Our LIPS Are Sealed: Interfacing Logic and Functional Programming Systems

Gary Lindstrom[1]
Jan Małuszyński[2]
Takeshi Ogi[3]

UUCS-92-009

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

March 23, 1992

Abstract

We report on a technique for interfacing an untyped logic language to a statically polymorphically typed functional language. Our key insight is that polymorphic types can be interpreted as "need to know" specifications on function arguments. This leads to a criterion for liberally yet safely invoking the functional language to reduce application terms as required during unification in the logic language. This method, called P-unification, enriches the capabilities of each language while retaining the integrity of their individual semantics and implementation technologies. An experimental test has been successfully performed, whereby a Horn clause logic programming (HCLP) interpreter written in Common Lisp was interfaced to the Standard ML of New Jersey system. The latter implementation was employed (i) on untyped or dynamically typed data, even though it is statically typed; (ii) lazily, even though it is strict, and (iii) on alien HCLP terms such as unbound variables — without the slightest modification!

[1]This material is based upon work supported by the National Science Foundation under Grants CCR-8920971 and ASC-9016131.

[2]Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden.

[3]Kyocera Corp., 2-14-9 Tamagawa-dai, Setagaya-ku, Tokyo, Japan.

1 Motivation

There have been countless attempts to combine the best features of functional programming (FP), e.g. functional notation, higher order objects, lazy evaluation, etc., with those of logic programming (LP), e.g. "don't know" nondeterminism, computation on non-ground data, constraint-based search, etc. Most of these approaches are described as "amalgamation," i.e. homogenization into a new language. At best, this semantic stew emerges as a purée; at worst, a goulash. In any case, the contributing languages surrender their semantic integrity, and combining their implementation technologies becomes problematic.

We address this problem from a different perspective, in keeping with the trend toward "open systems." Our approach invokes the STANDARD ML OF NEW JERSEY (SMLNJ) FP language (FPL) system as a reduction server to an LP language (LPL) client, written in COMMON LISP.

This work builds on the approach presented in [BM88] to integrating a logic language with an external functional language. The declarative semantics of this integration is defined in terms of an equational logic. The operational semantics suggested there is based on an extended unification method, called *S-unification*. S-unification is inherently incomplete since it combines term unification with function application, restricted to ground arguments. We present here an extended form of S-unification, taking into account polymorphic type information. This allows us to relax the ground argument requirement in many cases.

2 Language Requirements

We assume that the FPL is (i) purely functional *in effect* (i.e. semantically benign use of imperative features is permitted); (ii) statically polymorphically typed; (iii) applicative-order (strict, non-lazy, call-by-value), and (iv) equipped with constructor-based user defined types. The LPL is not assumed to be statically typed, since many of today's LPL's, including most versions of PROLOG, are untyped. However, we assume that the type signatures of all accessible FPL data objects (including function identifiers) are known to the LPL.

The polymorphic type domain underlying our LPL and FPL combination is shown in Fig. 1. This includes the core of STANDARD ML [Myc84], since:

1. Primitive data types int, bool, string etc. are subsumed by (T_1, ..., T_n) Dtype, with $n = 0$ and parentheses omitted.

2. Function data types $\mathsf{T}_1 \rightarrow \mathsf{T}_2$ are subsumed by (T_1, T_2) func.

3. Tuple types are subsumed by (T_1, ..., T_n) cross.

Note that substitution on Tvar's creates a partial order on T, whereby $\mathsf{T}_1 \sqsubseteq \mathsf{T}_2$ iff there exists a substitution Θ such that $\mathsf{T}_1\Theta = \mathsf{T}_2$. Traditional surface representations will be used henceforth, e.g. int, $\alpha \times \beta$, α list, α list $\rightarrow \alpha$, etc.

3 LPL Terms

Our term language TL is a first-order LPL term language extended by a binary functor apply, with the interpretation that apply(fn, arg) means "rewrite this term by evaluating fn arg in the FPL".

$$\begin{aligned}\mathsf{Term} ::= {} & \mathsf{Var} \mid \mathsf{Constant} \mid \mathsf{Functor}(\mathsf{Term}_1, \ldots, \mathsf{Term}_n) \qquad (n \geq 1)\\ & \mid \mathsf{apply}(\mathsf{Term}_1, \mathsf{Term}_2)\end{aligned}$$

Var denotes logical variables, and Constant denotes atomic symbols (e.g. integers). The syntax Functor(Term_1, ..., Term_n) is reserved for *constructions* (i.e. constructor applications), where the constructor may be (i) known to both languages (e.g. cons(...)), or (ii) unknown in the FPL (e.g. foo(...)). We consider parentheses to be optional for nullary constructors, i.e. f and f() are synonyms. The construction tuple(Term_1, ..., Term_n), for $n \geq 2$, represents functor-less tuples. Once again, more congenial surface representations will often be used henceforth, e.g. [1, 2] (rather than cons(1, cons(2, nil))), [true, false | X] (rather than cons(true, cons(false, X))), (1, true) (rather than tuple(1, true)), and 1+2 (rather than apply(plus, tuple(1, 2))).

4 Datatypes: Common, Alien, and Mixed

Some types in T represent values that are meaningful to both the FPL and LPL. These values include common primitive types (int, bool, ...), and certain constructions, e.g. tuples and lists. The surface representation of these values will generally need conversion as they pass between languages (e.g. the FPL value (v_1, ..., v_n) will need to be converted to the LPL

$$\mathsf{T} ::= \mathsf{Tvar} \mid (\mathsf{T}_1, \ldots, \mathsf{T}_n)\ \mathsf{Dtype}$$

$$\mathsf{Tvar} ::= \alpha \mid \beta \mid \ldots$$

$$\mathsf{Dtype} ::= \mathsf{Identifier}$$

All schemes are implicitly closed (via outermost $\forall\alpha$ ...).

Figure 1: FPL polymorphic type domain.

term tuple(v_1, ..., v_n) — see §3), but we assume that no semantically significant mapping issues arise.

There are, in addition, data values that each language can manipulate but must regard as semantically "alien." For example, the LPL must appeal to the FPL for interpretation of non-constructor functions, while the FPL cannot be expected to make sense of logical variables, ill-typed expressions, or insufficiently instantiated constructions such as [1 | X] $\equiv$ cons(1, X), where X is an unbound variable. Hence the data under manipulation will in general be, from the perspective of each language, a mixture of common, private and alien objects.

5 A "Need to Know" Strategy For Term Reduction

Clearly one cannot expect the FPL to reduce terms such as apply(plus, tuple(1, true)). Yet one might hope that apply(head, [1, X]) could yield 1, and even that apply(tail, [1, X]) could yield [X]. How might such a "liberalized" sense of function application be safely obtained? By observing that *variables in function domain types indicate "hands off" treatment.* Thus length: α list $\rightarrow$ int means that list elements: (i) are not inspected by length; (ii) need not be evaluated; (iii) can be objects "alien" to the FPL, and (iv) need not even be typeable, static FPL typing notwithstanding!

Note this type scheme interpretation significantly extends the interpretation customarily observed in SML, which construes the polymorphic type α list $\rightarrow$ int to be the union of all its monomorphic instances (int list $\rightarrow$ int, bool list $\rightarrow$ int, int list list $\rightarrow$ int, etc.).

Our strategy is embodied in a mechanism for enlisting the FPL as a reduction server as needed by the LPL during unification. Our approach is thus (yet another) version of *extended unification* [DV87], which we term *P-unification.* A term apply(fn, arg) is reduced *lazily*, i.e. as needed during unification, but only when (i) fn has become a function, and (ii) arg has become acceptably instantiated to the degree dictated by the domain type of fn. Examples are given in Fig. 2 (resulting substitutions are omitted).[4]

We will define P-unification in three increasingly precise presentations. In each case, the method will be described operationally by means of prioritized, symmetric rules for eliminating *term disagreement pairs.* We omit the occur check in each case to simplify the presentation. In the presence of apply terms the occur check becomes more elaborate, since e.g. the terms X and apply(+, tuple(X, 0)), where + is the addition function, are unifiable. In this paper we focus on the problem of using types for interfacing functional and logic

[4]The fact that unifying [P, Q] and apply(apply(map, add1), [13, Y]) *suspends* rather than *succeeds* with {P := 14, Q := apply(add1, Y)} is a consequence of our FPL strictness assumption.

U	V	*P-unify*(U, V)
X	apply(length, Y)	succeeds
1	apply(length, [ ])	fails
2	apply(length, [R, S])	succeeds
3	apply(length, [R, S | T])	suspends
[P, Q]	apply(apply(map, add1), [13, Y])	suspends
[2, 0]	apply(apply(map, length), [[13, Y], [ ]])	succeeds

Figure 2: P-unification examples.

programming systems. A proper treatment of the occur check problem would follow the approach of S-unification [BM88, Bon91].

P-unification Formulation 1:

1. $\{v$, x$\}$, where v is an unbound variable, and x is arbitrary: Bind v to x, and remove this pair from the disagreement set.

2. $\{f_1(t_{1,1}, ..., t_{1,n1}), f_2(t_{2,1}, ..., t_{2,n2})\}$, where f_1 and f_2 are constructors: If $f_1 \neq f_2$, or $n1 \neq n2$, fail; otherwise, remove this pair from the disagreement set and add the pairs $\{t_{1,i}, t_{2,i}\}$ for $i = 1, ..., n1$ (subsumes constants and tuples).

3. {x, y}, where x = apply(fn, arg):

   (a) If the FPL can be invoked without type error to reduce x, do so and reconsider this pair.

   (b) If x might be instantiated to meet condition (a), defer consideration of this pair until that possibility is resolved affirmatively or negatively.

   (c) Otherwise, fail.

   The apply rule is the focus of this work, and will be made more precise in the following sections.

Failure arises in 3(c) because no instantiation of arguments could permit reduction of the apply term, even in the "liberalized" sense being defined here. Hence a fundamental typing error has been detected, and the apply term is meaningless in both languages.

Because of the concept of deferred pairs, a non-failed P-unification results generally in a set of bindings and a set of deferred pairs, which may or may not be processed at later unification steps, depending on subsequently produced variable bindings. A formal presentation of this idea in the context of S-unification can be found in [Boy91], where sufficient conditions for the absence of deferred pairs upon non-failed termination of a program are given and in [KK91] where an implementation of this concept is described.

Note that constructions receive special treatment with respect to other function applications. In particular, they: (i) enjoy a special syntax f(t_1, ..., t_n); (ii) can (if f is known to the FPL) be either applied by the FPL or be selected directly upon by the LPL during unification, and (iii) have an LPL meaning even if they do not conform to the FPL's static typing. Hence we contrast X+Y and cons(X, Y), in the case that X becomes 1 and Y becomes true. That is, cons(1, true), while malformed as an FPL object, can continue to serve as a valid construction in the (untyped) LPL, while 1+true is meaningless in both languages.

6 Reduction Desiderata

How should an apply(fn, arg) term be reduced during P-unification? We claim the following properties are desirable:

1. **Safety:** No reduction service request should cause an FPL type error.

2. **Laziness:** Subterms in apply(fn, arg) terms should be reduced only if their values are needed (i) by an FPL reduction, or for successful completion of a P-unification step. For example, we aspire *not* to apply acker in apply(head, [1, apply(acker, (100, 99))]), since under a lazy evaluation regime we need not evaluate the elements of a list to which head is applied.

3. **Maximality:** FPL evaluation requests should include a maximal composition of function applications, consistent with our laziness criterion. Thus add1(add1(add1(add1(13)))) should be evaluated in one service request, rather than in four.

In fact, laziness and maximality are somewhat opposing criteria. We will achieve a pragmatically attractive middle ground by means of (i) *sealed envelopes*, (ii) *maximal consensus types*, and (iii) *minimal term "truncations"* bearing these types.

7 Sealed Envelopes

Our type-based "need to know" strategy lets us encapsulate data objects alien to the FPL in carriers that hide the identity of these objects along with their type idiosyncrasies. This is accomplished through:

1. Augmenting T to include a *nullary type constructor* union, which will be attributed to subterms whose type is not germane to an FPL reduction request.
2. A *term constructor* seal(Term) of type $\alpha \ \rightarrow$ union. A seal application constitutes a static typing boundary. We denote the TL term language augmented with seal as the augmented term language ATL.
3. An *FPL datatype definition* datatype union = seal of int.
4. An *LPL* $\leftrightarrow$ *FPL* interface convention:
   (a) On output to the FPL, seal(x) is translated to seal(χ), where χ = loc x, the address of the expression x in the LPL address space.
   (b) On input from the FPL, seal(i) is translated to deref(i), the expression at address i.[5]

To illustrate, consider t = head [1, true]. This is not acceptable to the FPL, due to its heterogeneous list [1, true]. However, if t is reformulated as head [seal(loc 1), seal(loc true)], then the FPL can be invoked to return seal(loc 1), which yields 1 upon dereferencing.

8 Attributing Types to TL Terms

P-unification, as outlined in §5, presumes testability of an apply term for error-free FPL reducibility. Our formal realization of this criterion is based on a notion of type attribution for TL terms, which permits (i) a maximal type notion for TL terms, guaranteeing reduction safety, and (ii) a type-driven reduction scheme meeting the laziness and maximality criteria of §6.

We begin by adapting the Hindley-Milner type attribution method [Mil78] to apply to ATL terms. This done by:

1. Interpreting apply(fn, arg) as having type β, where the type of fn is constrained to $\alpha \rightarrow \beta$, and the type of arg is constrained to α.

[5]If addresses are unstable, as with compacting garbage collection, then symbolic names can be used.

2. Interpreting constructions $\mathsf{f(t_1, ..., t_n)}$, where f has an FPL-type, as $\mathsf{apply(f, tuple(t_1, ..., t_n))}$, if $n \geq 2$, and $\mathsf{apply(f, t_1)}$ if $n = 1$.

3. Viewing as untypeable:

   (a) Unbound variables, and

   (b) Constructions foo(a, 13), where foo is an LPL-only constructor. Note this implies untypeability of LPL-only symbols, e.g. a in [1, a].

4. Interpreting seal(t) to have type union, as per §7, independent of whether t is typeable.

This extension to the Hindley-Milner method preserves its principal type property.

Theorem 1 *If $t \in$ ATL has a type, then t is attributed a most general type, i.e. all other attributable types can be obtained by substitutions on this most general type.*

Proof. If t has a type, then all its subterms must be typeable, except possibly those within seal(...) occurrences. Let us substitute seal(true) in t for each occurrence of seal(...), and call the new term t'. Clearly, the set of types attributable to t and to t' are the same. But t' is now isomorphic to an ordinary FPL term, which has a principal Hindley-Milner type. Moreover, our extended method reduces to the Hindley-Milner method on t'. Hence our method must yield a principal type for t', and that must be the principal type of t. $\square$

We denote the principal type of $t \in$ ATL, if it exists, by $\tau(t)$. Let t be a TL term. A *truncation* $t1 \in$ ATL of t is constructed from t by the introduction of zero or more unnested seal constructions. Let $trunc(t)$ be the set of all truncations of t. A subterm occurrence s in $t1 \in trunc(t)$ is *sealed* if s is immediately surrounded by a seal construction, or appears within a sealed subterm occurrence. We say $t1 \sqsubseteq t2$ if every sealed subterm occurrence in $t2$ is sealed in $t1$. Observe that $\sqsubseteq$ is a partial order on $trunc(t)$, and forms a (finite) lattice, with $\perp =$ seal(t), and $\top = t$.

We now change the ordering on our type domain T to deal more effectively with the effects of truncation. The primitive type union is repositioned such that (i) $\alpha \sqsubseteq$ union $\sqsubseteq ty$ for all $ty \neq \alpha$ (or any other type variable), and (ii) the same ordering holds recursively under type constructors, e.g. α list $\sqsubseteq$ union list $\sqsubseteq$ int list, etc.

If $t \in$ TL possesses no subterms with principal type α then we say it is *α-free*. Henceforth we will assume all TL terms are α-free.

Theorem 2 *Let $t \in$ TL be α-free. Then τ is monotonic on $trunc(t)$, wherever it is defined. That is, if $t1$ and $t2$ are both typeable truncations of t, with $t1 \sqsubseteq t2$, then $\tau(t1) \sqsubseteq \tau(t2)$.*

Proof. Let $t1$ and $t2$ be two typeable truncations of a TL term with $t1 \sqsubseteq t2$. The typeability of a truncation t is equivalent to the existence of a most general unifier (mgu) Θ solving a set of equations $eq(t)$ on type variables $dom(\Theta)$ associated with the nodes of t that are either unsealed or directly sealed (i.e. occurrences of seal(...)).

Since $t1$ and $t2$ are typeable, $\Theta_1 = mgu(eq(t1))$ and $\Theta_2 = mgu(eq(t2))$ both exist. Let $v(t)$ be the type variable associated with the overall term t. Since $t1$ and $t2$ are truncations of the same TL term, $v(t1)$ is $v(t2)$. Hence $\tau(t1) = v(t1)\Theta_1$, and $\tau(t2) = v(t2)\Theta_2 = v(t1)\Theta_2$.

Let $tr(t)$ denote the equations in $eq(t)$ of the form $\mathsf{T}_i = \mathsf{union}$ arising from directly sealed subterms in t, and $e(t)$ denote $eq(t) - tr(t)$. Then $eq(t1) = e(t1) \cup tr(t1)$, and $eq(t2) = e(t1) \cup e'$, where $e' = (e(t2) - e(t1)) \cup tr(t2)$. Note that $mgu(eq(t1)) = mgu(mgu(e(t1)), mgu(tr(t1))) = mgu(mgu(e(t1)), tr(t1))$, and $mgu(eq(t2)) = mgu(mgu(e(t1)), mgu(e'))$.

Since $mgu(eq(t2))$ exists, so must $mgu(e')$. By α-freeness, all type variables in $dom(e')$ must be bound to values union or greater. We say two mgu's Θ and Θ' obey $\Theta \sqsubseteq \Theta'$ iff $dom(\Theta) \subseteq dom(\Theta')$, and $\forall v \in dom(\Theta), v\Theta \sqsubseteq v\Theta'$. We have $dom(tr(t1)) \subseteq dom(e')$, so $mgu(tr(t1)) \sqsubseteq mgu(e')$; hence $mgu(eq(t1)) \sqsubseteq mgu(eq(t2))$, and $\tau(t1) \sqsubseteq \tau(t2)$[6] $\Box$

A word of explanation is appropriate concerning our exclusion of ATL terms possessing subterms of principal type α. Such subterms must denote "fully polymorphic" FPL values, which are anomalous. Expressions denoting such objects include: (i) head nil; (ii) f 13 where fun f x = f x, and (iii) f 13 where fun f x = raise exception1. In short, α-typed expressions can never deliver values, due to inescapable divergence or exception.

9 Maximal Types of TL Terms

Let us define $t1 \in trunc(t)$ to be *FPL-safe* if $t1$, as outputted under the seal interface protocol described in §7, is typeable by the FPL.

Theorem 3 *Every typeable truncation of a* TL *term is FPL-safe.*

Proof. Follows by an argument similar to that for Theorem 1. $\Box$

Note that at least one typeable truncation always exists for any $t \in \mathsf{TL}$, since at worst we can seal t in its entirety.

Theorem 4 *If $t1$ and $t2$ are two typeable truncations of a term t, then $t1 \sqcup t2$ is typeable.*

Proof. Our proof uses the notation introduced in the proof of Theorem 2. To prove typeability of $t1 \sqcup t2$ it suffices to prove existence of $mgu(eq(t1 \sqcup t2))$. Denote by $e(t1, t2)$ the intersection of the sets $e(t1)$ and $e(t2)$, and by $e'(t2)$ the set $e(t2) - e(t1, t2)$. Hence

[6]What happens if a TL term is not α-free? Consider t = apply(head,nil), with Hindley-Milner principal type α, and $trunc(t) = \{t, \text{seal}(t)\}$. The maximal typeable truncation of t is t itself with type α, but its maximal type is union. Hence τ is not monotonic on $trunc(t)$.

$eq(t1 \sqcup t2) = e(t1) \cup e'(t2) \cup tr(t1 \sqcup t2)$ and we seek to prove existence of its mgu.

Clearly the set $e(t1)$ has a unifier since $t1$ is typeable. The set $e'(t2)$ has a unifier since it is a subset of $e(t2)$. The set $tr(t1 \sqcup t2)$ includes the equations arising from the directly sealed subterms of $t1 \sqcup t2$. The set $e'(t2) \cup (tr(t1 \sqcup t2) - tr(t1))$ is a subset of $eq(t2)$ and has a unifier. The remaining equations in $tr(t1 \sqcup t2)$ share no variables with this set. Thus $e'(t2) \cup tr(t1 \sqcup t2)$ also has a unifier. Now $mgu(e(t1) \cup e'(t2) \cup tr(t1 \sqcup t2))$ can be computed as $mgu(\tau 1, \tau 2)$, where $\tau 1 = mgu(e(t1))$ and $\tau 2 = mgu(e'(t2) \cup tr(t1 \sqcup t2))$.

We now show existence of $mgu(\tau 1, \tau 2)$. Consider the set V of variables of $e(t1)$ denoting the types of the directly sealed subterms of $t1$. Since the type equations in $e(t1)$ include no occurrence of union $\tau 1$ binds these variables at most to type variables (or leaves them unbound). Otherwise they could not be bound to union and $t1$ would not be typeable. On the other hand, $\tau 2$ binds every variable of V to a type not including the variables of V, since these variables characterize types of the disjoint subterms of $t1 \sqcup t2$ truncated by the boundary of $t1$.

Two cases should be considered.

Case 1: No variables of V are bound by $\tau 1$ to a common type variable. In this case $mgu(\tau 1, \tau 2)$ exists and is obtained by binding the variable $\tau 1(x)$ to the type $\tau 2(x)$ for every $x \in V$.

Case 2: Some variables x and y in V are bound by $\tau 1$ to a common type variable.

- *Case 2a:* Both x and y occur either in (1) $tr(t1) - eq(t2)$ or in (2) $tr(t1) \cap eq(t2)$. If (1) applies, $\tau 2(x) = \tau 2(y) = \mathsf{union}$. If (2) applies, then by the assumption of typeability of $t2$ terms $\tau 2(x)$ and $\tau 2(y)$ must be unifiable, since $e'(t2)$ is a subset of the unifiable set $eq(t2)$. From these observations one can conclude existence of $mgu(\tau 1, \tau 2)$.

- *Case 2b:* One of the variables x, y, say x, occurs in $tr(t1) - eq(t2)$ while the other occurs in (2) $tr(t1) \cap eq(t2)$. In that case there exists a variable z in $tr(t2) \cap eq(t1)$ which is bound by $\tau 1$ to the same variable as x. This is because y is both in $eq(t1)$ and $eq(t2)$ while x is not in $eq(t2)$. As they are bound to the same variable they must "communicate" over the boundary of $t2$ in $t1$. The mgu of $e(t2)$ binds z to a variable since it is bound to union by the mgu of $eq(t2)$. It also must bind z and y to a common variable since $\tau 1$ does that. Hence $e'(t2)$ which is a subset of $e(t2)$ can only bind y to a variable. Consequently $\tau 2(x) = \tau 2(y) = \mathsf{union}$ which allows us to conclude the existence of $mgu(\tau 1, \tau 2)$.

Given that *Case 1* and *Case 2* are exhaustive, we see that the set $e(t1) \cup e(t2)$ has an mgu.
□

The maximal typeable truncation of a term is the lub of its typeable truncations. Hence every term has a unique maximal type, yielded by its maximal typeable truncation. This permits us to extend τ to be total on TL, by defining $\tau(t)$ to be the type of the maximal Hindley-Milner typeable truncation of t.

Note that the maximal type of t may be assumed by more than one of its truncations. Consider for example t = head [1, true]. The maximal type of t is union, obtained from both $t1$ = seal(head [1, true]) and $t2$ = head [seal(1), seal(true)]. However, $t1 \sqsubset t2$, so $t2$ corresponds to a greater amount of FPL reduction. Indeed, maximal typeable (FPL-safe) truncations indicate maximal FPL reductions. This leads to:

P-unification Formulation 2:

When reducing t = apply(fn, arg) in Step 3 of Formulation 1:

- Compute the maximum typeable truncation $t1$ of t.
- If $t1$ = seal(...), suspend.
- Otherwise, reduce $t1$ and continue.

Unification of two TL terms fails when (i) both of their types are of the form $\delta \rightarrow \rho$ (since we decline to unify functions for soundness reasons), or (ii) their types are not compatible, as explained in §11.

This strategy is *safe* (by Theorem 3) but sacrifices *laziness* (consider the apply(head, [1, apply(acker, (100, 99))]) given earlier). We assert informally that it is also *maximal*.

10 Implementing Type Attribution

Truncation enumeration and typeability testing can be merged into a simple, acceptably efficient algorithm succinctly expressible in PROLOG (see Fig. 3). Let $R(t)$ denote our PROLOG representation of TL term t. $R(t)$ = `e(T, S)`, where:

1. `T` is a Prolog term representing a type attributed to t, i.e. `int`, `bool`, `union`, etc., `cross(T`$_1$`, ..., T`$_n$`)`, `list(T`$_1$`)`, `arrow(T`$_1$`, T`$_2$`)`, or `_` (an unbound variable, for untyped terms).
2. `S` is a Prolog term representing the syntax of t:

```
type(e(R, apply(F, A))) :-
   F = e(arrow(D, R), _), A = e(D, _), type(F), type(A).

type(e(cross(T1, T2), tuple(E1, E2))) :-
   E1 = e(T1, _), E2 = e(T2, _), type(E1), type(E2).

type(e(list(T1), cons(E1, E2))) :-
   E1 = e(T1, _), E2 = e(list(T1), _), type(E1), type(E2).

type(e(list(_), nil)).

type(e(int, S)) :- integer(S).

type(e(bool, true)).

type(e(bool, false)).

type(e(arrow(list(T), T), head)).

type(e(arrow(int, arrow(int, int)), plus)).

type(e(arrow(arrow(T1, T2), arrow(list(T1), list(T2))), map)).

type(e(arrow(X, X), ident)).

...

type(e(union, _)).
```

Figure 3: Type attribution in Prolog

t	S
V $\in$ Var	var(V)
f(t_1, ..., t_n) *(f an LPL-only constructor)*	f(t_1, ..., t_n) *(subsumes constants and tuples)*
apply(t_1, t_2) *(includes FPL-constructions; see §8)*	apply($R(t_1)$, $R(t_2)$)

Hence: nil $\Leftrightarrow$ e(_, nil); foo(1, true) $\Leftrightarrow$ e(_, foo(1, true)); apply(add1, X) $\Leftrightarrow$ e(_, apply(e(_, add1), e(_, var(X)))), etc.

Note that in the algorithm of Fig. 3:

1. Typing of subterm occurrences is attempted in a top-down, left-right order.
2. Clause `type(e(union, _)).` implicitly seals a given subterm, as last resort.
3. Sealed subterms are unnested.
4. Type union is the only attribution to variables and LPL-only constructions.
5. All typeable truncations are enumerated, with principal types, in topologically sorted order (maximal first).
6. Backtracking is somewhat "intelligent," in that attributing union to a subterm will "short circuit" type attribution search on subterms that must agree in type. For example, once union is attributed to true (by implicitly sealing it) in [true, apply(acker, tuple(100, 99))], attribution of union to the apply term will ensue without attempting any type attribution of tuple(100, 99).

11 P-unification Formulation 3

We can now precisely define our P-unification algorithm. Given the preliminary algorithm in §5, we need only refine the case where x = apply(fn, arg), and y is either a construction or an apply term. Let $L = R($[x, y]$)$. First, we solve `type(e(T,` L`))`, obtaining τ_{max}, the maximal type of L.

- *Case 1:* τ_{max} = `list(arrow(_, _))`: *Fail.*
- *Case 2:* τ_{max} = `list(union)`:
  - ◇ *Case 2a:* Only one truncation of L has this type, viz. [seal(x), seal(y)]: *Suspend.*
  - ◇ *Case 2b:* More than one truncation of L is attributed type `list(union)`. *Let $t1$ be a minimal typeable truncation of L greater than* [seal(x), seal(y)]. *Reduce* x *and* y *using their respective truncations in $t1$, and reconsider this pair.*
- *Case 3:* `list(union)` $\sqsubset$ τ_{max}: *Let t be a minimal truncation of L with type τ_{max}. Reduce* x *and* y *using their respective truncations in t, and reconsider this pair.*

This *"max type / min truncation"* reduction strategy is our key to achieving both *maximality* and *laziness*. The favorable enumeration order of typeable truncations provided by the algorithm in Fig. 3 means that we can abort the solution of `type(e(T,` L`))` as soon as the *second* distinct binding for `T` results (and select the truncation constructed immediately prior). This strategy encourages laziness, as the following examples indicate.

- τ_{max} = `list(union)`: Let $L = R($[apply(head, [X]), apply(head, [Y])]$)$. The maximal typeable truncation of L is t_{max} = [apply(head, [seal(X)]), apply(head, [seal(Y)])], and the minimal truncation with this type is t_{min} = [seal(apply(head, [X])), seal(apply(head, [Y]))]. Two other typeable truncations lie between t_{max} and t_{min}, viz. t_1 = [apply(head, [seal(X)]), seal(apply(head, [Y]))] and t_2 = [seal(apply(head, [X])), apply(head, [seal(Y)])]. Unfortunately, t_1 and t_2 are incomparable, and $t_1 \sqcap t_2 = t_{min}$, which indicates no reduction of the two terms. Selecting *either* t_1 or t_2 will cause some reduction to take place, and constitute progress toward success or failure of the unification step (successful, in this case). Which truncation to choose is imponderable, so an arbitrary selection must be made. Of course, a more aggressive (less lazy) implementation may opt for reduction as indicated by t_{max}, which is always unique (and outputted first by our algorithm).

- $\tau_{max} \sqsubset$ `list(union)`: Now let L = [apply(head, [false]), apply(first, tuple(true, apply(acker, tuple(100, 99))))], where τ(first) $= \alpha \times \beta \rightarrow \alpha$. The maximal typeable truncation t_{max} of L is L itself, with type `list(bool)`. However, another typeable truncation of t exists with the same type: t_1 = [apply(head, [false]), apply(first, tuple(true, seal(apply(acker, tuple(100, 99)))))]. We elect to reduce according to t_1 in the spirit of laziness, thereby avoiding evaluation of acker(100, 99).

One may ask: what are sufficient grounds for turning suspension (*Case 2a*) into failure? *When the maximal typeable truncations of all ground instances of L seal both arguments of its list.* An effective test for this is easily implemented:

- Solve `susp(e(T,` L`))`, where procedure `susp/1` renames `type/1`, with added clause `susp(e(X, var(X))).` positioned before the final clause `susp(e(union, _)).`

- If there is only one typeable truncation of L with type greater than union (necessarily, with both of its list elements sealed), *fail*. Otherwise, *suspend*.

Note the trick in clause `susp(e(X, var(X))).` of pressing a variable `X` into service as its own (unbound) type denotation. This verifies that a potential type consensus exists for all uses of `X`. Hence we will detect that (i) {1+X, 13} and {1+X, 3-X} should *suspend*, but (ii) {1+X, true}, {1+true, 13}, and {apply(length, X), 1+X} should *fail*.

12 Type Retention, Polymorphism, and Lazy Copying

Our type attribution method for TL terms has another important monotonicity property:

Type	Term Representation
union	`union(_)`
int	`union(int)`
α list	`union(list(_))`
union list	`union(list(union(_)))`
int list	`union(list(union(int)))`
α list list	`union(list(union(list(_))))`

Figure 4: Monotonic term representation of types

Theorem 5 *Let t and t' be typeable* TL *terms, with $t\Theta = t'$ for some substitution Θ. Then $\tau(t) \sqsubseteq \tau(t')$.*

Proof. Let t'' be t' with a seal(...) surrounding each occurrence in t of a variable bound by Θ. Clearly, t'' is typeable, and has the same type as t. Since $t'' \sqsubseteq t'$, by Theorem 2 $\tau(t'') \sqsubseteq \tau(t')$, hence $\tau(t) \sqsubseteq \tau(t')$. $\square$

Since type attribution is monotonic on variable instantiation, could we retain prior attributions of a term t to give us a "head start" on subsequent typings? The answer is *yes* in principle, but several complications arise.

1. A term representation for types must be designed that permits type attributions to be "raised" by variable binding. This not true of the representation used in Fig. 3, since union $\sqsubseteq$ int, but `union` may not be instantiated to become `int`. Such a representation is illustrated in Fig. 4.

2. The type attribution code of Fig. 3 can be amended to use the representation shown in Fig. 4. However, care must be taken that sealed subterms are given *fixed* union attributions, lest [1, true] be attributed type bool list. This can result from the `union(_)` attribution of seal(1) being raised by unification with the type of true to `union(bool)`. A suitable defense is to use `type(e(union(fixed), _)).` as our "sealing" rule. In contrast, `susp(e(union(X), var(X))).` gives exactly the right "optimistic" typing effect. However, in both cases the `union(_)` argument bindings must be undone, if the type attributions are to be permanently retained.

3. The most severe impediment to retaining type attributions is the unfortunate collision of the "cultures" underlying polymorphism and lazy copying. Consider, for example,[7] goal G = `type(e(T, tuple(X, X)))`, where `X` has been bound in a prior unification step to `e(list(T1), nil)`. The sharing of `X`'s binding in G will cause the code in Fig. 3 to bind `T` to `cross(list(T1), list(T1))`, which incorrectly attributes only one

[7]For clarity, we revert in this example to our prior type representation.

degree of polymorphism to G. The correct binding is `cross(list(T1), list(T2))`. The implication is that we must copy the type attribution of every bound variable on each dereference — a very disheartening prospect indeed.

For these reasons, we advocate the simplicity and space economy of building transient type attributed `e(T,` $R(t)$`)` representations of TL terms only as needed during P-unification. Since this would be done by direct traversal, type variables would be created anew (i.e. "polymorphically") for each subterm encounter, independent of whether or not that term's representation is physically shared (example: `buildexp([], e(_,nil)).`).

Note that this construction would only be undertaken when unification cannot proceed without reducing an apply term. Hence ordinary LPL proceeds unimpeded (indeed, more efficiently than if type attributions were permanently associated with every term).

13 What About Equality?

The only primitive polymorphic (actually, overloaded) operator in SML is equality. Types required to admit equality are indicated by special type variables, denoted $'\alpha$. Consequently, functions which apply equality to their arguments indicate this fact in their type signatures, e.g. mkset $=$ $'\alpha$ list $\rightarrow$ $'\alpha$ list which removes duplicates from a list. We can easily distinguish types with equality in our domain by incorporating all $'\omega$ type variables, and ensuring that $'\omega$ variables in function argument types do not unify with function types or types containing union's. The effect of this on P-unification is suggested by the following examples:
(i) P-unifying 3 with length(mkset [X, Y, Z]) *suspends*;
(ii) P-unifying 3 with length(mkset, [1, 2, 1]) *fails*.

14 Experimental Test

An experimental test of P-unification was successfully performed using (i) an HCLP interpreter written in Common Lisp, and (ii) the SMLNJ implementation, each running as a separate Unix process [Ogi90]. Given that these processes communicate by stdin/stdout character streams, a quirky pragmatic difficulty arises: how to capture FPL results that don't print in full detail (e.g. functions, ref values, and truncated print representations)? The answer lies in yet another trick. Suppose we invoke head [add1, ident]. The printed result is val it $=$ fn : int $\rightarrow$ int. Although (appropriately enough) we cannot inspect the function returned, we *can* capture it by binding it to a global symbol, achieved by exporting e.g. val genfn1 = it, and henceforth referring to the function by its external name genfn1.

One may ask why we did not implement our HCLP interpreter in SML, and avoid the overhead of interprocess communication. In fact, we began our implementation in SMLNJ, with confidence that we could "finesse" the function application by the SML diction fn arg. Alas, we quickly learned that such is not possible within the expressive limits of SML's static type system, due to the need for unboundedly polymorphic user defined datatypes. The upshot is yet another confirmation of the folkloric fact that static typing is lovely until one undertakes "system" programming.

15 Related Work

In addition to the S-unification work mentioned earlier, the freeze / thaw notions in sequential implementations such as MU-PROLOG, and representative approaches to narrowing, are relevant. Barklund and Millroth [BM87] discuss dealing with alien ("hairy") data structures in PROLOG; their techniques bear some similarity with our sealed envelopes. A sound treatment of dynamic typing in an extension of SML is described in [ACPP89]. Our type union is very similar to their type dynamic; however, we also consider partially instantiated and lazily evaluated expressions. In contrast, their treatment includes a typecase expression permitting dynamic type testing within the source language. The work most vitally related to ours is that of Phil Wadler [Wad89], who derives theorems about functions working simply from their type signatures, and that of Mary Sheeran, who has applied category theoretical interpretations to type signatures for similar results.

The issue of types in logic programming has been studied by many authors (see [Red88] and [Pfe90] for recent surveys). The approaches can be classified as *prescriptive* (e.g. [Han89]), where type declarations restrict the success set of the program, and *descriptive*, where types (declared as e.g. in [MO85] or inferred as in e.g. [Mis84]) describe properties of the success set of the program. (A recent paper [LR91] reconstructs the Mycroft-O'Keefe type system as a prescriptive one.) Types of the predicates of a logic program are in the focus of attention of both categories. Introduction of types is often motivated by their potential for early detection of errors and by their usefulness for program analysis and optimization. Our work has different motivations and objectives. We assume that the external functional procedures used in our logic programs are typed. We do not care whether these types have been inferred or declared by the user. We leave as the topic of future work the question how to use them for inferring types of the predicates. Thus our predicates are not typed. Our main objective is the use of the types of the functional procedures for interfacing them with our logic programs. Types provide the only source of information about the external procedures which are otherwise considered black boxes. As shown in the paper this information may often be sufficient to know that an external procedure can safely be invoked with non-ground arguments. This allows for improvement of the operational semantics of logic programs with external procedures described in [BM88]: some error denotations under S-unification can be avoided by using instead our P-unification.

Our language allows for the use of higher-order functional procedures but the syntax of terms is restricted to applicative terms and λ-abstraction is not allowed in our logic programs. Higher-order features are commonly supported by functional programming languages and can be used from logic programs by the interface based on P-unification. In this way we avoid full higher-order unification which is required for clean integration of higher-order features in logic programming, as exemplified by λ-Prolog [NM88]. However, as pointed out in [Mil90] many interesting λ-Prolog programs can be executed with a restricted kind of higher-order unification. In contrast to λ-Prolog our functional procedures are external and the only information about their behavior is given by their type signatures. This causes inherent incompleteness of P-unification, which may result in unresolved deferred disagreement pairs.

16 Conclusions

We have employed SML: (i) on *alien data types* (e.g. logical variables); (ii) in a *lazy* manner, even though it is strict; (iii) on a *dynamically typed and untyped* terms, even though it is *statically typed* — all without changing a single bit of its *real* (New Jersey) implementation. We view this exercise as an initial experiment in using types as a basis for securely interfacing “open languages.” A particularly challenging long term goal is to interface logic (and functional) languages to object oriented imperative languages, which have rather different, but equally advanced, notions of polymorphism [CCH+89].

17 Acknowledgments

The insights of Staffan Bonnier, Laurie Hannon, and Charles Clark have been very helpful in the preparation of this paper.

References

[ACPP89] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. Dynamic typing in a statically typed language. Technical Report 47, DEC Systems Research Center, June 10, 1989.

[BM87] Jonas Barklund and Hakan Millroth. Integrating complex data structures in Prolog. In *Symposium on Logic Programming*, pages 415–425, San Francisco, August 1987. IEEE Computer Society.

[BM88] Steffan Bonnier and Jan Małuszyński. Towards a clean amalgamation of logic programs with external procedures. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 311–326, Seattle, 1988. MIT Press.

[Bon91] Staffan Bonnier. Unification in incompletely specified theories: a case study. In A. Tarlecki, editor, *Mathematical Foundations of Computer Science 1991*, pages 84–92. Springer-Verlag, 1991. LNCS 520.

[Boy91] Johan Boye. S-SLD-Resolution: An operational semantics for logic programs with external procedures. In Jan Maluszynski and Martin Wirsing, editors, *Proc. PLILP'91*, pages 383–394. Springer-Verlag, 1991. LNCS 528.

[CCH$^+$89] Peter Canning, William Cook, Walt Hill, John Mitchell, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *Proc. of Conf. on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989. Also Technical Report STL-89-5, Hewlett-Packard Labs.

[DV87] M. Dincbas and P. Van Hentenryck. Extended unification algorithms for the integration of functional programming into logic programming. *Journal of Logic Programming*, 4(3):199–227, 1987.

[Han89] Michael Hanus. Horn clause programs with polymorphic types: Semantics and resolution. In *Proc. of TAPSOFT'89*, pages 225–240, 1989. Springer LNCS 352.

[KK91] A. Kågedal and F. Kluzniak. Enriching Prolog with S-Unification. In *Proc. of Phoenix Seminar on Declarative Programming*. Springer-Verlag, 1991.

[LR91] T.K. Lakshman and U.S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In *Proc. of ILPS*, pages 202–217. MIT Press, San Diego, 1991.

[Mil78] R. Milner. A theory of type polymorphism. *J. of Comp. and Sys. Sci.*, 17(3):348–375, 1978.

[Mil90] Dale Miller. A logic programming language with lambda-abstraction function variables, and simple unification. Technical Report MS-CIS 90-54, University of Pennsylvania, Philadelphia, 1990.

[Mis84] Prateek Mishra. Towards a theory of types. In *Symp. on Logic Programming*, pages 289–298, Atlantic City, 1984. IEEE.

[MO85] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3), August 1985.

[Myc84] Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *International Symposium on Programming*, pages 217–228. Springer-Verlag LNCS 167, 1984.

[NM88] Gopalan Nadathur and Dale A. Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming Proceedings of the Fifth International Conference and Symposium.* IEEE Computer Society, MIT Press, August 1988.

[Ogi90] Takeshi Ogi. Using types to interface functional and logic programming. MS thesis, University of Utah, May 1990.

[Pfe90] Frank Pfenning. Types in Logic Programming. Tutorial Notes, Symposium on Logic Programming, Jerusalem, June 1990.

[Red88] Uday Reddy. Notions of polymorphism for predicate logic programs. In Robert Kowalski and Kenneth Bowen, editors, *Proc. 5th Int. Conf. and Symp. on Logic Programming*, Seattle, 1988. MIT Press.

[Wad89] Phil Wadler. Theorems for free! In David MacQueen, editor, *Proc. Symposium on Functional Languages and Computer Architecture*, London, September 11-13 1989. Springer-Verlag.