Type Checking In *Exp*: An Algebraic Approach

Jon Shultis
Oregon Graduate Center

Abstract

An algebraic approach to static analysis is introduced through an example: checking types in *Exp*, a simple applicative language. The method extends an existing body of theory about static analysis for imperative languages to include functional languages, exploiting their natural algebraic structure. The method shows how results from a variety of theoretical areas - algebraic semantics, denotational semantics, and equational reasoning - can be used to develop a flexible and conceptually simple solution to a practical problem.

Oregon Graduate Center Technical Report No. CS/E-82-03, March 8, 1982

Type Checking In *Exp*: An Algebraic Approach

Jon Shultis

Computer Science and Engineering Department
Oregon Graduate Center
19600 N. W. Walker Rd.
Beaverton, OR 97006

1. Introduction

The *static analysis* of a program traditionally consists of *control flow analysis* and *data flow analysis*. The information gleaned from such anayses is an estimate, or approximation, of some properties of the program. These estimates are used to reason about the program relative to its specification, which itself is understood statically if at all. Information about static properties is also important for translators and code improvers.

The theory of *abstract interpretation* [Cou77a] unifies classical data flow analysis for languages that have a fixed, sequential flow of control. Sequential control flow is characteristic of imperative languages like Pascal.

Applicative languages have been getting increased attention recently. One feature of such languages is their inherent logical parallelism, which makes them attractive candidates for programming loosely coupled multiprocessors [ACM81a]. Until recently, however, these languages have been implemented on conventional von Neumann computers using evaluators that employ safe sequential computation rules. Although abstract interpretation can be used to analyze the data flow of applicative languages with respect to such sequential evaluators [Jon81a], it is neither conceptually appropriate nor practically beneficial to make unnecessary assumptions about control flow for such languages. In fact, Eric Hehner has recently suggested that such assumptions are often inappropriate even for imperative languages, in that natural opportunities for parallel evaluation are overlooked[†].

This paper presents a new way of looking at static analysis problems, one that is naturally suited to the analysis of applicative languages. It exploits results from algebraic semantics [Gog77a] and work on equational reasoning [Hue80a] to compute program properties algebraically.

The method is introduced through an example taken from Milner's 1978 paper on polymorphic type checking based on his experience with ML [Mil78a]. This new approach provides some useful insights into the nature of the problem and its solution, while leading to the same results. Algebraic static analysis provides a unified and natural way to express and solve many similar problems.

2. Some Algebraic Preliminaries

An *algebra* A is a system (S,Σ,C,I) where

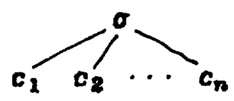
- S is a set of symbols called the *sorts* of A
- Σ is an $S^*\times S$-indexed family of sets of symbols called the *signature* of A. That is, Σ is a family of sets $\Sigma_{w,s}$, where $w\in S^*$ and $s\in S$. ε denotes the empty string.
- C is an S-indexed family of sets called the *carriers* of A.
- I is a correspondence that assigns to each $\sigma\in\Sigma_{w,s}$ a function $f_\sigma:C_{s_1}\times C_{s_2}\times\cdots\times C_{s_n}\rightarrow C_s$, where $w=s_1s_2\cdots s_n$. When $w=\varepsilon$, σ is assigned an element $c_\sigma:C_s$ and is ordinarily called a *constant*. I is called the *interpretation* of A.

[†]Personal communication.

When necessary to distinguish among algebras, symbols will be superscripted with the name of the algebra; e.g., $\Sigma^{A}_{w,s}$ is the set $\Sigma_{w,s}$ of algebra A. In an attempt to avoid truly rococo notation, I will gradually slip into the customary practice of glossing over the distinction between an algebra and its carrier, writing the name of the algebra where the carrier is intended. Similarly, the function symbol σ will be used ambiguously to denote the corresponding function f_σ. One final convention is that fx and $f(x)$ both signify the application of the function f to argument x.

As an example of an algebra, let $G = (S^G, \Sigma^G, C^G, I^G)$ as shown in figure 1. G is a familiar algebra, namely the cyclic group of order two. The reader may readily verify that e is the group identity, inv is the group inverse, and $\circ$ is associative.

As a second, somewhat more abstract, example let T_Σ be defined for any S-sorted signature Σ as follows. The carrier of sort s, C_s, is defined recursively. As a basis, $\Sigma_{\varepsilon,s} \subseteq C_s$. For the recursive step, let $\sigma \in \Sigma_{w,s}$, where $w = s_1 \cdots s_n$. Then each tree with σ as root and $c_1, \ldots, c_n$ as children, where each $c_i \in C_{s_i}$, as illustrated below, is also a member of C_s.



Under I the operation corresponding to $\sigma \in \Sigma_{w,s}$ simply gathers trees of the appropriate sorts together with σ as the root; e.g., the tree above is the result of $f_\sigma(c_1, c_2, \ldots, c_n)$. This algebra can always be constructed, and is called the algebra of Σ-trees. For a more formal development of T_Σ, see [Gog77a].

Consider next classes of algebras where the set of sorts S and the signature Σ are held fixed. That is, consider classes of *S-sorted Σ-algebras* or,

$S^G = \{ par \}$

$\Sigma^G = \{ \quad \Sigma^G_{\epsilon,par} = \quad \{ e, o \}$

$\qquad \Sigma^G_{par,par} = \quad \{ inv \}$

$\qquad \Sigma^G_{par\ par,par} = \quad \{ \circ \}$

$\quad \}$

$C^G = \{ 1, 0 \}$

$I^G = \quad o \longmapsto 1$

$\qquad e \longmapsto 0$

$\qquad inv \longmapsto \{ <0,1>, <1,0> \}$

(In this example the functions corresponding to *inv* and $\circ$ are presented as sets of <domain,range> pairs.)

$\qquad \circ \longmapsto \{ <<0,0>,0>, <<0,1>,1>, <<1,0>,1>, <<1,1>,0> \}$

Figure 1. Parity Algebra

more simply, ***Σ-algebras***.

Let $\mathbf{C}$ be the class of Σ-algebras, and let A and B belong to $\mathbf{C}$[†]. A mapping $\eta{:}A \to B$ is a ***homomorphism*** (of algebras) if

$$\eta(\sigma^A(a_1, \ldots, a_n)) = \sigma^B(\eta a_1, \ldots, \eta a_n)$$

where $\sigma \in \Sigma_{s_1 \cdots s_n, s}$ and $a_i \in C_{s_i}$ $(1 \le i \le n)$.

Notice that every homomorphism $\eta{:}A \to B$ induces an equivalence relation $\sim_\eta$ on A in the obvious way, namely:

$$a \sim_\eta a' \iff \eta a = \eta a'$$

[†]I am assuming here that a Σ-algebra has a "small" carrier; i.e., that C is the closure of the images of the constant symbols under the operations. An immediate consequence of this assumption is that all homomorphisms of algebras in $\mathbf{C}$ are epimorphisms. This assumption can also be seen as an insistence on referential transparency.

Furthermore, B is isomorphic to the quotient of A by this equivalence relation:

$$B \cong A / \sim_\eta$$

where, of course, two algebras are isomorphic if they are mutually homomorphic, and quotients are taken in the usual way.

Now, the class $\mathbf{C}$ of Σ-algebras always has one rather curious member, called (the) *initial* Σ-algebra, with the property that there is a (unique) homomorphism from the initial algebra to every algebra in $\mathbf{C}$. Thus every algebra in $\mathbf{C}$ is, effectively, a quotient of the initial algebra. The initial algebra is unique, up to isomorphism, and it is not difficult to construct. In fact, T_Σ is always an initial Σ-algebra!

The class $\mathbf{C}$ of Σ-algebras also has a *trivial* algebra, which is a quotient of every algebra in $\mathbf{C}$.

Let $\leq$ be the relation of being homomorphic; i.e., $A \leq B$ if and only if there is a homomorphism from B to A. The reader may readily verify that $\leq$ is a partial order, and that $\mathbf{C}$ is a complete lattice under $\leq$, with T_Σ as $\top_\mathbf{C}$ and the trivial algebra as $\bot_\mathbf{C}$.

3. Syntax of Exp

Let us begin by looking at the abstract syntax of Milner's toy language, *Exp*. We describe the abstract syntax of *Exp* in the usual algebraic way; i.e., abstract syntax is an initial algebra. *Exp* has a single sort:

$$S^{Exp} = \{exp\}$$

The signature of *Exp*, Σ^{Exp}, is:

$\Sigma^{Exp}_{\varepsilon,exp} = \{x, x', x'', \cdots\} \equiv \mathbf{Id}$

(Note: ε is the empty string)

$\Sigma^{Exp}_{exp,exp} = \{lambda_x, lambda_{x'}, \cdots, fix_x, fix_{x'}, \cdots\}$

The concrete syntax for $lambda_x(e)$ and $fix_x(e)$ is $\lambda x.e$ and $fix\ x.e$, respectively.

$\Sigma^{Exp}_{exp\ exp,exp} = \{apply, let_x, let_{x'}, \cdots\}$

The concrete syntax for $apply(e, e')$ is ee'.

The concrete syntax for $let_x(e, e')$ is $let\ x = e\ in\ e'$.

$\Sigma^{Exp}_{exp\ exp\ exp,exp} = \{cond\}$

The concrete syntax for $cond(e, e', e'')$ is $if\ e\ then\ e'\ else\ e''$.

The abstract syntax of *Exp* is just the initial Σ^{Exp}-algebra, $T_{\Sigma^{Exp}}$.

4. The primary semantics of Exp

The primary semantics of a programming language is intended to capture all of the behaviours that any implementation of the language is expected to exhibit. It may be the case that an implementation will have properties that are not covered by the primary semantics. Indeed, unless the primary semantic algebra is *final* in the sense of [Kam80a], there will always be some degree of choice among possible implementations. The important thing for portability is that the programmer never write a program that exploits the peculiarities of an implementation. Knowing programmers, this probably means that only proved programs will ever be portable. In any event, semantic ambiguity is probably here to stay; the problem facing designers of programming environments is to make it possible to live with it.

In [Mil78a], the semantics of *Exp* is given in the ***denotational*** style developed by Scott and Strachey [Sco71a]. In that style, the meaning of a syntactic phrase is given by an equation whose left-hand side is the syntactic phrase being defined and whose right-hand side is an expression in a *semantic model*. The semantic model is itself an algebra, and the equations define a homomorphism from the abstract syntax of the language to the semantic model. The main point of their work, however, which has not been sufficiently appreciated even by theoretically inclined computer scientists, is that with a modicum of care the right-hand sides of these equations are guaranteed to make sense; that is to say, they describe non-trivial models.

Perhaps a small digression will be allowed me here to clarify this idea with an example. In the early 1960's John McCarthy designed the famous symbolic language LISP [McC60a]. Pure LISP was based on the λ-calculus of Alonzo Church [Chu51a]. At the time, the λ-calculus had a well developed *proof theory*, thanks to the fact that Curry and Feys had finally managed to give a correct proof of the famous Church-Rosser theorem [Cur68a]. This theorem guarantees that the reduction rules for the λ-calculus are well-behaved. This in turn justified McCarthy's confidence that he could write a well-behaved computer program to perform the symbolic manipulations called for by the λ-calculus.

The fact that the system is well behaved, however, does not of itself justify the desired interpretation of λ-expressions as denoting *functions*. For that, one must show that there is a function space that models the λ-calculus. This is precisely what Scott has done, and it is no mean thing. Ever since the paradoxes of self-reference in classical mathematics drove Bertrand Russell to introduce his theory of types in the early part of this century [Rus08a], logicians and mathematicians had believed that function spaces of the kind required to model the type-free λ-calculus simply did not exist. The significance of this for computer science is that the assumption that λ-expressions denote functions, so central to the theoretical foundations of computing, was in danger of being utterly specious.

Thanks to the work of Scott and Strachey and their followers, though, we now know that some of what had been taken for granted does in fact make sense. Also, we have been shown how to tell when we are or are not making sense. Finally, we have been shown a way to compose things that are known to make sense in such a way that it is comparatively easy to show that their synthesis also makes sense. In my opinion we all owe them a great deal.

Returning now to the semantics of *Exp*, the constructions allowed for defining semantic algebras are the same in principle as those used in the denotational approach, but there is a difference in the style of presentation. In the denotational style, the meaning of a syntactic fragment is defined by displaying the result of applying that meaning to its arguments, in a manner akin to the way we were all taught to define functions in grammar school, viz.:

$$f(x)=x^2$$

As an example, take the following definition of the meaning of a variable reference:

$$E[\![v]\!]\eta = \eta v$$

Here, $E[\![v]\!]$ corresponds to f; it signifies the *meaning* of v, which is the function being defined. η is the argument of this function, and corresponds to the x in the first equation. (Remember that application of a function to its argument is sometimes indicated by juxtaposition (i.e., fx

instead of $f(x)$)). Finally, the expression on the right, ηv, conveys the result of applying the meaning of v to η.

The algebraic style is more direct. Instead of defining the function in terms of its effect, its meaning is stated outright, using an expression on the right-hand side that denotes the appropriate function. Of course, this necessitates some way of building higher-order functions. Since λ-abstraction is a familiar means of indicating higher-order functions, I shall use it here. This notation is not really algebraic; however, a development of an algebraic metalanguage is beyond the scope of this paper, so I must make some compromises. Readers wishing to see such a development should see [Shua]. In the algebraic style, then, the two equations above would be rendered as:

$$f = \lambda x.x^2$$

and

$$E\,[\![\, v \,]\!] = \lambda\eta.\eta v$$

respectively. To be complete, note that the λ above is a typed λ, and I ought to subscript it with *Env*, but for the purposes of this paper, at least, it should be clear from context what the subscripts (if any) should be.

Having taken care of these preliminaries, we shall now build a primary semantic model for *Exp*. Begin by defining the domain V of *values* to be the solution to the domain equation

$$V = B_0 \oplus B_1 \oplus \cdots \oplus F \oplus W$$

where $F = V \to V$ (the continuous functions from V to V) and $W = \{*\}$ (error). We assume that B_0 is the flat domain of truth values. $Env = \mathbf{Id} \to V$ is the domain of *environments*. $X \oplus Y$ denotes the disjoint union (sum) of the domains X and Y. The semantic model, M^{Exp}, has as its carrier M the space of continuous functions from environments to values; i.e., $M = Env \to V$.

Included among the operators in the algebra M^{Exp} are the following:

$$\begin{aligned}
&cond : T \to V \to V \to V\\
&\quad :\langle \mathbf{true}, v, v' \rangle \longmapsto v\\
&\quad :\langle \mathbf{false}, v, v' \rangle \longmapsto v'\\
&\quad :\langle \bot_T, v, v' \rangle \longmapsto \bot_V
\end{aligned}$$

$cond(t,v,v')$ is written $t \to v, v'$.

For each identifier $x \in \mathbf{Id}$, there is a function

$$\begin{aligned}
&assign_x : Env \to V \to Env\\
&\qquad :\langle \eta, x \rangle \longmapsto v\\
&\qquad :\langle \eta, y \rangle \longmapsto \eta y \text{ for } y \neq x
\end{aligned}$$

$assign_x(\eta, v)$ is written $\eta\{v/x\}$.

If U is the disjoint union of domains U_i, $i \in I$, then

$$\overset{U}{\underset{U_i}{\uparrow}}$$

is the *injection* of U_i into U, for each $i \in I$. Similarly,

$$\overset{U}{\underset{U_i}{\downarrow}}$$

is the *ejection* of U_i from U, and is defined for all $u \in U$ such that

$$u = \overset{U}{\underset{U_i}{\uparrow}} u_i$$

for some $u_i \in U_i$ and is undefined otherwise. Finally, define the *compatibility* of u with U_i to be

$$\begin{aligned}
u \mathrel{\mathbf{E}} U_i = {} & \mathbf{true} \text{ if } \overset{U}{\underset{U_i}{\downarrow}} u \text{ is defined,}\\
& \bot_T \text{ if } u = \bot_U\\
& \mathbf{false} \text{ otherwise.}
\end{aligned}$$

This notation follows closely that of [Mil78a]. Given these basic functions, make M^{Exp} into a Σ^{Exp}-algebra via the homomorphism $E : T_{\Sigma^{Exp}} \to M$, as defined by the equations in figure 2. In these equations, η ranges over Env, v

ranges over V, and μ, μ', μ'' range over M. I write $\overset{V}{\underset{W}{\uparrow}}\bullet$ as "wrong", to hint at the intuitive content of this expression. As an aid to reading the equations, bear in mind that $\mu \in M$ is the meaning of some expression, and the value $\mu\eta \in V$ is the meaning of that expression in the environment η.

Technically, these equations do not define the homomorphism E itself, only a correspondence between the operator symbols in Σ^{Exp} and suitable operations in M^{Exp}; however, it is well known that such a correspondence uniquely extends to a homomorphism from the initial algebra $T_{\Sigma^{Exp}}$ to the target algebra M^{Exp}, thus justifying our abuse of notation (see, e.g., [Bur69a]).

5. The Algebra of Types

So far, all I have done is to rephrase the syntax and semantics of *Exp* in algebraic terms. In this section I define an algebra of "types", or "functionalities", F^{Exp}, and make this algebra into an alternative semantics for $T_{\Sigma^{Exp}}$. These two semantic algebras are related in an important way: F^{Exp}

2.1	$E\,x$	$=$	$\lambda\eta.\eta x$
2.2	$E\,apply(\mu,\mu')$	$=$	$\lambda\eta.\mu\eta\ \mathbf{E}\ F \rightarrow (\mu'\eta\ \mathbf{E}\ W \rightarrow wrong,\ (\overset{V}{\underset{F}{\downarrow}}(\mu\eta))(\mu'\eta)),\ wrong$
2.3	$E\,cond(\mu,\mu',\mu'')$	$=$	$\lambda\eta.\mu\eta\ \mathbf{E}\ B_0 \rightarrow (\overset{V}{\underset{B_0}{\downarrow}}(\mu\eta) \rightarrow \mu'\eta,\ \mu''\eta), wrong$
2.4	$E\,lambda_x(\mu)$	$=$	$\lambda\eta.\overset{V}{\underset{F}{\uparrow}}(\lambda v.\mu\eta\{v/x\})$
2.5	$E\,fix_x(\mu)$	$=$	$\lambda\eta.\overset{V}{\underset{F}{\uparrow}}(\Upsilon(\lambda v.\mu\eta\{v/x\}))$
2.6	$E\,let_x(\mu,\mu')$	$=$	$\lambda\eta.\mu\eta\ \mathbf{E}\ W \rightarrow wrong,\ \mu'\eta\{\mu\eta/x\}$

Figure 2. Primary Semantics of Exp

"abstracts" M^{Exp}. Intuitively, one semantic model abstracts another if the meaning of a program in the former is consistent with that in the latter. This idea will be made precise in section 7.

The net result of all this is that to determine the type of an expression, simply *evaluate it in the model* F^{Exp}!. Hence this example illustrates a general paradigm for analyzing static properties of programs. First, build an algebraic model of the property of interest (here, F^{Exp}). Next, make the model into a Σ-algebra, and show that it is consistent with the primary semantics. A standard term reducer and unification perform the evaluation, and provide a nicely parametrized flow analyzer.

Without further fanfare, let Π_i be the flat domain containing a single proper element, π_i, and suppose there is one of these *primitive types* for each basic domain B_i. Define *Type* to be the solution to the domain equation:

$$Type = \Pi_0 \oplus \Pi_1 \oplus \cdots \oplus TF \oplus TW$$

where $TF = Type \rightarrow Type$, and $TW = \{ \nabla \}$. Injection, ejection, compatibility, replacement (on *type environments* $Tenv = \mathbf{Id} \rightarrow Type$), and cond are defined for *Type* in the obvious way. We write $\underset{TW}{\overset{Type}{\uparrow}} \nabla$ as "bad".

In addition, define the operation $map \in \Sigma^{Type}_{type\ type\ ,\ type}$ so that it satisfies the equation $map(t,t')t = t'$, for all $t,t' \in Type$. Notice that for this equation to make sense the range of *map* must be *TF*. For the simple example of typing in *Exp*, this is sufficient. More complicated languages with richer operations would require further operations on *Type*, such as "cartesian product", "disjoint sum", and "list", with equations describing their behaviour, as well. This process is straightforward and presents no real difficulties.

Notice that I have not actually given a proper definition of the function *map*; rather, I have given a purely syntactic *equation* serving to induce an equivalence relation on the "raw version" of *Type* (proto-*Type*?). The algebra *Type* is thus the quotient of proto-*Type* by this equivalence relation. It is partly an ***equational theory***. In an equational theory, the basic tool for reasoning about equality of terms (satisfiability of equations) is ***unification*** [Rob65a]. When the equations are given a direction, they become ***rewrite rules*** and can be used to drive simplifying interpreters. An example, mentioned earlier, is the λ-calculus, where the rewrite rules are the rules of λ-conversion; two λ-expressions are considered equal if they reduce to the same normal form. There is a vast literature on such syntactic, proof-theoretic manipulation of terms; an excellent survey of the area is [Hue80a].

The semantic model F^{Exp}, then, has as its carrier $F = Tenv \rightarrow Type$, and F becomes a Σ^{Exp}-algebra via the homomorphism Φ: $T_{\Sigma^{Exp}} \rightarrow F$ as shown in Figure 3, where ψ ranges over *Tenv*, τ, τ', and τ'' range over F, and t ranges over *Type*. Bear in mind that τ is the meaning of some expression ***in the model*** F^{Exp}, and $\tau\psi$ is the type of that expression in the type environment ψ.

Notice that in the equation for "cond" the test for equality in *Type* requires reasoning with the equations for type operators - in this case the

3.1	$\Phi\, x$	=	$\lambda\psi.\psi x$
3.2	$\Phi\, apply(\tau,\tau')$	=	$\lambda\psi.\tau\psi\ \mathbf{E}\ TF \rightarrow (\tau\psi\ \mathbf{E}\ TW \rightarrow bad,\ (\overset{Type}{\underset{TF}{\downarrow}}(\tau\psi))(\tau'\psi)),\ bad$
3.3	$\Phi\, cond(\tau,\tau',\tau'')$	=	$\lambda\psi.\tau\psi\ \mathbf{E}\ \Pi_0 \rightarrow (\tau'\psi=\tau''\psi \rightarrow \tau'\psi,\ bad),\ bad$
3.4	$\Phi\, lambda_x(\tau)$	=	$\lambda\psi.\overset{Type}{\underset{TF}{\uparrow}}(\lambda t.map(t,\tau\psi\{t/x\}))$
3.5	$\Phi\, fix_x(\tau)$	=	$\lambda\psi.\overset{Type}{\underset{TF}{\uparrow}}(Y(\lambda t.map(t,\tau\psi\{t/x\})))$
3.6	$\Phi\, let_x(\tau,\tau')$	=	$\lambda\psi.\tau\psi\ \mathbf{E}\ TW \rightarrow bad,\ \tau'\psi\{\tau\psi/x\}$

Figure 3. Type Semantics of Exp

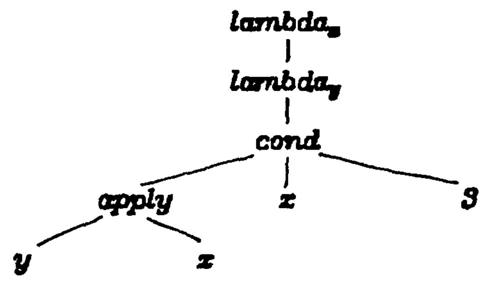
one for *map*. Without going into details, suffice it to say that *Type* has a decidable unification problem, and hence the evaluation of an *Exp* program in the model *Type* is guaranteed to terminate.

6. Evaluation in F^{Exp}

As an illustration of how the evaluation of an *Exp* expression in the model F^{Exp} proceeds, consider the expression:

$$\lambda x.\lambda y.\textit{if } y(x) \textit{ then } x \textit{ else } 3 \qquad (*)$$

Its abstract syntax is:



Evaluation proceeds in a purely bottom-up, synthetic way. The equations defining Φ directly evaluate the leaves of such an abstract syntax tree. The values obtained at each level are combined at successive levels according to the equations. Proceeding in this way leads to the sequence of evaluation steps shown in figure 4. The evaluation steps are presented as rewritings of the form $a \Rightarrow b$, meaning that Φa is evaluated to give b.

Now formula 4.8, the meaning of $(*)$ in F^{Exp}, is not very illuminating as it stands. This is partly the fault of the notation and partly the fault of the way in which the language *Exp* itself constructs higher order functions, both of which are due to λ, the annihilator of orthogonality. One way of getting some insight into the meaning of a λ-expression is to close it. The result of closing formula 4.8 with

4.1 $y \Rightarrow \lambda\psi.\psi y$

4.2 $x \Rightarrow \lambda\psi.\psi x$

4.3 $apply(4.1,4.2) \Rightarrow \lambda\psi.\psi y \in TF \rightarrow (\psi x \in TW \rightarrow bad,\ (\overset{T}{\underset{TF}{\downarrow}}(\psi y))(\psi x)),\ bad$

4.4 $x \Rightarrow \lambda\psi.\psi x$

4.5 $3 \Rightarrow \lambda\psi.\psi 3$

4.6 $cond(4.3,4.4,4.5) \Rightarrow \lambda\psi.$

$[\psi y \in TF \rightarrow (\psi x \in TW \rightarrow bad,\ (\overset{T}{\underset{TF}{\downarrow}}(\psi y))(\psi x)), bad] \in \Pi_0$

$\rightarrow (\psi x{=}\psi 3 \rightarrow \psi x,\ bad),\ bad$

4.7 $lambda_y(4.6) \Rightarrow \lambda\psi.\overset{T}{\underset{TF}{\uparrow}}(\lambda t.map(t,$

$[t \in TF \rightarrow (\psi x \in TW \rightarrow bad,\ (\overset{T}{\underset{TF}{\downarrow}}t)(\psi x)),\ bad] \in \Pi_0$

$\rightarrow (\psi x{=}\psi 3 \rightarrow \psi x,\ bad),\ bad)))$

4.8 $lambda_x(4.7) \Rightarrow \lambda\psi.\overset{T}{\underset{TF}{\uparrow}}(\lambda t'.map(t',\ \overset{T}{\underset{TF}{\uparrow}}(\lambda t.map(t,$

$[t \in TF \rightarrow (t' \in TW \rightarrow bad,\ (\overset{T}{\underset{TF}{\downarrow}}t)(t')),\ bad] \in \Pi_0$

$\rightarrow (t'{=}\psi 3 \rightarrow t',\ bad),\ bad)))))$

Figure 4. Evaluation in F^{Exp}

$\psi = \bot_{Tenv}\{int/3\}$

$t = map(int,bool)$

$t' = int$

where $bool \equiv \pi_0$, is simply:

$$map(int,map(map(int,bool),int))$$

or, with a bit of "syntactic sugar",

$$(int \rightarrow ((int \rightarrow bool) \rightarrow int))$$

which, as anyone with access to an ML implementation can readily verify, is the desired type of (*).

7. An Algebraic Theory of Static Analysis

What is the connection between M^{Exp} and F^{Exp}? Recall that **C**, the class of Σ-algebras, is a complete lattice under the homomorphism partial ordering $\leq$. The abstract syntax of a language Λ is identified with the initial Σ-algebra T_Σ. Every algebra in **C** is a possible model for T_Σ. In other words, every algebra in **C** is a possible semantic model for Λ, and semantics is the corresponding homomorphism. Hence E and Φ give two alternate semantics for *Exp*.

Since a homomorphism is a structure-preserving mapping, $A \leq B$ means that A is consistent with B. In general, A is said to be an ***abstraction*** of B, and the study of algebraic abstractions is called ***abstract algebra***. The algebra of types is consistent with the primary semantics of *Exp* via an obvious homomorphism from the algebra M^{Exp} to F^{Exp}. (We leave the details of this to the reader, as they are easy but (by now) tedious.)

The primary semantic algebra M for Λ is itself initial in a sublattice of **C**, the lattice of algebraic abstractions of M. If M belongs to the class of "deterministic discrete dynamic systems", then it is initial in the lattice of ***abstract interpretations*** [Cou77a, Cou81a].

One main point of this paper is that the techniques of fixed point approximation are not restricted to "deterministic discrete dynamic systems", but apply equally well to *any* continuous algebra. In fact, the determination of necessary constraints on sequencing (i.e., the choice of control flow) is a data flow problem which, as I indicated in the introduction, is especially important for applicative programming languages. For example, ***lazy*** *evaluation* is an approximation that reflects the semantics of *non-strict functions* - two very different ideas!

A second main point of this paper is the connection between model-theoretic and proof-theoretic definitions. A constructive, model-theoretic approach to programming language definition synthesizes a new model from known models and constructors, and then makes it a semantic model for the language via an explicit homomorphism, which can be interpreted mechanically in terms of the base models and constructors. A proof-theoretic approach analyzes an initial algebra with syntactic formulae (equations, axioms, proof rules). Then, the issues of soundness, completeness, and the existence of nontrivial models arise, because one does not know *what* the language is talking about. The benefit is that one knows how to reason *about* the language and its utterances. If decidable, this proof theory can be automated with well-established and uniform methods.

So to show that one semantic model is consistent with another, either exhibit an explicit homomorphism or show that the one is (isomorphic to) a quotient of the other. The latter approach is illustrated in [Don79a]. *Type* is a mixed theory, and requires a mixed approach. That proto-*Type* $\leq M$ is easily shown; one must also show, e.g., that the equation for *map* does not make *Type* trivial. Another possibility (hint) is to construct an explicit model that satisfies the equation; i.e., define a function on types such that $map(t,t')t = t'$ holds.

8. Type Assignment

Returning to the abstract algebra of types, note that Milner's real contribution is an algorithm "...to *discover* a legal type assignment, given a program with incomplete type information," rather than "...just verifying that a given type assignment is legal" [Mil78a, p.359]. The development so far shows how to do the latter. How does the discovery of type assignments fit into the algebraic framework?

The design of an algorithm to discover legal type assignments is guided, in a very transparent way, by what we intend by "legality". A legal type assignment for an expression e is supposed to ensure that e never takes the value *wrong*. What does this mean in terms of the algebra of types?

The astute reader will have noticed that the algebra of types departs from Milner's notion of "type" in that *every* expression has a type in F^{Exp}, including *wrong*, whose type is *bad*. Hence a legal type assignment ensures that $(\Phi e)\psi$ does not have the type *bad* in its range, for all $\psi \in Tenv$.

Evaluation of an expression e in F^{Exp} is purely synthetic (i.e., bottom-up). The result is a mapping from type environments to types. A natural approach is to attempt to generate, during the synthesis, a set of equations constraining the values that the type variables introduced can assume. The choice of these constraints is governed by the need to avoid, in the equations for Φ, the arms of the conditionals that lead to "*bad*" places. These considerations lead to an inductive analysis of the equations defining Φ, as follows.

- The synthesis always begins with the leaves of the abstract syntax tree, and these are always variable references or constants. Equation 3.1 therefore provides the basis of the induction. In particular, there must be no constant or variable having type *bad* in the initial type environment.

The inductive hypothesis is that the arguments of an operator cannot be *bad*. The inductive step is to derive constraints for each operator that preserve the inductive hypothesis. That is, equations whose satisfaction guarantees that the result of applying each operator cannot be *bad*, assuming that the arguments cannot be *bad*. These constraints are derived by inspection of the remaining equations defining Φ:

- Equation 3.2 requires that:

$$\underset{TF}{\overset{Type}{\downarrow}}(\tau\psi) = map(t,t')$$

$$\tau'\psi = t$$

- Equation 3.3 requires that:

$$\tau\psi = \pi_0$$

$$\tau'\psi = \tau''\psi$$

- Equations 3.4 and 3.5 require that the constraints on τ be duplicated with $\psi\{t/x\}$ substituted for ψ; this guarantees that t is constrained so as not to interfere with the hypothesis that τ cannot be *bad*.

- Similarly, equation 3.6 requires that the constraints on τ be duplicated with $\psi\{\tau'\psi/x\}$ substituted for ψ.

Type assignment for an expression e occurs during the evaluation of e in $\mathcal{F}^{Exp}$. Initially, the types of all constants must be known. At each step of the evaluation, the constraining equations corresponding to the operation being performed are generated, and the resulting system of equations is unified, if possible. The types assigned to identifiers and type variables are simply their unifiers.

Notice that for $\boldsymbol{lambda_x}$ and $\boldsymbol{fix_x}$, the constraints on τ must be unified with t as a free variable, whereas for $\boldsymbol{let_x}$ the constraints on τ' must be unified in conjunction with those on $\tau\psi$. Consequently, the constraining equations for

$$\mathbf{let}\ I = \lambda x.x\ \mathbf{in}\ I(I)$$

are satisfiable, whereas those for

$$(\lambda I.I(I))(\lambda x.x)$$

are not.

Why should this be so? Aren't these two expressions semantically equivalent? The problem is that λ-abstraction is an operation that acts on the *presentations* of values, instead of the values themselves. This makes it oblique to the rest of the purely functional† operations of the metalanguage, resulting in the usual bizarre and subtle interactions.

9. An Example of Type Assignment

Figure 5 shows the constraining equations that are generated during the evaluation of (*). The numbers in figure 5 are the step numbers from figure 4; the equations shown beside each number are those generated for the corresponding step of the evaluation. Notice that the final type assignment sets

$\phi 3 = int$

$t = map(int, bool)$

$t' = int$

which gives (*) its expected type as shown in section 6.

10. Conclusions

Does this method work? Yes. The type assignment algorithm sketched here is a bottom-up version of Milner's "Algorithm J". Semantic soundness is proved by formalizing the reasoning used to derive the algorithm. I am omitting the formal proof here because the result is already known and the proof technique is well-known and not germane. The system of equations

† An implementation of these operators may act on presentations but this must be transparent.

4.1

4.2

4.3 $\psi y = map(t_1,t_2)$
$\psi x = t_1$

4.4

4.5 $\psi 3 = int$
(Recall that the types of all constants are known.)

4.6 $t_2 = bool$
$t_1 = \psi 3 = int$

4.7 $\psi\{t/y\}y = map(t_1,t_2)$ which implies $t = map(int,bool)$

4.8 $\psi\{t'/x\}x = t_1$ which implies $t' = int$

Figure 5. Type Assignment

generated during the synthesis of a typing are sufficient to guarantee that the type assignment is legal, but they are not necessary. In fact, many different type algebras are possible. Some are more restrictive, some less. The type assignment algorithm developed here provides a static approximation of the "most general" typing conjectured by Milner. As is clear from the algebraic theory presented in this paper, however, many models of "type" are possible, and whether or not a particular typing is "most general" depends on the model chosen.

φινισ

Acknowledgements

I would like to thank my colleagues - Richard Kieburtz, Gene Rollins, and John Givler - for their insights and criticism which helped me to clarify this material and for their endurance through several conceptual revisions. I would also like to thank the Oregon Graduate Center for providing an excellent working environment and for funding this research.

References

ACM81a.
Proc. ACM Conf. on Functional Programming Languages and Computer Architecture. Oct. 1981.

Bur69a.
Burstall, R. M. and P. J. Landin, "Programs and their Proofs: an Algebraic Approach," *Machine Intelligence* **4** pp. 17-43 (1969).

Chu51a.
Church, A., "The Calculi of Lambda-Conversion," *Annals of Mathematical Studies* **6**Princeton University Press, (1951).

Cou77a.
Cousot, P. and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," *POPL IV*, pp. 238-252 (Jan. 1977).

Cou81a.
Cousot, P., "Semantic Foundations of Program Analysis," pp. 303-342 in *Program Flow Analysis: Theory and Applications*, ed. N. D. Jones and S. S. Muchnick,Prentice-Hall, Englewood Cliffs (1981).

Cur68a.
Curry, H. B. and R. Feys, *Combinatory Logic*, North-Holland, Amsterdam (1968).

Don79a.
Donzeau-Gouge, V., "Denotational Definition of Properties of Program Computations," IRIA RR#349, Le Chesnay (Avril 1979).

Gog77a.
Goguen, J. A., J. W. Thatcher, E. G. Wagner, and J. B. Wright, "Initial Algebra Semantics and Continuous Algebras," *JACM* **24**(1) pp. 68-95 (Jan. 1977).

Hue80a..
Huet, G. and D. C. Oppen, "Equations and Rewrite Rules: A Survey," STAN-CS-80-785, Stanford Computer Science Dept. (Jan. 1980).

Jon81a.
Jones, N. D., "Flow Analysis of Lambda Expressions," pp. 376-406 in *Proc. Symposium on Functional Languages and Computer Architecture*, ,

Goteborg (April 1981).

Kam80a.
Kamin, S., "Final Data Type Specifications," *POPL VII*, pp. 131-138 (Jan. 1980).

McC60a.
McCarthy, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," *CACM* 3(4) pp. 184-195 (April, 1960).

Mil78a.
Milner, R., "A Theory of Type Polymorphism in Programming," *J. Comp. & Sys. Sci.* **17** pp. 348-375 (1978).

Rob65a.
Robinson, J. A., "A Machine-Oriented Logic Based on the Resolution Principle," *JACM* **12**(1) pp. 23-41 (Jan., 1965).

Rus08a.
Russell, B., "Mathematical Logic as Based on the Theory of Types," *Am. J. Math.* **30** pp. 222-262 (1908).

Sco71a.
Scott, D. S. and C. Strachey, "Toward a Mathematical Semantics for Computer Languages," pp. 19-46 in *Proc. Symposium on Computers and Automata*, ed. J. Fox,Polytechnic Institute of New York, New York (1971).

Shua.
Shultis, J., "Hierarchical Semantics, Reasoning, and Translation," Ph.D. Thesis (in preparation) ().