_0\) and \(m_0 F_0 \sim \text{pure}\), then \(m' \models C_0 \cup \{F_0, \text{pure}\}\). For the first term \(C'\) of \(C'\) defined by

\[
C' = [N/I][\nu_1 / I](C_0 \cup \{F_0, \text{pure}\})
\]
$m' \models C'$ can be successively rewritten as:

$$m' \models C' = (m_1 L / \nu_I) m_0 \models [N / I][\nu_I / L] (C_0 \cup \{(F_0, \text{pure})\}) =$$

$$m_0 \models [N / I][m_1 L / L] (C_0 \cup \{(F_0, \text{pure})\}) =$$

$$(m_1 \models C_1) \land (m_1 P_1 \sim \text{pure})$$

since the set of free variables $\{L_1\}$ of $F_0$ and $C_0$ is the same (more precisely, is isomorphic to) the one of $P_1$ 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 tractability is rooted in 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:

$$(\text{maxeff } X_1 \ldots X_m) \sim (\text{maxeff } Y_1 \ldots Y_n)$$

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 $E_i$ to $E_N$ drawn from the constraint set, we can rewrite an equation as the following set of $N$ double implications:

$$(E_1 \in X_1 \lor \ldots \lor E_1 \in X_m) \Leftrightarrow (E_1 \in Y_1 \lor \ldots \lor E_1 \in Y_n)$$

$$(E_N \in X_1 \lor \ldots \lor E_N \in X_m) \Leftrightarrow (E_N \in Y_1 \lor \ldots \lor E_N \in Y_n)$$

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:

$$(E_i \in X_1 \Rightarrow (E_i \in Y_1 \lor \ldots \lor E_i \in Y_n))$$

... $$(E_i \in X_m \Rightarrow (E_i \in Y_1 \lor \ldots \lor E_i \in Y_n))$$

$$(E_i \in Y_1 \Rightarrow (E_i \in X_1 \lor \ldots \lor E_i \in X_m))$$

... $$(E_i \in Y_n \Rightarrow (E_i \in X_1 \lor \ldots \lor E_i \in X_m))$$

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

$$E_i \not\in X_j \Leftrightarrow (E_i \not\in Y_1 \land \ldots \land E_i \not\in Y_n)$$

and similarly for $Y$. 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:

$$E_i \not\in E_j \text{ if } i \text{ is different from } j$$

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 lambda construct is encountered. Thus constraint set size is exponential in the nesting level of lambda constructs. We expect this not to be a major problem in practice since lambda 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 lambda constructs.

#### 6.1 Definitions

The language KFXlet adds the let binding construct to KFX:

$$E \leftrightarrow \begin{cases} \text{(let (I E_0) E_1) local binding} \end{cases}$$

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 if $E$ is not a variable, a lambda expression or a lambda 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 KFX_{let} includes all the rules used in KFX, plus the (Glet)\(^2\) and (let) rules:

\[
\begin{align*}
A & \vdash E_0 : T_0 \text{ pure} \\
A & \vdash [E_0/I_0]E_1 : T_1 \text{ ! } F_1 \\
\text{(Glet)} \quad \text{if } E_0 \text{ is not expandible} \\
A & \vdash (\text{let } (I E_0) E_1) : T_1 \text{ ! } F_1 \\
\text{(let)} \quad A \vdash E_0 : T_0 \text{ ! } F_0 \\
A & \vdash I_0 : T_1 \text{ ! } F_1 \\
\text{where } \nu_i \text{ are fresh} \\
\text{else fail}
\end{align*}
\]

6.3 Correctness Theorems

\textbf{Theorem 7 (KFX_{let})} All the previous theorems about KFX stated in Section 5 extend trivially to KFX_{let}.

\textbf{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 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 \cite{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 Domic, James O'Toole, Mark Sheldon and Jean-Pierre Talpin for their insightful comments.

References


\(^2\)One can show \cite{M89} that, in ML, this is equivalent to the usual type-scheme based approach.


