An ideal model for recursive polymorphic types

David MacQueen

Bell Laboratories
Murray Hill, New Jersey 07974

Gordon Plotkin

Department of Computer Science
University of Edinburgh
Edinburgh EH9 3J2

Ravi Sethi

Bell Laboratories
Murray Hill, New Jersey 07974

1. The setting

We will consider *types* as somehow being or generating *constraints* on expressions in a language. A consistent type discipline will ensure that any expression satisfying the constraints will not produce a run-time error. For example during any evaluation of the expression $f(x)$, the value of f must be a function; otherwise, a run-time error occurs because the computation cannot proceed. As we want to guarantee the absence of such run-time errors, the constraints on the values of f and x must be spelled out further. In particular, if the value of x satisfies constraint s, then it suffices that f is a function that is applicable to all such values. For instance, f might satisfy the constraint of sending all values satisfying s to values satisfying another constraint t. These constraints on the values of f and x will be written as:

$$x : s$$

$$f : s \rightarrow t$$

It follows from the above constraints that the value of $f(x)$ satisfies t. This inference can be written as the rule:

$$\frac{f : s \rightarrow t \quad x : s}{f(x) : t}$$

Inferences like the above (i.e. $f(x) : t$) will be made using a formal system of axioms and rules in which terms like s, t, and $s \rightarrow t$ are called *type expressions*, or simply *types*. One advantage of the formal rules is that they allow type inference to be studied separately from the underlying intuition of types being sets of values. Consider for example the expression $x(x)$. Reasoning as for $f(x)$ above,

$x \ : \ s$

$x \ : \ s \rightarrow t$

It is natural to formulate a system of rules in which we equate the constraints on x, leading to $s = s \rightarrow t$, and we use the notation $\mu s.s \rightarrow t$ to formally denote a solution of this equation. Morris [16, pp. 122-124] observes that such recursive or circular types allow types to be inferred for combinators like **Y** (see Section 3). However, it is nontrivial to model constraints as *sets* of values so that there is a set of values s satisfying the equality $s = s \rightarrow t$. We will present a model in which almost all such equations have solutions, and these solutions will be unique.

Our semantic model of types, essentially that described in [11] (see also [13, 21, 22]), was developed to explain the implicit form of polymorphism based on type quantification, as used in the programming languages ML [6] and Hope [3]. It is a formalization of the naive view of types as sets of values; a value x has a type s if x is a member of the set of values modeling s. Other models (e.g. [12]), which combine recursion with the explicit form of polymorphism expressed in terms of type parameters [18], do not lend themselves to such an intuitive interpretation. The main technical innovation of this paper is the use of a metric structure on types to establish the existence and uniqueness of solutions of most recursive type equations.

2. An aside on self application

The expression $x(x)$ is an example of a *self application*, because x is applied as a function to itself. Self application is essential to the treatment of recursion in the lambda-calculus; this is sufficient motivation for studying the type checking of expressions containing self application.

It may however be helpful to give an example showing that pure self application also occurs in languages like Pascal, C, and Lisp that allow function parameters with incomplete or nonexistent type specifications. The following definition of the factorial function in Pascal uses self application of the auxiliary function f and its parameter function g.

```
function factorial(n: integer): integer;
    function f(function g: integer): integer;
    var m: integer;
    begin
         m := n;
         if m = 0 then f := 1
              else begin n := n - 1; f := m * g(g) end
    end;
begin
    factorial := f(f)
end;
```

The types of both f and g could be given by $\mu s.s \rightarrow$ **int**, but in fact the function compiles because the parameter types of function parameters are not checked (in the original definition of Pascal).

The language in this paper is based on the lambda calculus, but the results are applicable to imperative languages as well. Henderson [7] relates type checking of most programming language constructs to type checking of a functional language.

3. Examples

The examples in this section suggest the type inferences we would like to make.

Similar to the expressions $f(x)$ and $x(x)$ mentioned above, is the "nonsensical" expression $3(x)$. The type of the subexpression 3 is **int**, the type of x can be represented by a variable s, but it is not possible to infer a type for $3(x)$ since an integer cannot be applied as a function. The remaining examples here consider expressions for which types can be found.

We return to the expression $x(x)$ since many of the questions addressed in this paper can be discussed in connection with it. Recall that the type constraints on the two instances of x lead to the type $s = s \rightarrow t$ for x. The discussion of $x(x)$ extends to the larger expression $\lambda x.xx$.[1] Since the type of x is $s = s \rightarrow t$, xx has type t, $\lambda x.xx$ has type $s = s \rightarrow t$, and the type of $(\lambda x.xx)(\lambda x.xx)$ is t.

Two remarks provide some perspective on the above discussion. (1) Since there are no constraints on the type variable t, the above discussion applies with any type expression substituted for t. Thus, $(\lambda x.xx)(\lambda x.xx)$ has type σ for any type expression σ. Since the meaning of $(\lambda x.xx)(\lambda x.xx)$ is $\bot$, it follows that $\bot$ has every type. The sets of values used to model types must therefore always be nonempty. Moreover, nonterminating expressions can have well defined types. (2) Any *pure* lambda expression has type $\mu t.t \rightarrow t$, i.e. $t = t \rightarrow t$, since the expression can be used either as a function or as an argument. It follows that constants like 3 are needed to construct expressions like $3(x)$ that do not have types.

A type can also be inferred for the **Y** combinator [16]. Recall that

$$\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

The first few lines are familiar by now:

x :	$s = s \rightarrow t$	
xx :	t	
f :	$t \rightarrow t'$	from $xx : t$ and $f(xx)$
$f(xx)$:	t'	
$\lambda x.f(xx)$:	$s \rightarrow t'$	from lambda abstraction

The self application in the expression $(\lambda x.f(xx))(\lambda x.f(xx))$ results in its type being $s = s \rightarrow t'$. Faced with the equalities

$$s \rightarrow t \;=\; s \;=\; s \rightarrow t'$$

we equate t and t'. Then $f : t \rightarrow t$ and (as might be expected)

$$\mathbf{Y} : (t \rightarrow t) \rightarrow t$$

4. Relation to type checking

In order to place the above discussion on a precise footing, we need a formal system of axioms and rules to infer types. We sketch such a system in Section 7. Finding an algorithm for discovering types that can be inferred from the rules is a separate problem. It is desirable for an algorithm to discover the most general type for an expression. For example, the **Y** combinator satisfies the constraint $(t \rightarrow t) \rightarrow t$ for any type t. A less general statement is that **Y** satisfies the constraint $(\mathbf{int} \rightarrow \mathbf{int}) \rightarrow \mathbf{int}$. Type expressions like $(t \rightarrow t) \rightarrow t$, containing type variables, are called *type schemes* following Hindley [8]. The type scheme discovered by an algorithm is *principal* if it is the most general type scheme that can be inferred for the expression from the rules. For the type systems of [8, 13, 16], unification [19] can be used to construct linear algorithms for discovering principal type schemes [4]. We do not address the existence of principal type schemes in the presence of recursive types.

However, it has been observed (e.g. [14]) that recursive types can be discovered using "circular" unification in which a type variable can be unified with a term containing it. In this way, the appropriate type can be found for the **Y** combinator. (Such an algorithm has been implemented for Scheme, which is a dialect of Lisp [23].) Algorithms for "circular" unification [15] can readily be adapted from algorithms for testing the equivalence of (1) finite automata and (2) linked lists with cycles [10, Section 2.3.5, Exercise 11]. The almost linear algorithm for testing

[1] As usual, function application is indicated by juxtaposition and associates to the left: both $f(x)y$ and fxy are equivalent to $(f(x))(y)$.

equivalence of finite automata in [9] can be viewed as an implementation of the sketch in [10, p. 594].

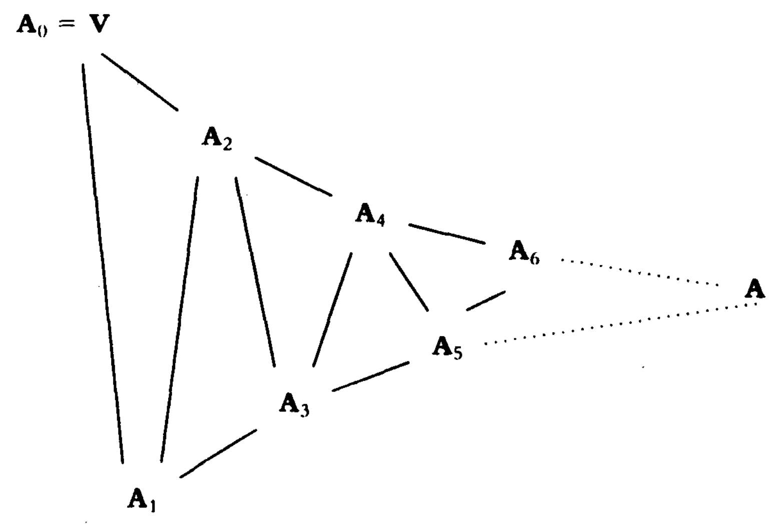

Figure 1. The sequence $\mathbf{A}_0$, $\mathbf{A}_1$, $\cdots$ converges to $\mathbf{A}$, yielding a solution to the equation $s = s \rightarrow t$. Lines indicate inclusions between sets.

5. Informal types for self application

The constraint $f : s \rightarrow t$, requires f to map all values satisfying s to values satisfying t. If the constraint s is weakened to s', then, informally, s' denotes a larger set than s. Weakening s has the opposite effect on the type of a function from s to t, because $f : s' \rightarrow t$ becomes a stronger constraint: f is required to map a larger set to values satisfying t.

Intuition on the role of $\rightarrow$ can be provided by considering a particular sequence of sets that arises in connection with the equality $s = s \rightarrow t$. The semantic counterpart of the operator $\rightarrow$ on types is the operator $\boxminus$ on sets of values modeling types. Informally, $\mathbf{D} \boxminus \mathbf{E}$ is the set of all functions that map elements of $\mathbf{D}$ to elements of $\mathbf{E}$.

In keeping with the view of types as sets, let $\mathbf{B}$ be the set of values modeling the type t. Starting with the set $\mathbf{V}$ of all values, we estimate the set $\mathbf{A}$ modeling s by writing the sequence:

$$\begin{aligned}
\mathbf{A}_0 &= \mathbf{V} \\
\mathbf{A}_1 &= \mathbf{A}_0 \boxminus \mathbf{B} = \mathbf{V} \boxminus \mathbf{B} \\
\mathbf{A}_2 &= \mathbf{A}_1 \boxminus \mathbf{B} = (\mathbf{V} \boxminus \mathbf{B}) \boxminus \mathbf{B} \\
\mathbf{A}_3 &= \mathbf{A}_2 \boxminus \mathbf{B} = (\,(\mathbf{V} \boxminus \mathbf{B}) \boxminus \mathbf{B}\,) \boxminus \mathbf{B}
\end{aligned}$$

Since $\mathbf{A}_0$ is the entire set $\mathbf{V}$ of values, $\mathbf{A}_1$ consists of functions that map all values in $\mathbf{V}$ to elements of $\mathbf{B}$ – a fairly restrictive condition. Since $\mathbf{A}_1$ must be a subset of $\mathbf{A}_0 = \mathbf{V}$, more functions belong to $\mathbf{A}_1 \boxminus \mathbf{B}$ than to $\mathbf{A}_0 \boxminus \mathbf{B}$. Therefore, $\mathbf{A}_2$ is a larger set than $\mathbf{A}_1$. The inclusions we get are (see Figure 1):

$$\mathbf{A}_0 \supseteq \mathbf{A}_2 \supseteq \mathbf{A}_4 \supseteq \cdots$$

$$\mathbf{A}_1 \subseteq \mathbf{A}_3 \subseteq \mathbf{A}_5 \subseteq \cdots$$

Fortunately, it can be shown that the limits of the even and odd sequences are the same, so there is a unique set $\mathbf{A}$ modeling the type satisfying the equality $s = s \rightarrow t$. However, the techniques used to show this result abandon the approach based on the convergence of the unions and intersections of nested sequences. Instead, convergence is established using a metric on sets modeling types. The progression is as follows:

1. We begin with the space of values **V** used to give the semantics of a lambda-calculus based language.
2. The informal notion of a type as a set of values is made precise by considering certain subsets of **V**. These subsets model collections of structurally similar values, where the term "structure" refers to notions like being a function, or being a pair.
3. The solution of recursive type equations is facilitated by considering the convergence of particular sequences of types. Note that the sequence converging to **A** in Figure 1 is neither monotonically increasing nor decreasing. Convergence of sequences cannot therefore be proved using monotonicity properties. Instead, we define a metric that measures the distance between types.
4. The Banach Fixed Point theorem [2] can be invoked to show the existence of unique fixed points for "contractive" functions on metric spaces in which limits exist. We show that this theorem can be applied to the metric space of types.

Metric spaces have been used previously to investigate the semantics of nondeterministic and parallel recursive programs [1, 5], where the main application was to obtain the metric completion of a space generated by finite elements. In our case, we start with a complete metric space of types and obtain fixed points by using the Banach Fixed Point theorem.

6. Semantics of type expressions

6.1. *Domains.* The space of values used to interpret the expressions of our language is **V**, which has an isomorphism:

$$\mathbf{V} \cong \mathbf{T} + \mathbf{N} + (\mathbf{V} \to \mathbf{V}) + (\mathbf{V} \times \mathbf{V}) + (\mathbf{V} + \mathbf{V}) + \{\mathbf{wrong}\}_\perp \qquad (6.1)$$

This can be read as saying that **V** is (isomorphic to) the sum of the truth values **T**, the integers **N**, continuous functions from **V** to **V**, the product of **V** with itself, the sum of **V** with itself, and a value **wrong** standing for (dynamic) type-errors. The mathematics needed to solve equations like (6.1) is due essentially to Scott [20]. Details may be found in many places, such as [17].

Solutions to equations like (6.1) are particular partially ordered sets: a *complete partial order* (*cpo*) $(D,\sqsubseteq)$ consists of a set D and a partial order $\sqsubseteq$ on D, such that (i) there is a least element $\perp$ in $(D,\sqsubseteq)$, and (ii) each increasing sequence $x_0 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$ has a least upper bound (lub) $\bigsqcup_{n \geq 0} x_n$.

It will be necessary to know much more of the structure of **V** than just that it is a cpo (in order to define a metric on sets modeling types). Well behaved cpos have two kinds of elements: *finite* elements; and *limit* elements, which are lubs of increasing sequences of finite elements.[2] The finite elements in any subset, X, of a cpo are denoted by X°. The cpos we consider are called *domains*; they have a countable number of finite elements.[3]

6.2. *Ideals.* Type expressions will be interpreted using certain subsets of **V**, called *ideals* [13, 21, 22, 11]. Recall from the summary at the end of Section 6 that we think of a *type* as a collection of structurally similar values. The structural distinctions that types are meant to capture satisfy the following basic principles; (1) structure is preserved as we go "downward" to approximations, and (2) structure is preserved when we go "upward" to limits of ascending

[2] An element of a cpo is ω-*finite* if and only if whenever it is less than the lub of an increasing sequence it is less than some element of the sequence. A set X is *directed* if every finite subset of X has an upper bound in X. A cpo is ω-*algebraic* if and only if it has countably many ω-finite elements and given any element, the set of ω-finite elements less than it is directed and has that element as its least upper bound. ω-algebraic cpos have lubs of arbitrary directed sets (sometimes cpos are taken to be partial orders with such lubs and $\perp$); the ω-finite elements are even *finite*, meaning that when one is below the lub of a directed set it is below some element.

[3] A cpo is a *domain* if and only if it is consistently complete and ω-algebraic. A cpo D is *consistently complete*, if any consistent subset of D has a least upper bound; here $X \subseteq D$ is consistent if it has an upper bound in D, that is there is a $y \in D$ such that $x \sqsubseteq y$ for all x in X.

sequences of values. These notions are made precise in the definition of ideals. For technical reasons the definition is in two stages: a subset I of some partial order P is an ***order-ideal*** if and only if

1. $I \neq \emptyset$
2. $\forall y \in I.\ \forall x \in D.\ x \sqsubseteq y \supset x \in I$

A subset I of a domain D is an *ideal* if and only if it is an order-ideal satisfying the additional constraint:

3. $\forall$ increasing sequences $<x_n>$ $(\forall n.\ x_n \in I) \supset \bigsqcup x_n \in I$

That is, ideals are the nonempty left closed sets closed under lubs of increasing sequences. Nonemptiness is needed because $\perp$ has every type (see Section 4). We write $\mathcal{I}_0(P)$ for the order-ideals of a partial order P and $\mathcal{I}(D)$ for the ideals of a domain D.

Ideals are determined by their finite elements. Regarding D° as a partial order (inherited from D) and ordering ideals by subset we find:

PROPOSITION. *The correspondence* $I \mapsto I^\circ$ *is an isomorphism of* $<\mathcal{I}(D),\subseteq>$ *and* $<\mathcal{I}_0(D^\circ),\subseteq>$ *with inverse* $J \mapsto \{\bigsqcup a_n \mid <a_n>$ an increasing sequence in $J\}$ □

This proposition allows us to restrict attention to finite elements while comparing ideals.

6.3. *Metric space of ideals.* The idea here is to solve recursive type equations by structuring the ideals as a complete metric space. The distance between two ideals will be measured via a notion of the smallest *rank* of a finite element in one but not the other. The rank function will be left unspecified for the moment (except that it maps finite elements to natural numbers). If I and J are ideals then a *witness* for I and J is any finite element that is in I but not in J or vice versa. The *closeness* $c(I,J)$ of I and J is the least possible rank of a witness for I and J, and if none exists it is ∞.

Given such an closeness function, one can define a metric d that measures the distance between two ideals. Here we take $d(I,J) = 2^{-c(I,J)}$ where, by convention, $2^{-\infty} = 0$. This is even an *ultrametric* meaning that

$$d(I,K) \leq \max(d(I,J),d(J,K))$$

holds, which is stronger than the triangle inequality.

A sequence of ideals $<I_i>_{i\geq 0}$ is called a *Cauchy sequence* if given any $\epsilon>0$ there exists n such that for all i,j larger than n, $d(I_i,I_j)<\epsilon$. A metric space is *complete* if every Cauchy sequence converges.

THEOREM 1. *The metric space of ideals is complete.* □

At first sight this theorem is a little surprising given the arbitrary nature of the rank function. But note, for example, that if the rank function is constant then the only Cauchy sequences are those which are eventually constant.

6.4. *Rank of an element.* In order to apply this result to $\mathbf{V}$ we need to construct a rank function and consider its properties. The domain $\mathbf{V}$ is constructed by a limiting process using a chain of domains $\mathbf{V}_n$ starting from $\mathbf{V}_0 = \{\perp\}$:

$$\mathbf{V}_{n+1} = \mathbf{T} + \mathbf{N} + (\mathbf{V}_n \to \mathbf{V}_n) + (\mathbf{V}_n \times \mathbf{V}_n) + (\mathbf{V}_n + \mathbf{V}_n) + \mathbf{W}$$

The rank of a finite element is taken to be the first place it appears in the chain.

6.5. *Unique fixed points.* In order to find ideals satisfying such equations as $I = I \boxminus \mathbf{N}$ we use the Banach Fixed-Point Theorem [2]. A (*uniformly*) *contractive* map $f{:}X \to Y$ of metric spaces is one such that there is a real number $0 \leq r < 1$ such that for all x and x' in X, we have

$$d(f(x),f(x')) \leq r\,d(x),$$

and it is *non-expansive* if this holds but with $r \leq 1$. The generalization to n-variable functions requires

$$d(f(x_1, \ldots, x_n), f(x'_1, \ldots, x'_n)) \leq r \max\{d(x_i,x'_i) \mid 1 \leq i \leq n\}$$

The Banach Fixed-Point Theorem states that if X is a nonempty complete metric space and $f:X \to X$ is contractive then it has a unique fixed point, namely $\lim_{n \to \infty} f^n(x_0)$ where x_0 is any point in X.

6.6. *Contractive maps.* In order to apply the Banach Fixed-Point Theorem to determine ideals modeling recursive types, we need to consider the contractiveness of maps on ideals. Some care is needed, since union and intersection have the weaker property of being nonexpansive.

Auxiliary maps. The projection functions $\pi_i : X_1 \times \cdots \times X_n \to X_i$ are not contractive but are non-expansive. The composition of a map $f:X_1 \times \cdots \times X_n \to Y$ with $g_i : V_1 \times \cdots \times V_m \to X_i$ is non-expansive if f and all g_i are; if f is contractive and all g_i are non-expansive then the composition is contractive and this holds also if all the g_i are contractive and f is nonexpansive. Finally, we note that when Y is an ultrametric space then a map $f:X_1 \times \cdots \times X_n \to Y$ as above is contractive (non-expansive) iff it is contractive (non-expansive) in each argument taken separately.

PROPOSITION. *Intersection and union are not contractive but are non-expansive, considered as binary functions over ideals.* □

Type constructors. We define three binary functions on ideals corresponding to the three domain constructions. The *sum* $I \boxplus J$ of two ideals is defined by injecting I and J into $\mathbf{V} + \mathbf{V}$ and taking the union of the injections. The *product* $I \boxtimes J$ of two ideals I and J is simply $I \times J$. The *exponentiation* (*function-ideal*) is defined by:

$$I \boxminus J = \{f \in \mathbf{V} \to \mathbf{V} \mid f(I) \subseteq J\}$$

It is straightforward to show that, when viewed as a subset of $\mathbf{V}$, each of these sets is an ideal. The idea behind the definition of the function-ideal appears in many papers (see [13, 11] for example). The next theorem is central to the results of this paper.

THEOREM 2. *All three functions, sum, product, and exponentiation, are contractive.* □

The basic idea behind the proof of this theorem is that two distinct compound ideals (e.g. $I \boxtimes J$ and $I' \boxtimes J'$) have a compound witness of least rank (e.g. $\langle i,j \rangle \in I \boxtimes J - I' \boxtimes J'$) whose components are simpler (i.e. lower rank) witnesses to the differences between the component ideals (e.g. $i \in I - I'$ and $j \in J - J'$). This implies that the compound elements are closer together than their components.

Note that this theorem would fail if we kept the same definition of exponentiation but allowed arbitrary sets. Since $(\emptyset \boxminus \emptyset) = \mathbf{V}$ and $(\mathbf{V} \boxminus \emptyset) = \emptyset$, exponentiation is not then contractive in its first argument.

Quantification. Let $\mathcal{I}(D)$ denote the collection of all ideals. Suppose $f:\mathcal{I}(D)^{n+1} \to \mathcal{I}(D)$ is a function of $n+1$ arguments. Then we can produce a function of n arguments by "quantifying" its first argument. The *universal quantification* of f relative to a given collection of ideals $\mathcal{K} \subseteq \mathcal{I}(D)$ is defined by:

$$(\forall_{\mathcal{K}} f)(J_1, \ldots, J_n) = \bigcap_{I \in \mathcal{K}} f(I, J_1, \ldots, J_n)$$

and the *existential quantification* by:

$$(\exists_{\mathcal{K}} f)(J_1, \ldots, J_n) = \bigsqcup_{I \in \mathcal{K}} f(I, J_1, \ldots, J_n)$$

It is here that fixing on a particular collection of sets as the types – the ideals – makes a difference to the *definition* of our operations, since it affects the range of variation of the ideal I in the above definitions.

THEOREM 3. *If $f:\mathcal{I}(D)^{n+1} \to \mathcal{I}(D)$ is contractive (non-expansive) in its last n arguments, so are its universal and existential quantification.* □

Fixed points. Our last construction makes sense in a general setting. Let $f:X \times Y_1 \times \cdots \times Y_n \to X$ be a function of non-empty complete metric spaces which is contractive in its first argument. Define the "parameterized fixed-point" function $\mu f : Y_1 \times \cdots \times Y_n \to X$ by taking $(\mu f)(y_1, \ldots, y_n)$ to be the unique element, x, of X such that

$x = f(x, y_1, \ldots, y_n)$ as guaranteed by the Banach Fixed-Point Theorem.

THEOREM 4. *If f is contractive (non-expansive) so is μf.* □

The functions shown to be contractive/non-expansive in the above theorems are the semantic counterparts of constructors appearing in the type expressions below. The results of this section will be applied to show that the semantics of type expressions is well defined.

6.7. *Semantics of type expressions.* The syntax of type expressions is given by an abstract syntax grammar. The set of type variables **TVar** is ranged over by t, and the grammar is given by

$$\sigma ::= \mathbf{int} \mid \mathbf{bool} \mid t \mid \sigma \to \sigma \mid \sigma \times \sigma \mid \sigma + \sigma \mid \sigma \cap \sigma \mid \sigma \cup \sigma \mid \forall t.\sigma \mid \exists t.\sigma \mid \mu t.\sigma$$

In fact, we cannot allow all such expressions since we can only give meaning to $\mu t.\sigma$ when σ denotes a contractive function of t. So say σ is (*formally*) *contractive* in t iff one of the following conditions hold:

1. σ has one of the forms **int**, **bool**, t' (with $t' \neq t$), $\sigma_1 \to \sigma_2$, $\sigma_1 \times \sigma_2$, or $\sigma_1 + \sigma_2$.
2. σ has one of the forms $\sigma_1 \cap \sigma_2$ or $\sigma_1 \cup \sigma_2$ with both σ_1 and σ_2 contractive in t.
3. σ has one of the forms $\forall t'.\sigma_1$, $\exists t'.\sigma_1$, or $\mu t'.\sigma_1$ with either $t' = t$ or σ_1 is contractive in t.

Now we take **TExp** to be the set of *well formed* type expressions where σ is *well formed* iff one of the following conditions hold.

1. σ is **int**, **bool**, or t'.
2. σ has one of the forms $\sigma_1 \to \sigma_2$, $\sigma_1 \times \sigma_2$, $\sigma_1 + \sigma_2$, $\sigma_1 \cap \sigma_2$, or $\sigma_1 \cup \sigma_2$ with both σ_1 and σ_2 well formed.
3. σ has one of the forms $\forall t.\sigma_1$ or $\exists t'.\sigma_1$ with σ_1 well formed.
4. σ has the form $\mu t.\sigma_1$ with σ_1 well formed and contractive in t.

For the semantics we define the semantic function

$$\mathcal{T}: \mathbf{TExp} \to \mathbf{TEnv} \to \mathcal{I}(\mathbf{V})$$

where $\mathbf{TEnv} = \mathbf{TVar} \to \mathcal{I}(\mathbf{V})$ is the set of *type environments* ranged over by ν. The definition is by structural induction and below $\mathcal{K} \subseteq \mathcal{I}(D)$ is the collection of ideals not containing **wrong**.

$$\begin{aligned}
\mathcal{T}[\![\mathbf{int}]\!]\nu &= \mathbf{N} \\
\mathcal{T}[\![\mathbf{bool}]\!]\nu &= \mathbf{T} \\
\mathcal{T}[\![t]\!]\nu &= \nu[\![t]\!] \\
\mathcal{T}[\![\sigma_1 \to \sigma_2]\!]\nu &= \mathcal{T}[\![\sigma_1]\!]\nu \boxminus \mathcal{T}[\![\sigma_2]\!]\nu \\
\mathcal{T}[\![\sigma_1 \times \sigma_2]\!]\nu &= \mathcal{T}[\![\sigma_1]\!]\nu \boxtimes \mathcal{T}[\![\sigma_2]\!]\nu \\
\mathcal{T}[\![\sigma_1 + \sigma_2]\!]\nu &= \mathcal{T}[\![\sigma_1]\!]\nu \boxplus \mathcal{T}[\![\sigma_2]\!]\nu \\
\mathcal{T}[\![\sigma_1 \cap \sigma_2]\!]\nu &= \mathcal{T}[\![\sigma_1]\!]\nu \cap \mathcal{T}[\![\sigma_2]\!]\nu \\
\mathcal{T}[\![\sigma_1 \cup \sigma_2]\!]\nu &= \mathcal{T}[\![\sigma_1]\!]\nu \cup \mathcal{T}[\![\sigma_2]\!]\nu \\
\mathcal{T}[\![\forall t.\ \sigma]\!]\nu &= \forall_{\mathcal{K}}(\lambda I \in \mathcal{I}(D).\ \mathcal{T}[\![\sigma]\!]\ (\ \nu[I/t]\ )\ ) \\
\mathcal{T}[\![\exists t.\ \sigma]\!]\nu &= \exists_{\mathcal{K}}(\lambda I \in \mathcal{I}(D).\ \mathcal{T}[\![\sigma]\!]\ (\ \nu[I/t]\ )\ ) \\
\mathcal{T}[\![\mu t.\ \sigma]\!]\nu &= \mu(\lambda I \in \mathcal{I}(D).\ \mathcal{T}[\![\sigma]\!]\ (\ \nu[I/t]\ )\ )
\end{aligned}$$

THEOREM. *The semantic function $\mathcal{T}$ is well defined.* □

We prove by structural induction on σ that: (1) for all ν, $\mathcal{T}[\![\sigma]\!]\nu$ is well defined; (2) for any t, $\lambda I \in \mathcal{I}(D).\ \mathcal{T}[\![\sigma]\!]\ (\ \nu[I/t]\ )$ is non-expansive, and is contractive if σ is contractive in t. The results of the last section make such a proof quite straightforward. It is also straightforward to prove that provided $\nu(t)$ does not contain **wrong** for any t then neither does $\mathcal{T}[\![\sigma]\!]\nu$.

7. Rules for type inference

The rules for type inference are keyed to the syntax of type expressions. The rule for function application is essentially like that for $f(x)$ in Section 1, except that constraints like $x : s$ and $f : s \rightarrow t$ are relative to an assignment $\mathcal{A}$ of types for the free variables in an expression: written $\mathcal{A} \vdash x : s$ and $\mathcal{A} \vdash f : s \rightarrow t$. If e and e' are expressions, the rule for function application is as follows (note that type expressions σ and τ are permitted instead of the type variables s and t):

$$\frac{\mathcal{A} \vdash e : \sigma \rightarrow \tau \quad \mathcal{A} \vdash e' : \sigma}{\mathcal{A} \vdash e(e') : \tau}$$

The turnstile symbol $\vdash$ denotes a *well typing* relationship between type assignments, value expressions, and type expressions that is defined by the inference rules: $\mathcal{A} \vdash e : \sigma$ holds if and only if there is a proof of it using the rules. Rules tend to come in pairs, corresponding to the introduction or elimination of a type construct. For instance, the above function application rule could be called the $\rightarrow$ elimination rule, and its complement is the following $\rightarrow$ introduction rule for typing lambda abstractions

$$\frac{\mathcal{A}, x{:}\sigma \vdash e : \tau}{\mathcal{A} \vdash \lambda x.e : \sigma \rightarrow \tau}$$

The rules for the usual type constructs, including universal quantification, are fairly standard, following those of Hindley [8] and Milner [13, 4]. The rules for existential quantification are novel, but are beyond the scope of this paper.

To deal with recursive type expressions, we add the following two rules that correspond respectively to "unwinding" and "winding" the recursive type

$$\frac{\mathcal{A} \vdash e : \mu t.\sigma}{\mathcal{A} \vdash e : \sigma[\mu t.\sigma/t]}$$

$$\frac{\mathcal{A} \vdash e : \sigma[\mu t.\sigma/t]}{\mathcal{A} \vdash e : \mu t.\sigma}$$

As usual, we verify the soundness of these rules by proving (by structural induction on the expression e) that $\mathcal{A} \vdash e : \sigma$ implies that the value of e in any environment consistent with $\mathcal{A}$ is a member of the ideal denoted by σ.

8. Conclusion

This paper justifies the extension of the type system of [11] to include recursive types. However, in contrast to the type system of Milner [13, 4], it is difficult to decide in general whether a given expression has a given type. It can be shown that this is a Π_1-complete question, even when restricted to terms of the pure λ-calculus and the type $\mathbf{int} \rightarrow \mathbf{int}$. It follows that no recursively enumerable axiomatic type system can be complete for the true type assertions.

On a practical level, this paper justifies the extension of unification based type checking algorithms for the type systems of [8, 13, 4] to allow circular unification. Similar algorithms can be applied to check the Algol family of languages, even though the types of procedure parameters are not specified. Note that dialects of Pascal that require full declaration of the types of procedure parameters do not allow self application to be expressed since they do not support recursive functional types.

References

1. A. Arnold and M. Nivat, "Metric interpretations of finite trees and semantics of nondeterministic recursive programs," *Theoretical Computer Science* **11**, pp. 181-205 (1980).
2. S. Banach, "Sur les opérations dans les ensembles abstraits et leurs applications aux équations intégrales," *Fund. Math.* **3**, pp. 7-33 (1922).
3. R. M. Burstall, D. B. MacQueen, and D. T. Sanella, "Hope: an experimental applicative language," *Lisp Conference*, Stanford, pp. 136-143 (August 1980).
4. L. Damas and R. Milner, "Principal type-schemes for functional programs," *Ninth Annual ACM Symposium on Principles of Programming Languages*, Albuquerque NM, pp. 207-212 (January 1982).
5. J. W. deBakker and J. I. Zucker, "Processes and the denotational semantics of concurrency," *Information and Control* **54**, pp. 70-120 (1982).
6. M. J. Gordon, A. J. Milner, and C. P. Wadsworth, *Edinburgh LCF*, Lecture Notes in Computer Science **78** (1979).
7. P. Henderson, "An approach to compile-time type checking," pp. 523-527 in *Information Processing* 77, ed. B. Gilchrist, North-Holland (1977).
8. R. Hindley, "The principal type-scheme of an object in combinatory logic," *Trans. AMS* **146**, pp. 29-60 (December 1969).
9. J. E. Hopcroft and R. M. Karp, "An algorithm for testing the equivalence of finite automata," TR-71-114, Dept. of Computer Science, Cornell Univ. (1971).
10. D. E. Knuth, *The Art of Computer Programming: Volume 1, Fundamental Algorithms*, Addison-Wesley, Reading MA (1968).
11. D. B. MacQueen and R. Sethi, "A higher order polymorphic type system for applicative languages," *1982 Symposium on Lisp and Functional Programming*, Pittsburgh PA, pp. 243-252 (August 1982).
12. N. J. McCracken, "An investigation of a programming language with a polymorphic type structure," Ph. D. Thesis, Computer and Information Science, Syracuse Univ. (June 1979).
13. R. Milner, "A theory of type polymorphism in programming," *JCSS* **17**(3), pp. 348-375 (December 1978).
14. F. L. Morris, "Automatic assignment of concrete type schemes to programs," unpublished (197?).
15. F. L. Morris, "On list structures and their use in the programming of unification," Report 4-78, School of Computer and Information Science, Syracuse Univ. (August 1978). A fast algorithm for circular unification is credited to G. Huet, G. Kahn, and J. A. Robinson.
16. J. H. Morris Jr., "Lambda-calculus models of programming languages," Ph. D. Thesis, Sloan School of Management, MIT (1968).
17. G. Plotkin, "Advanced domains," Summer School, Pisa (1978).
18. J. C. Reynolds, "Towards a theory of type structure," pp. 408-425 in *Programming Symposium, Paris, 1974*, Lecture Notes in Computer Science **19**, Springer Verlag, Berlin (1974).
19. J. A. Robinson, "A machine-oriented logic based on the resolution principle," *J. ACM* **12**, pp. 23-41 (1965).
20. D. S. Scott, "Continuous lattices," pp. 97-136 in *Lecture Notes in Mathematics* **274** (1972).
21. A. Shamir and W. W. Wadge, "Data types as objects," pp. 465-479 in *Automata, Languages and Programming, 4th Colloquium, Turku*, Lecture Notes in Computer Science 52, Springer-Verlag, Berlin (1977).
22. W. W. Wadge, *Personal communication to R. Milner*, March 1978.
23. M. Wand, *personal communication*, January 1983.