6B-2 Inserting Injection Operations to Denotational Specifications

Masato Takeichi

Department of Computer Science
The University of Electro-Communications
1-5-1 Chofugaoka, Chofu-shi, Tokyo 182, Japan

Abstract

In describing denotational semantics of programming languages, insertion of injections into sum domains is commonly assumed as a convention and is omitted for the sake of brevity. This in turn leads to difficulties for semantic processing systems which accept denotational specifications as input and mechanically calculate them for debugging the semantics.

In this paper we describe an algorithm for inserting injection operations to denotational specifications as part of typechecking process.

1. Introduction

The Semantics Implementation System (SIS) of Peter Mosses [8] can be considered as the first system which generates a compiler or an interpreter from syntactic and semantic specifications written in the style of denotational semantics. Experience with SIS led us to designing an experimental system [9] which includes a typechecker for which SIS lacks, and a translator to convert denotational description into functional programs. Since it has no interface to the parser, only a few example specifications have been processed. More recently, a Semantic Prototyping System (SPS) of Mitchell Wand has been built [13]. Most of the inefficiencies of SIS pointed out in [1] have been improved in SPS using the tools available in Unix such as Yacc, Franz Lisp, and Scheme 84. Wand has also implemented a typechecker and insists on its importance from his experience with sizable examples. He makes an interesting remark that the most common error detected by the typechecker was failure to inject operands into corresponding sum domains. The principal cause underlying such failure is to be sought in the fact that we usually take a value both as an element of a sum domain and as an element of its summand domain when we write down denotational specifications. This kind of convention is widely adopted in textbooks on denotational semantics [4,12] for avoiding a tedious task of balancing types of operands, and making the specifications more readable. As mentioned in [4], such convention might be taken as a conversion analogous to the coercion operation of programming languages. The necessary operations for retaining consistent types could be inserted in such a way that conversion operations between integers and reals are generated by compilers. This would serve for a concise notation to be accepted by the semantic processing system.

In this paper we deal with an algorithm to insert injection operations to denotational specifications written in a semantics description language, which we define in the next section before going into the main discussion on the algorithm.

2. A Semantics Description Language

Major components of SIS are a parser-generator and an evaluator. The parser-generator accepts concrete syntax of the language and generates a parser to analyze programs written in that language into some form of the abstract syntax tree. The abstract syntax tree corresponds to an element of the syntactic domain which is represented by concrete data structures manipulated by the evaluator. Semantics of the language is written using Denotational Semantics Language (DSL), which is an extension of lambda notation. DSL description is converted into a simpler form to be evaluated when the parse tree is given.

The SPS of Wand consists of similar components. In addition, included is a type-checker for guaranteeing type consistency. Yacc and Scheme 84 take the part of the parser-generator and the evaluator, respectively. Semantics is described using Scheme 84 functions. As mentioned in the last section, programs compiled by SPS run much faster than those by SIS. This proves that the efficient interpreter provided by Scheme 84 means a great deal. And the Yacc parser-generator adopted by SPS seems to do much to increase the efficiency of syntactic processing.

Our Semantics Description Language (SDL) is independent of the parser-generator, while the interface to it should be assumed. We expect to use popular software tools such as Yacc. We also assume that SDL description can be directly evaluated, or translated into certain functional language for execution. In our previous work [9], we chose ML [3,5] as an implementation language of the evaluator. SDL consists of the facilities to define domains and functions; no syntactic definitions are written in SDL. We refrain from giving a complete definition of SDL in this paper as space is limited. Instead, we will informally define a small set of primitives. And expected facilities to be implemented by the evaluator will be mentioned where appropriate.

2.1 Domains

In the semantics processing system, every domain has to be implemented in some way; that is, every element of defined domains should be represented as a value to be calculated by the evaluator.

The ways of defining domains in SDL follow the standard textbook. We leave the details to Section 3.3 of [4].

Let D, D_1, D_2, ... stand for arbitrary domains, and Int, Bool, ? for primitive domains.

(D1) Primitive Domains

(D1.1) Standard domains:
Int of integer values, and
Bool of boolean values,

(D1.2) Singleton domains:
? of a distinguished element **?**, and arbitrary values represented by symbols.

(D1.3) Abstract domains:
Domains of which structures or values are not specified in DSL but are to be provided by the evaluator.

(D2) Function Domains
$D_1 \rightarrow D_2$ of functions with the source D_1 and the target D_2.

(D3) Product Domains
$D_1 \times D_2 \times \ldots \times D_n$ of n-tuples of elements from $D_1, D_2, \ldots, D_n$.

(D4) Sequence Domains
D^* of finite sequences of elements from D.

(D5) Sum Domains
$t_1[D_1] + t_2[D_2] + \ldots + t_n[D_n]$ of $D_1, D_2, \ldots, D_n$ with tags $t_1, t_2, \ldots, t_n$.

Domain equations are used to define recursive domains:

$$D_1 = G_1[D_1, \ldots, D_m]$$
$$\ldots$$
$$D_m = G_m[D_1, \ldots, D_m]$$

where each G_i is a domain expression constructed from $D_1, \ldots, D_m$ and primitive domains using the domain constructors $\rightarrow$, $\times$, $*$, and $+$ described above.

Although it would be unnecessary to explain each of these in detail, several points should be noted.

The domain '?' of (D1.2) is intended to be one consisting of a single element **'?'** representing the "semantically nonsensical" value as in SIS.

Implementation of abstract domains of (D1.3) remains open in the sense of the abstract type in programming languages. This is similar to "holes" of the type system of SPS. We require for these domains only that all the necessary functions and their types are to be given in the part of expression definitions.

The construction rule (D5) of the sum domain differs slightly from that of [4]. Every summand of a sum domain must have a unique tag which is used to discriminate among summands and to inject the summand into the sum domain.

2.2 Expressions

Expressions used in defining semantic functions are usually written in a particular version of lambda notation. We assume in SDL that an expression is either

(E1) a constant of type integer or boolean,

(E2) a variable,

(E3) a combination, i.e., an applicative expression of the form
$$f\ e_1 \ldots e_n$$
where f is a variable and $e_1, \ldots, e_n$ are expressions,

(E4) a lambda expression of the form
$$\lambda v.e,$$
or

(E5) a case expression of the form
$$\textbf{case}\ e_0\{t_1[v_1].\ e_1 \mid \ldots \mid t_n[v_n].e_n\}$$
where $e_0, e_1, \ldots, e_n$ are expressions, $t_1, \ldots, t_n$ are tags of a sum domain and $v_1, \ldots, v_n$ are bindings.

The binding in (E4) and (E5) is an extension to lambda notation. It is either

(B1) a variable x,

or

(B2) a structured binding of the form
$$\delta\ v_1 \ldots v_n$$
where δ is a constructor, $v_1, \ldots, v_n$ are bindings.

The constructor in bindings may be any curried function which makes a structured data of a particular type from its components. Examples are

$$\text{prefix} : \alpha \rightarrow \alpha^* \rightarrow \alpha^*$$
$$\text{pair} : \alpha \rightarrow \beta \rightarrow \alpha \times \beta$$

Constructors 'prefix' and 'pair' are polymorphic (See Section 3.1) in the sense that they are applicable to any types α and β [6]. We will shortly discuss about polymorphism in our typechecking algorithm.

The case expression (E5) is a further extention to conventional lambda notation. It tests values of a sum domain and binds their components to variables of its corresponding summands. To gain a better understanding of the usage of the case notaion, consider the primitive operations over sum domains used in [4].

For a sum domain
$$D = t_1[D_1] + \ldots + t_n[D_n],$$
there are primitive functions

(Test) $\quad isD_i : D \rightarrow \text{Bool}$

$$isD_i\ d = \begin{cases} \textbf{true} \text{ if d comes from summand } D_i \\ \textbf{false} \text{ otherwise} \end{cases}$$

(Projection) $\quad outD_i : D \rightarrow D_i$

$$outD_i\ d = \begin{cases} d_i \text{ in } D_i \text{ if } (isD_i\ d) \text{ holds} \\ ? \quad \text{otherwise} \end{cases}$$

and

(Injection) $\quad inD_i : D_i \rightarrow D$
$$inD_i\ d = d \text{ in } D.$$

As mentioned in the previous section, we use tag t_i to inject values of D_i into D; that is, (inD_i d) is written as (t_i d) in SDL. The test isD_i and the projection $outD_i$ can be written using the case notation:

$$\textbf{case}\ d\ \{t_1[v_1].\textbf{false} \mid \ldots \mid t_i[v_i].\textbf{true} \mid \ldots\}$$
and
$$\textbf{case}\ d\ \{t_1[v_1].\ ? \mid \ldots \mid t_i[v_i].v_i \mid \ldots\},$$
respectively.

As far as typechecking is concerned, special forms like infix notation for addition are not relevant and are excluded from SDL. The conditional expression "**if** e **then** e_1 **else** e_2" could be defined by a function

$$\text{if} : \text{Bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$$
or
$$\text{if} : (\alpha \times \alpha) \rightarrow \text{Bool} \rightarrow \alpha$$

as appropriate. Polymorphic function appears here again.

Polymorphism eliminates the need to consider special functions one by one, and thus makes our typechecking algorithm versatile and independent of the implementation language of the evaluator.

We now extend the domain expression by adding

(D0) type variables α, β, ...

for specifying polymorphic functions.

The expression definition is a system of recursive equations

$E_1 : T_1 = F_1[E_1, \ldots, E_n]$

...

$E_n : T_n = F_n[E_1, \ldots, E_n]$,

where each E_i is a variable, T_i is a domain expression constructed by (D0)-(D4), and each F_i is an expression built by (E1)-(E5). Note that the domain construction rule (D5) is not allowed here.

Expression $F_i[E_1,\ldots,E_n]$ may be omitted; in this case $E_i:T_i$ simply states that E_i is of type T_i which may be polymorphic, i.e., may contain type variables. Otherwise, E_i is defined by $F_i[E_1,\ldots,E_n]$ of type T_i which must not be polymorphic.

3. Types and Typechecking

In order to ensure consistent treatment of the type, we follow a clear exposition of polymorphic typechecking by [2]. We start with discussion of types with relation to domains described in the previous section.

3.1 Types

Correspondence between domains and types is straightforward. In short, types are "structures" given by domain definitions. Given a domain equation

$D_i = G_i[D_1, \ldots, D_m]$

the type specified by D_i is one by $G_i[D_1,\ldots,D_m]$. That is, we are concerned with the structure of the domain.

A type can be either a type variable α, β, ..., or a type operator. Type variable stand for arbitrary types. The type operator corresponds to one of the primitive domains or the domain constructors. Operators standing for the primitive domains like 'Int', 'Bool', '?' and the abstract ones are nullary. Parametric operators like $\rightarrow$, $\times$, $*$, and $+$ takes one or more types as arguments. Types containing type variables are called polymorphic. Other types are monomorphic. It should be noted that types corresponding to domain expressions in domain definitions are monomorphic. In SDL, an expression, particularly a function, can be polymorphic. Thus, a type is either

(T1)

(T1.1) a type variable,

(T1.2) a primitive type operator **int**, **bool**, ..., corresponding to a primitive domain,

(T2) $T_1 \rightarrow T_2$, where T_1 and T_2 are types,

(T3) $T_1 \times T_2 \times \ldots \times T_n$, where T_1, T_2, ..., T_n are types,

(T4) T^*, where T is a type,

or

(T5) $t_1[T_1] + t_2[T_2] + \ldots + t_n[T_n]$, where T_1, T_2, ..., T_n are types, and t_1, t_2, ..., t_n are tags.

3.2 Typechecking and Injection Operations

Typechecking is a process of checking whether every term, or subexpression of an expression has a type consistent with ones of other terms. In particular, we are concerned with the consistency of types of combinations. Let e_0 and e_1 be of type $\mu_0 \rightarrow \pi$ and μ_1 respectively. Then in what condition is the term $(e_0\ e_1)$ meaningful? A sufficient answer to this question would be $\mu_0 = \mu_1$ and the type of $(e_0\ e_1)$ is π. In another case where μ_0 is a sum type and μ_1 is its summand with tag t_1, we could transform the term into acceptable one $(e_0\ (t_1\ e_1))$ using injection operator $t_1 : \mu_1 \rightarrow \mu_0$. We will make a generalization of this idea in our typechecking algorithm.

In SDL, and in many textbooks, the type of the expression is given in its definition as described in the last section. Although naming convention such as 'e' for a variable of type 'Exp' might be applied, there is no declaration of types for locally used variables. This is contrast to conventional typed languages like Pascal. Typechecking in SDL is, therefore, the process of checking whether every $F_i[E_1,\ldots,E_n]$ does or does not have a type consistent with T_i in some way under the condition that each E_j has type T_j for j=1,2,...,n. No types for local variables are specified in SDL.

In addition, our typechecker does not only check the consistency of types but also insert necessary injection operations to SDL specifications.

3.3 Typechecking Algorithm

Our typechecking algorithm partly relies on one for polymorphic type systems in ML [2,6]. It is different, however, in that our algorithm determines types of constituents of an expression from the type of that expression. That is, types are propagated downwards from an expression to its constituent subexpressions. This enables us to insert injection operations into subexpressions properly to keep the type of the larger expression unaffected.

The basic algorithm can be described as follows.

(A1) If a constant c of type **int** appears in the context of required type π, injection function Δ: **int** $\rightarrow \pi$, if any, is inserted to have $(\Delta\ c)$ of type π.
Similarly for a constant of type **bool.**

(A2) If a variable x of type μ appears in the context of required type π, injection function Δ: $\mu \rightarrow \pi$, if any, is inserted to have $(\Delta\ x)$ of type π. Types of variables are assigned by expression definitions or bindings in expressions of the forms (E4) or (E5) and given as an environment.

(A3) For a combination $(f\ e_1\ \ldots\ e_n)$ in the context of required type π, assume that variable f is assigned the type $\mu_1 \rightarrow \ldots \rightarrow \mu_n \rightarrow \mu$, which can be polymorphic with type variables α_1, ..., α_m. Find type environment of type variables α_j by unifying μ with π or its summands. Then replace type variables α_1, ..., α_m in $\mu_1 \rightarrow \ldots \rightarrow \mu_n \rightarrow \mu$ by monotypes using the type environment just found to obtain a monotype $\mu'_1 \rightarrow \ldots \rightarrow \mu'_n \rightarrow \mu'$.
Make a new combination $\Delta(f\ e'_1\ \ldots\ e'_n)$ where Δ is the injection function of type $\mu' \rightarrow \pi$, and each e'_i is a transformed version of e_i in the context of type μ'_i with the

environment of variables remains unchanged.

(A4) For a lambda expression $\lambda v.e$ in the context of type π, find a functional type $\pi_1 \to \pi_2$ from π itself or its summand with injection function Δ: $(\pi_1 \to \pi_2) \to \pi$. Update the environment of variables to reflect the lambda variables in binding v of type π_1 (See below). Then check and transform expression e in the context of type π_2 with the new environment to obtain e'. Make a combination $\Delta(\lambda v.e')$ of Δ and a new lambda expression $(\lambda v.e')$ of type π.

(A5) For a case expression

$$\textbf{case}\ e_0\ \{t_1[v_1].e_1 | \ldots | t_n[v_n].e_n\}$$

in the context requiring type π, find a functional type $\pi_1 \to \pi_2$ from π itself or its summands with injection Δ: $(\pi_1 \to \pi_2) \to \pi$. Assume that π_1 is a sum type

$$\pi_1 = t_1[\mu_1] + \ldots + t_n[\mu_n].$$

Transform e_0 into e_0' of type π_1. For each e_i, find the new environment for variables which reflects binding v_i, and transform e_i with respect to type μ_i to obtain e_i' under that environment. Then make a new expression

$$\Delta(\textbf{case}\ e_0'\ \{t_1[v_1].e_1' | \ldots | t_n[v_n].e_n'\}).$$

of type π.

In (A4) and (A5), the environment of variables needs to be updated. Bindings are either a simple variable or a composite variable structure.

(AB1) If the binding is a simple variable x and the type given is π, then new environment is one updated as x has type π.

(AB2) If the binding is of the form $\delta\ v_1 \ldots v_n$ and the type given is π, assume that δ has type $\mu_1 \to \ldots \to \mu_n \to \mu$, which can be polymorphic with type variables $\alpha_1, \ldots, \alpha_m$. Note that π should be monomorphic. Then, find type environment of type variables α_j by unifying μ with π. Then replace the type variables in $\mu_1 \to \ldots \to \mu_n \to \mu$ to obtain $\mu_1' \to \ldots\ \mu_n' \to \mu'$ $(\mu'=\pi)$. The new environment is obtained by applying this algorithm recursively.

It should be noted that the unification algorithm [11] is used in stages (A3) and (AB2) in a similar way to find particular instances of polymorphic types.

A small example of typechecking and transforming SDL opecification is shown in the Appendix. In this example SDL description is written in the form of S-expression of Lisp.

4. Conclusion

We have devised an algorithm for inserting injection operations to denotational specifications. No local binding mechanisms such as "**let** ... **in** ...", or, "... **where** ...", are included in SDL described in this paper. Extension of the above algorithm to such expressions is straightforward.

In a sense, our algorithm is based on a compromise reached by restricting the class of acceptable expressions to ones described in Section 2.2 at the cost of generality. Although the rule (E3) seems too restrictive at first sight, it turns out to be no practical problem. If more general form of combination $(e_0\ e_1)$ were allowed, the types of e_0 and e_1 could not always been deduced from the type of $(e_0\ e_1)$ alone. A functional type should be produced for $e_0 = \lambda v.e_0'$ from little knowledge of the type of e_0. This can be done in ML where we do not insert injection operations. From this observation, we have chosen the restricted class of expressions for our specification language.

Mitchell [7] deals with polymorphic type-checking with automatic coercions between types. Allowable coercion there should be of the form "α is coercible to β" where α and β are atomic types. This is too restrictive for our purpose.

This work has come from an experimental system by Tsuyoshi Ohira [10] in which the algorithm for typechecking and insertion of injection operations was partly implemented but not formally treated.

We plan to construct an efficient system for semantics prototyping in which our type-checker will be incorporated.

References

[1] Bodwin, J., Bradley, L., Kanda, K., Litle, D, and Pleban, U., "Experience with an experimental compiler generator based on denotational semantics", Proc. ACM '82 Symp. on Compiler Construction, SIGPLAN Notices Vol. 17, No.6, June 1982, pp. 216-229.

[2] Cardelli, L., "Basic Polymorphic Typechecking", Polymorphism, Vol. 2, No. 1, 1984.

[3] Chujo, H., and Takeichi, M., "Porting ML on a New Machine", Proc. 28th Symp. of Information Processing, 1984, pp. 427-428, (In Japanese), also "VAX-Unix ML" developed by Luca Cardelli, Bell Labs, March 1983.

[4] Gordon, M. J. C., The Denotational Description of Programming Languages, Springer-Verlag, 1979.

[5] Gordon, M. J., Milner, R., Wadsworth, C. P., Edinburgh LCF, LNCS 78, Springer-Verlag, 1979.

[6] Milner, R., "A theory of type polymorphism in programming", J. Comput. Syst. Sci., No. 17, 1978, pp. 348-375.

[7] Mitchell, J. C., "Coercion and Type Inference (Summary)", 11th ACM Symp. on POPL, 1984, pp. 175-185.

[8] Mosses, P. "SIS — Semantics Implementation System: Reference Manual and User Guide", DAIMI MD-30, Department of Computer Science, University of Aarhus, Denmark, 1979.

[9] Ohira, T., and Takeichi, M., "A Language Development System", Proc. 28th Symp. of Information Processing, 1984, pp. 329-330, (In Japanese).

[10] Ohira, T., A Language Development System, Master's thesis, 1984, The University of Electro-Communications (In Japanese).

[11] Robinson, J. A., "A machine-oriented logic based on the resolution principle", J. ACM, Vol. 12, No. 1, Jan 1965, pp. 23-49.

[12] Stoy, J. E., Denotational Semantics, MIT Press, 1977.

[13] Wand, M., "A Semantic Prototyping System", Proc. ACM '84 Symp. on Compiler Construction, SIGPLAN Notices, Vol. 19, No. 6, June 1984, pp. 213-221.