Embedding Type Structure in Semantics

Mitchell Wand

Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, IN 47405 USA

Abstract

We show how a programming language designer may embed the type structure of of a programming language in the more robust type structure of the typed lambda calculus. This is done by translating programs of the language into terms of the typed lambda calculus. Our translation, however, does not always yield a well-typed lambda term. Programs whose translations are not well-typed are considered meaningless, that is, ill-typed. We give a conditionally type-correct semantics for a simple language with continuation semantics. We provide a set of static type-checking rules for our source language, and prove that they are sound and complete: that is, a program passes the typing rules if and only if its translation is well-typed. This proves the correctness of our static semantics relative to the well-established typing rules of the typed lambda-calculus.

1. Introduction

Denotational semantics has proved to be a powerful language for expressing the language designer's decisions about the behavior of programs in his language. Control structure, environment structure, and parameter-passing mechanisms are all examples of complex mechanisms that may be clarified by using semantic methods. In this paper we consider a technique for specifying the designer's decisions about type structure.

This material is based on work supported by the National Science Foundation under grant number MCS 8303325.

Author's address: Computer Science Department, Brandeis University, Waltham, MA 02254 USA

Conventional treatments of semantics already include some information about data types. For example, one might find a domain specification of the form

$$V = Bas + F$$
$$F = V \to V$$

and fragments of equations that look like

$$f \in F \to fa, wrong$$

This, however, is dynamic typing, in that it refers to the value of f at run time. We wish instead to specify static typing, which refers to decisions that are made prior to running the program.

The conventional approach to static typing is to introduce a "static semantics". In this approach a declaration of the form **int** x is regarded as an annotation or loop invariant. The static semantics inspects the syntax tree, essentially doing some variant of data-flow analysis, inserting run-time checks and coercions as necessary, or rejecting the tree if the annotations are inconsistent with the results of the analysis. The "dynamic" or run-time semantics is then applied only to the trees resulting from the static semantics.

The use of static analysis has a number of shortcomings. First, there is no clear statement of the relation between the static and dynamic semantics. What are the properties that the static semantics is designed to ensure? Second, this approach does not serve to differentiate the checking of types from the many other things one might want to check using data-flow analysis.

We adopt the Russelian principle that the purpose of type structure is to distinguish those terms or phrases that denote something from those that do not denote.* Though denotational semantics normally maps program phrases to elements of certain function spaces, it may also be considered to provide a syntactic transformation from program phrases to terms of the metalanguage (typically a lambda-calculus) that denote those elements. Such a translation

* We borrow from philosophy the convention that "denote" may be used as an intransitive verb.

is called a *concrete semantics*. Our strategy is to express the type structure of the programming language through concrete semantics: we arrange the translation so that the well-typed programs are just those whose translations denote. We consider a program to be *meaningful* iff its translation denotes. Ill-typed programs will be meaningless. We refer to such a translation as *conditionally type-correct*.

In this context, the purpose of static semantics is to examine a program and determine whether it is meaningful. At worst, one can perform the translation and check whether it denotes. Given a set of static semantic rules, one can compare them with the concrete semantics. The rules are *sound* if any program that passes them is meaningful. They are *complete* if they pass every meaningful program.

Now, this technique does not solve the problem of type. It is always a design decision as to which terms should not denote and which should denote *wrong*. The technique enables us to express these decisions in the semantics. It also gives a basis for justifying any proposed set of static semantic rules. Last, it enables us to use the many results about the typing rules of the typed lambda-calculus and related formalisms [*e.g.* Hindley 69, 83ab; Coppo 84; Kahn, MacQueen & Plotkin 84] to study more realistic programming languages.

In this paper, we work out the details of such a translation for a simple language. We use continuation semantics to make the translation non-trivial. We use the simply-typed lambda calculus as our metalanguage. (Note that in a conventional translation (*e.g.* [Gordon 79; Milne & Strachey 76]) the metalanguage is the typed lambda-calculus with a richer type structure, but there every program yields a well-typed term. This fact is often used in implementations of such semantics, *e.g.* [Wand 84b]).

We then give a set of static type-checking rules for the language, and show that they are sound and complete: a program phrase passes the type-checking rules iff its translation is well-typed. This proves the correctness of the static semantics with respect to our semantics: a program is meaningful in the semantics iff it passes the static checks.

2. Program Types and their Semantics

We will be manipulating a simple applicative language, with a continuation semantics. We imagine that our programming language has some basic types, such as integer or boolean, and that its types are closed under procedure-formation, that is, if α and β are program types, then so is the type $\alpha \Rightarrow \beta$, which denotes the type of procedures that expect an argument of type α and produce a result of type β. These are the only program types.

We next need to consider the types of the typed lambda-calculus which is our target language. We will assume that for each basic program type ι, there is a corresponding basic lambda-type, which we will also call ι (there may be other basic types as well). Furthermore, we assume that the lambda-types are freely generated by the basic lambda-types under the binary operation '$\rightarrow$'. We shall consider the difference between $\Rightarrow$ and $\rightarrow$ momentarily.

Since we use continuations, we need to talk about the different kinds of continuations we will use. We assume we have some lambda-type C of command continuations (not necessarily basic). All expression continuations will be of type $\sigma \rightarrow C$ for some lambda-type σ. We introduce the notation $K[\sigma]$ to denote the lambda-calculus type $\sigma \rightarrow C$ of σ-continuations.

We next need to decide how values of the different program types are represented. We do this by defining a function $\mathcal{T}$ from program types to lambda-calculus types, with the intention that a value of program type α is represented in the semantics by a lambda-calculus value of type $\mathcal{T}[\![\alpha]\!]$. We start by representing the basic types by themselves, *i.e.* for ι basic, $\mathcal{T}[\![\iota]\!] = \iota$.

Let us consider the representation of a procedure of type $\alpha \Rightarrow \beta$. Procedures are conventionally represented by values of type $K \rightarrow D \rightarrow C$. They take a continuation and an argument (a denotable value) and produce a command continuation.* In our new scheme, the representation will be similar, but we will need to keep track of the types. In the old scheme, the continuation expected to receive the result (an expressed value) returned by the procedure. The procedure produces a value of program type β, represented by a value of type $\mathcal{T}[\![\beta]\!]$, so the continuation should be replaced by a $\mathcal{T}[\![\beta]\!]$-continuation, of type $K[\mathcal{T}[\![\beta]\!]]$. In the old scheme, the argument was a denoted value. Now it will be a representation of type α, so it will be replaced by $\mathcal{T}[\![\alpha]\!]$. Given this information, the procedure should produce a command continuation as before. Hence the representation of a procedure of type $\alpha \Rightarrow \beta$ should be a value of type $K[\mathcal{T}[\![\beta]\!]] \rightarrow \mathcal{T}[\![\alpha]\!] \rightarrow C$, or more simply, $K[\mathcal{T}[\![\beta]\!]] \rightarrow K[\mathcal{T}[\![\alpha]\!]]$.

We can summarize this as follows:

$$\mathcal{T}[\![\iota]\!] = \iota \qquad \text{for } \iota \text{ basic}$$
$$\mathcal{T}[\![\alpha \Rightarrow \beta]\!] = K[\mathcal{T}[\![\beta]\!]] \rightarrow K[\mathcal{T}[\![\alpha]\!]]$$

3. Eliminating the Environment

In a conventional semantics, an environment is a map from identifiers to denotable values. This functionality is not compatible with our goals, since it lumps together all denotable values in one type. We might have used Martin-Löf's more complex functional types [Martin-Löf 79, Constable & Zlatin 84], but that would take us beyond the robust typing of the conventional typed lambda calculus.

We adopt a different approach. In a conventional semantics, the usual semantic clause looks like

$$\lambda\rho\kappa.M$$

where ρ is an environment and κ is a continuation. Rather

* [Milne & Strachey 76] use $D \rightarrow K \rightarrow C$ instead. Our choice yields a prettier definition for $\mathcal{T}$; [Gordon 79] uses this choice as well.

than lump all the values of identifiers into a single environment, we will spread this information across a sequence of lambda-variables. Our usual semantic clause will look like

$$\lambda x_1 \ldots x_n \kappa . M$$

where $x_1, \ldots, x_n$ carry the information that was previously in the single lambda-variable ρ. This way we can attach type information to the x_i that will help us keep track of types.

To do this translation, we will have to maintain, at translation time, some information about the lexical variables. We will do this as a string

$$\xi = [x_1 : \alpha_1, \ldots, x_n : \alpha_n]$$

consisting of the current lexical variables, innermost last, and their associated program types. We refer to this string as a *symbol table* or *type environment*. Our translation will take a symbol table and a program phrase and produce a lambda calculus term for that phrase in the syntactic environment represented by the symbol table. Thus, for any program phrase e, $\mathcal{E}_\xi[\![e]\!]$ will be a lambda-calculus term (the translation of e in context ξ).

We will need notation for some operations on these tables. Associated with a table ξ will be its *variable string*, denoted $\bar{x}^\xi$, that abbreviates the string of typed lambda-variables

$$x_1^{\mathcal{T}[\![\alpha_1]\!]} \ldots x_n^{\mathcal{T}[\![\alpha_n]\!]}$$

Thus a typical semantic clause will be of the form $\lambda \bar{x}^\xi \kappa . M$ and where we might earlier have written $\mathcal{E}[\![e]\!]\rho\kappa$, we will now write $\mathcal{E}_\xi[\![e]\!]\bar{x}^\xi\kappa$. We write $[\xi \mid x : \alpha]$ for the symbol table formed by appending the pair $x : \alpha$ to ξ.

We will also need some notation for types associated with ξ. If X is any lambda-calculus type, we define the lambda-calculus type $\xi \to X$ as follows:

$$([\,] \to X) = X$$
$$([\xi \mid x : \alpha] \to X) = (\xi \to (\mathcal{T}[\![\alpha]\!] \to X))$$

4. The Source Language and its Semantics

The source language will be given by the following grammar:

$$\begin{aligned}
\langle \textit{pgm-type} \rangle &::= \langle \textit{basic type} \rangle \\
\langle \textit{pgm-type} \rangle &::= \langle \textit{pgm-type} \rangle \Rightarrow \langle \textit{pgm-type} \rangle \\
\langle \textit{exp} \rangle &::= \langle \textit{identifier} \rangle \\
\langle \textit{exp} \rangle &::= \textbf{fn } \langle \textit{identifier} \rangle : \langle \textit{pgm-type} \rangle . \langle \textit{exp} \rangle \\
\langle \textit{exp} \rangle &::= (\langle \textit{exp} \rangle \langle \textit{exp} \rangle) \\
\langle \textit{exp} \rangle &::= \textbf{doio}
\end{aligned}$$

This is to be a simple applicative language, using continuation semantics. To emphasise that the continuation semantics is fundamental, we include an imperative operation **doio**, which we imagine does some non-trivial manipulation of the command continuation.

We now give the translation. For each table ξ and expression e, $\mathcal{E}_\xi[\![e]\!]$ will be a closed term of the typed lambda-calculus (though not necessarily a well-typed term!). Let us assume that $\xi = [x_1 : \alpha_1, \ldots, x_n : \alpha_n]$. We first do the clause for identifiers:

$$\mathcal{E}_\xi[\![\langle \textit{identifier} \rangle]\!] = \begin{cases} \lambda \bar{x}^\xi \kappa^{K[\mathcal{T}[\![\alpha_i]\!]]} . \kappa x_i, & \text{if } \langle \textit{identifier} \rangle = x_i \\ (\lambda x . xx)(\lambda x . xx) & \text{otherwise} \end{cases}$$

If the $\langle \textit{identifier} \rangle$ appears in ξ, then we generate a term which sends the appropriate value to a continuation κ (of the right type); in the case of duplicate i's we choose the largest (innermost). If the $\langle \textit{identifier} \rangle$ does not appear in ξ, then we generate an untypable term. (Any untypable term will do; here we have chosen a term that is unsolvable as well.)

The case of applications is also straightforward:

$$\begin{aligned}
&\mathcal{E}_\xi[\![(\langle \textit{exp} \rangle_1 \langle \textit{exp} \rangle_2)]\!] = \\
&\qquad \lambda \bar{x}^\xi \kappa . \mathcal{E}_\xi[\![\langle \textit{exp} \rangle_1]\!]\bar{x}^\xi(\lambda f . \mathcal{E}_\xi[\![\langle \textit{exp} \rangle_2]\!]\bar{x}^\xi(\lambda a . f \kappa a))
\end{aligned}$$

This is precisely as in the usual case, except that the environment is spread over the string $\bar{x}^\xi$ of variables.

It is in the case of abstractions that we see most clearly the machinery at work. Let id_C denote the term $\lambda x^C . x$, the identity function on C. Then the translation is given by:

$$\begin{aligned}
&\mathcal{E}_\xi[\![\textbf{fn } \langle \textit{identifier} \rangle : \langle \textit{pgm-type} \rangle . \langle \textit{exp} \rangle]\!] = \\
&\lambda \bar{x}^\xi \kappa . id_C(\kappa(\lambda \kappa' a . \mathcal{E}_{[\xi \mid \langle \textit{identifier} \rangle : \langle \textit{pgm-type} \rangle]}[\![\langle \textit{exp} \rangle]\!]\bar{x}^\xi a \kappa'))
\end{aligned}$$

This is again analogous to the standard equation for abstractions, except for the id_C, which is needed for technical reasons. The body of the abstraction is translated in an appropriately extended symbol table, and, in place of the usual $\rho[a/id]$, we have $\bar{x}^\xi a$, so that a becomes the value associated with the innermost lexical variable.

Last, let *doio* denote some term of the typed lambda-calculus of type is $K[t'] \to K[t]$, where t and t' are base types. (In the usual case, where C is $S \to A$, *doio*'s type becomes $K[t'] \to t \to S \to A$. Then *doio* might make arbitrary decisions based on its t and S arguments, either passing a value to its $K[t']$ argument or producing *wrong*. All of this is compatible with this functionality for *doio*). Then we can set $\mathcal{E}_\xi[\![\textbf{doio}]\!] = \lambda \bar{x}^\xi \kappa . \textit{doio}\,\kappa$.

We can make several observations about the semantics:

First, for any term e and table ξ, $\mathcal{E}_\xi[\![e]\!]$ is a closed term, not necessarily well-typed.

Second, identifiers have now disappeared entirely from the semantics. In the conventional treatment, we need a semantic domain corresponding to identifiers to serve as the source domain for environments [Clinger, Friedman, & Wand 82]. This is no longer necessary. The semantics resembles the display semantics in [Wand 82], without the tortuous derivation in that paper.

Third, the terms we have generated are not quite in the typed lambda-calculus, because we have not included types on all of the lambda-variables. We shall regard a phrase of our language as meaningful iff its translation is *typable*, that is, iff it is possible to fill in all the types. This is decidable. [Hindley 69,83ab, Milner 78]. It will turn out that the translation always yields a term with at most one type. In the next section, we shall state the main result about typability.

5. The Static Semantics and its Correctness

The static semantic rules for our language will be similar to those of the typed lambda calculus (without polymorphism). We write $\xi \vdash_P e{:}\alpha$ if expression e, in type environment (symbol table) context ξ, has static program type α. We say e is *well-typed* in context ξ iff there is some α such that $\xi \vdash_P e{:}\alpha$. The rules are as follows:

$\xi \vdash_P x{:}\alpha$ iff the type of x in ξ is α
if $[\xi \mid id{:}\alpha] \vdash_P e{:}\beta$, then $\xi \vdash_P (\mathbf{fn}\ id : \alpha.e){:}(\alpha \Rightarrow \beta)$
if $\xi \vdash_P e_1{:}(\alpha \Rightarrow \beta)$ and $\xi \vdash_P e_2{:}\alpha$, then $\xi \vdash_P (e_1\ e_2) : \beta$
$\xi \vdash_P \mathbf{doio} : t \Rightarrow t'$

We write $\vdash_A M{:}\sigma$ if the closed lambda term M can have its type superscripts filled in to yield lambda-calculus type σ. There is a considerable semantic theory associated with this syntactic notion [Hindley 69, 83ab]. If there is some σ such that $\vdash_A M : \sigma$, we say M is *well-typed*. The context will serve to differentiate the two uses of this term.

We can now state the main theorem:

Theorem. *(The static rules are sound and complete). Let e be an expression and ξ be a type environment. Then:*

(i) e is well-typed in context ξ iff $\mathcal{E}_\xi[\![e]\!]$ is well-typed.

(ii) $\xi \vdash_P e{:}\alpha$ iff $\vdash_A \mathcal{E}_\xi[\![e]\!]{:}(\xi \rightarrow K[\mathcal{T}[\![\alpha]\!]] \rightarrow C)$

Proof: Soundness is easy. Furthermore, it is easy to show that if $\xi \vdash_P e{:}\alpha$,then $\xi \rightarrow K[\mathcal{T}[\![\alpha]\!]] \rightarrow C$ is the *only* type of $\mathcal{E}_\xi[\![e]\!]$. For completeness, we proceed by structural induction on expressions. We have three cases, one for each kind of expression.

1. Let e be an identifier x, and let $\xi = [x_1 : \alpha_1, \ldots, x_n : \alpha_n]$. Let $P = \mathcal{E}_\xi[\![e]\!]$. Then we have

$$P = \begin{cases} \lambda \bar{x}^\xi \kappa^{K[\mathcal{T}[\![\alpha_i]\!]]}.\kappa x_i, & \text{if } x = x_i \\ (\lambda x.xx)(\lambda x.xx) & \text{otherwise} \end{cases}$$

If P is well-typed, then it must be of the first form, since the second is untypable. Hence $x = x_i$, and $\xi \vdash_P e : \alpha_i$. We must also show that P has type $\xi \rightarrow K[\mathcal{T}[\![\alpha_i]\!]] \rightarrow C$, but that is, by inspection, precisely the type of P.

2. Let e be the application $(e_1\ e_2)$. Let $P = \mathcal{E}_\xi[\![e]\!]$. Then we have

$$P = \lambda \bar{x}^\xi \kappa^\rho . M \bar{x}^\xi (\lambda f^\sigma . N \bar{x}^\xi (\lambda a^\tau . f \kappa a))$$

where $M = \mathcal{E}_\xi[\![e_1]\!]$ and $N = \mathcal{E}_\xi[\![e_2]\!]$, and type variables have been added on the lambda variables κ, f, and a to facilitate the analysis of typing. We assume P is well-typed and we need to show that e is well-typed. Since P is well-typed, then M and N are well-typed. Then, by the induction hypothesis, e_1 and e_2 must be well-typed. Assume that they have program types α_1 and α_2 respectively. Then, by soundness, we have

$$\vdash_A M : (\xi \rightarrow K[\mathcal{T}[\![\alpha_1]\!]] \rightarrow C)$$
$$\vdash_A N : (\xi \rightarrow K[\mathcal{T}[\![\alpha_2]\!]] \rightarrow C)$$

Since the term $\lambda f^\sigma . N \bar{x}^\xi (\lambda a^\tau . f \kappa a)$ follows $\bar{x}^\xi$ as an argument to M, it must be of type $K[\mathcal{T}[\![\alpha_1]\!]]$, and so $\sigma = \mathcal{T}[\![\alpha_1]\!]$. Similarly, we deduce that $\tau = \mathcal{T}[\![\alpha_2]\!]$. Now, observing the application of f in $f\kappa a$, we deduce that

$$\sigma = \rho \rightarrow \tau \rightarrow C$$

and therefore

$$\mathcal{T}[\![\alpha_1]\!] = \rho \rightarrow K[\mathcal{T}[\![\alpha_2]\!]]$$

Thus, by the definition of $\mathcal{T}$, ρ must be of the form $K[\mathcal{T}[\![\alpha_3]\!]]$ for some program type α_3. Hence $\alpha_1 = \alpha_2 \rightarrow \alpha_3$, and e is well-typed.

3. Let e be the abstraction $(\mathbf{fn}\ i : \alpha.e_1)$ and let $P = \mathcal{E}_\xi[\![e]\!]$. Then

$$P = \lambda \bar{x}^\xi \kappa^\rho . id_C(\kappa(\lambda \kappa_1^\sigma a^\tau . M a \kappa_1))$$

where $M = \mathcal{E}_{[\xi \mid i:\alpha]}[\![e_1]\!]$, and type variables have again been added to facilitate the analysis. Again, assume P is well-typed in ξ. Hence M is well-typed, and therefore e_1 is well-typed in $[\xi \mid i : \alpha]$, say $[\xi \mid i : \alpha] \vdash_P e_1 : \beta$. Then $\xi \vdash_P (\mathbf{fn}\ i : \alpha.e_1) : (\alpha \Rightarrow \beta)$. To complete the proof, we need to show that $\vdash_A P : \xi \rightarrow K[\mathcal{T}[\![\alpha \Rightarrow \beta]\!]] \rightarrow C$. Now, by soundness, M has type $\xi \rightarrow \mathcal{T}[\![\alpha]\!] \rightarrow K[\mathcal{T}[\![\beta]\!]] \rightarrow C$. From this we deduce that $\sigma = K[\mathcal{T}[\![\beta]\!]]$ and $\tau = \mathcal{T}[\![\alpha]\!]$. Therefore the inner lambda-expression in P is of type $K[\mathcal{T}[\![\beta]\!]] \rightarrow \mathcal{T}[\![\alpha]\!] \rightarrow C = K[\mathcal{T}[\![\beta]\!]] \rightarrow K[\mathcal{T}[\![\alpha]\!]] = \mathcal{T}[\![\alpha \Rightarrow \beta]\!]$. Furthermore, the result of the application of κ must be of type C because of the presence of the id_C. Therefore P has type $\xi \rightarrow K[\mathcal{T}[\![\alpha \Rightarrow \beta]\!]] \rightarrow C$, as required

4. The case of **doio** is immediate. *QED.*

6. Extensions

Functions of several arguments can also be handled; the representation of the program type

$$\alpha_1 \times \ldots \times \alpha_n \Rightarrow \beta$$

is given by

$$\mathcal{T}[\![\alpha_1 \times \ldots \times \alpha_n \Rightarrow \beta]\!] = K[\beta] \rightarrow \mathcal{T}[\![\alpha_1]\!] \rightarrow \ldots \rightarrow \mathcal{T}[\![\alpha_n]\!] \rightarrow C$$

This representation is quite different from that obtained by currying the procedure.

We can do catch and throw (non-local jumps) as well. We add to the syntax the productions

$$\begin{aligned}\langle \textit{pgm-type}\rangle &::= \langle \textit{pgm-type}\rangle\text{-cont}\\ \langle \textit{exp}\rangle &::= (\textbf{catch}\ \langle \textit{identifier}\rangle : \langle \textit{pgm-type}\rangle\ \textbf{in}\ \langle \textit{exp}\rangle)\\ \langle \textit{exp}\rangle &::= (\textbf{throw}\ \langle \textit{pgm-type}\rangle\langle \textit{exp}\rangle\langle \textit{exp}\rangle)\end{aligned}$$

Here we introduce new program types of the form α-cont, which is to be the type of continuations that expect to receive a value of type α. A catch expression creates a value of continuation type by seizing the current continuation, binding it to the identifier, and executing the body in the resulting environment. The static rules will force the $\langle \textit{pgm-type}\rangle$ to be of the form α-cont. This continuation may be invoked by using a throw expression. This may be used to cause a premature exit, as in

$$(\textbf{catch}\ x : \textit{int}\text{-cont}\ \textbf{in}\ (\text{add1}\ (\textbf{throw}\ x\ 3)))$$

which returns 3 rather than 4.

We set $\mathcal{T}[\![\alpha\text{-cont}]\!] = K[\mathcal{T}[\![\alpha]\!]]$, and translate as follows:

$$\begin{aligned}&\mathcal{E}_\xi[\![(\textbf{catch}\ i : \alpha\ \textbf{in}\ e)]\!] = \lambda\bar{x}^\xi\kappa.\mathcal{E}_{[\xi|i:\alpha]}[\![e]\!]\bar{x}^\xi\kappa\kappa\\ &\mathcal{E}_\xi[\![(\textbf{throw}\ \beta\ e_1e_2)]\!] =\\ &\qquad\lambda\bar{x}^\xi\kappa^{K[\mathcal{T}[\![\beta]\!]]}.\mathcal{E}_\xi[\![e_1]\!]\bar{x}^\xi(\lambda\kappa'.\mathcal{E}_\xi[\![e_2]\!]\bar{x}^\xi(\lambda a.id_C(\kappa' a)))\end{aligned}$$

Again, these equations are adaptations of the familiar versions of these equations (*e.g.* [Clinger, Friedman, and Wand 82]). In the first equation, κ is both bound as the innermost lexical variable and passed as the continuation. In the second e_1 evaluates to a continuation, which is passed the result of evaluating e_2.

The static typing rules are:

$$\begin{aligned}&\text{if } [\xi \mid i : \alpha\text{-cont}] \vdash_P e{:}\alpha,\\ &\qquad\text{then } \xi \vdash_P (\textbf{catch}\ i : \alpha\text{-cont}\ \textbf{in}\ e) : \alpha\\ &\text{if } \xi \vdash_P e_1 : \alpha\text{-cont and } \xi \vdash_P e_2 : \alpha,\\ &\qquad\text{then } \xi \vdash_P (\textbf{throw}\ \beta\ e_1e_2) : \beta\end{aligned}$$

The first typing rule is straightforward. The second may be explained by noting that since the **throw** expression does not return a value in the normal way, it may be assigned any type β. This is analogous to the treatment of goto's in conventional partial correctness theories.

With these extensions, the theorem carries through straightforwardly.

It would be desirable to eliminate the need to specify the program type β in the the throw-expression, but that would invalidate the key technical point that the translation of a program phrase has only one type. These typing rules also prohibit so-called "full upward continuations" [Friedman & Haynes 85], because in $(\textbf{catch}\ x : \alpha\text{-cont}\ \textbf{in}\ x)$, α would need to match α-cont, which is impossible in our well-founded set of program types. Thus, replacing **catch** and **throw** as "special forms" by functions such as **call/cc** [Friedman *et al.* 84] seems to require a better treatment both of reflexive types and of polymorphism.

Because this system works in the presence of continuation semantics, extension to other imperative features should not cause difficulties. **doio** illustrates the pattern. However, care must be taken to distinguish types in the store as well.

Loops and recursion seem treatable as well [Coppo 84]. More complicated program-type disciplines may require more complex target languages, as suggested in [Fortune, Leivant, & O'Donnell 83] for polymorphic types or [Hook 84] for Russell. In this case the correctness of the static semantics may be harder to prove.

7. Conclusions

We have presented an embedding of type structure in denotational semantics. In this approach, a program phrase is meaningful iff its translation is meaningful (that is, well-typed). One can then give static semantic rules and prove that they are sound and complete, that is, the static semantics exactly characterizes those programs that are meaningful.

Our technique allows the embedding of the type structure of complex program constructs, such as continuations, in the more robust type structure of the lambda calculus. In this way, results on the type structure of the lambda calculus can be applied to the problem of type correctness of more realistic programming languages.

Acknowledgements

The idea of a translation that was not always type-correct was suggested in a rather different context by our student Marek Lao. Albert Meyer suggested investigating continuation semantics in a typed domain, leading to the definition of $\mathcal{T}$. We thank Dan Friedman for his comments on the first draft of this paper.

References

[Clinger, Friedman, and Wand 82]
Clinger, W., Friedman, D.P., and Wand, M. "A Scheme for a Higher-Level Semantic Algebra," presentation at US-French Seminar on the Application of Algebra to Language Definition and Compilation, Fontainebleau, France, 1982; proceedings to appear.

[Constable & Zlatin 84]
Constable, R.L., and Zlatin, D.R. "The Type Theory of PL/CV3" *ACM Trans. on Prog. Lang. and Sys. 6* (1984) 94-117.

[Coppo 84]
Coppo, M. "Completeness of Type Assignment in Continuous Lambda Models" *Theoret. Comp. Sci. 29* (1984) 309-324.

[Fortune, Leivant & O'Donnell 83]
Fortune, S., Leivant, D., and O'Donnell, M. "The Expressiveness of Simple and Second-Order Type Structures," *J. ACM 30* (1983), 151-185.

[Friedman *et al.* 84]
Friedman, D.P., Haynes, C.T., Kohlbecker, E., and Wand, M. "The Scheme 84 Reference Manual" Indiana University Computer Science Department Technical Report No. 153 (March, 1984).

[Friedman & Haynes 85]
Friedman, D.P., and Haynes, C.T. "Constraining Control," *Conf. Rec. ACM Symp. on Principles of Programming Languages* (1985).

[Gordon 79]
Gordon, M.J.C. *The Denotational Description of Programming Languages*, Springer, Berlin, 1979.

[Hindley 69]
Hindley, R. "The Principal Type-Scheme of an Object in Combinatory Logic," *Trans. Am. Math. Soc. 146* (1969) 29–60.

[Hindley 83a]
Hindley, R. "The Completeness Theorem for Typing λ-Terms" *Theoret. Comp. Sci. 22* (1983) 1–17.

[Hindley 83b]
Hindley, R. "Curry's Type-rules are Complete with Respect to the F-Semantics Too" *Theoret. Comp. Sci. 22* (1983) 127-133.

[Hook 84]
Hook, J.G. "Understanding Russell: A First Attempt," in Kahn, G., MacQueen, D.B., & Plotkin, G.D., (Eds.) *Semantics of Data Types: Proceedings*, Lecture Notes in Computer Science, Volume 173, Springer-Verlag, 1984, pp. 69–86.

[Kahn, MacQueen, & Plotkin 84]
Kahn, G., MacQueen, D.B., & Plotkin, G.D., (Eds.) *Semantics of Data Types: Proceedings*, Lecture Notes in Computer Science, Volume 173, Springer-Verlag, 1984.

[Martin-Löf 79]
Martin-Löf, P. "Constructive Mathematics and Computer Programming," *Proc. 6th Intl. Congress for Logic, Methodology, and Philosophy of Science*, (Hannover, August, 1979).

[Milne & Strachey 76]
Milne, R. and Strachey C. *A Theory of Programming Language Semantics*, Chapman & Hall, London, and Wiley, New York, 1976.

[Milner 78]
Milner, R. "A Theory of Type Polymorphism in Programming," *J. Comp. & Sys. Sci. 17* (1978), 348–375.

[Strachey 73]
Strachey, C. "The Varieties of Programming Language," Oxford University Computing Laboratory, Technical Monograph PRG-10 (1973).

[Wand 82]
Wand, M. "Deriving Target Code as a Representation of Continuation Semantics" *ACM Trans. on Prog. Lang. and Systems 4*, 3 (July, 1982) 496–517.

[Wand 84]
Wand, M. "A Types-as-Sets Semantics for Milner-style Polymorphism," *Conf. Rec. 11th ACM Symp. on Principles of Programming Languages* (1984), 158–164.