New Generation Computing, 4 (1986) 365-381
OHMSHA, LTD. and Springer-Verlag

NEW GENERATION COMPUTING

© OHMSHA, LTD. 1986

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.

Received 16 December 1985
Revised manuscript received 16 June 1986

Abstract In describing denotational semantics of programming languages, injection operations into sum domains are conventionally 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. This paper describes an algorithm for inserting injection operations to denotational specifications as part of the typechecking process.

Keywords: Semantics Implementation, Denotational Semantics, Typechecking, Polymorphic Type.

§1 Introduction

The *Semantics Implementation System* (SIS) of Peter Mosses[9] 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[10] which includes a typechecker for which SIS lacks, and a translator to convert denotational description into functional programs. Since it had no interface to parser generator, only a few example specifications and programs were processed. More recently, a *Semantic Prototyping System* (SPS) of Mitchell Wand has been built.[13] Most of the inefficiencies of SIS pointed out in Ref. 1) have been improved in SPS using the tools such as Yacc and Scheme 84 available in Unix operating systems.* Wand has also implemented a typechecker and insists on its importance from his experience with sizable examples. He makes an interesting

* Developed and Licensed by AT & T.

remark that most common error detected by the typechecker is 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 one of its summand domain when we write down denotational specifications. This kind of convention is widely adopted in the literature on denotational semantics[5,12] for avoiding a tedious task of balancing types of operands, and for making the specification more readable. As mentioned in Ref. 5), such convention might be taken as a conversion analogous to the coercion operation of programming languages. The necessary operations for retaining the type consistency 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 tentative semantics description language.

§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 the program written in that language into an abstract syntax tree. The abstract syntax tree corresponds to an element of the syntactic domain which is represented by concrete data structure manipulated by the evaluator. Semantics of the language is written using *Denotational Semantics Language* (DSL), which is an extension of lambda notation. The DSL description is then 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 typechecker 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. According to Ref. 13), programs compiled by SPS run much faster than those by SIS. Wand concludes 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 *Sementics Description Language* (SDL) is independent of the parser generator, while the interface to it should be assumed. We expect to use the state-of-the-art software tools such as Yacc and Lex. We also assume that the SDL description can be directly evaluated, or translated into certain functional language for execution. In our previous work,[10] we chose ML[3,6] 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 because SDL is currently being developed and it is not definitive. Instead, we will informally introduce 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. In other words, each domain should be represented by a concrete data type of the language with which the evaluator is to be implemented. We do not deal with *polymorphic values* in our evaluator, while do with *polymorphic functions* in describing semantic functions. Hence we make the assumption that every value in denotational description lies in certain data type of the implementation language, and a correspondence between them exists.

In this section, we use the term *type* to refer to the domain when its representation is in mind.

The way of defining domains in SDL follows standard textbook. We leave the details to Section 3.3 of Ref. 5).

Let D, D_1, D_2, $\cdots$ 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 any domain with a single element represented by a symbol.
- (D1.3) Abstract domains: Domains of which structures and values are not specified in SDL 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, ..., 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 union of D_1, D_2, ..., D_n with tags t_1, t_2, ..., 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, ..., 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. In our typechecking algorithm, failure in matching the type of an expression with required type results in assigning **?** to that expression (See Section 3.3). Other singleton domains will be used to define finite domains in combination with operation + of (D5). For example, we can define a finite domain

$$Finalstate = abort[Error] + normal[Stop]$$

where *Error* and *Stop* are singleton domains of which values are *error* and *stop*, respectively.

Implementation of abstract domains (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 (See Section 2.2). For example, the domain *Ide* of identifiers commonly used in specifications of programming languages might be treated as an abstract domain; we are not interested in how the element of *Ide* is represented. If we use a function *equal_ide* to check the equality of two identifiers, all that we need in typechecking is the type of that function, i. e., $(Ide \times Ide) \rightarrow Bool$.

Rule (D5) for construction of the sum domain differs slightly from that of Ref. 5). 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. The usage of the tag will be explained in the next section.

As mentioned earlier, we make the assumption that the element of the syntactic domain is to be represented by an abstract syntax tree generated by the parser. DSL of SIS has a particular construction rule for syntactic domains. In contrast to this, our syntactic domains are constructed using general rules for sums and products of syntactic domains themselves. For example, a syntactic domain *Cmd* of commands

$$cmd ::= cmd \; ; \; cmd \mid ide := exp$$

can be defined as

$$Cmd = cmd_seq[Cmd \times Cmd] + cmd_asg[Ide \times Exp]$$

where *Ide* and *Exp* are syntactic domains for identifiers and expressions.* We do not distinguish between syntactic and semantic domains in domain construction.

* One of the referees suggested that the use of terminal symbols ought to be allowed in SDL to write $Cmd = [Cmd \text{ ";" } Cmd] + [Ide \text{ ":=" } Exp]$ as an abbreviaton to $Cmd = $ "Cmd; Cmd" $[Cmd \times Cmd]$ + "Ide:=Exp" $[Ide \times Exp]$. Improvements should be made on SDL for the user to write the specification more comprehensible, though it is not our main concern in this paper.

2.2 Expressions

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

(E1) a constant of type integer or Boolean,
(E2) a variable,
(E3) a combination, or an applicative expression of the form

$$f\ e_1\ \dots\ e_n$$

where $n \geq 1$, f is a variable, and $e_1, \dots, e_n$ are expressions. Note that f must be a variable; expression of other form is not allowed. (This is motivated later in Sections 3 and 5).
(E4) a lambda expression of the form

$$\lambda v.e,$$

where v is a binding,
or
(E5) a case expression of the form

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

where $e_0, e_1, \dots, e_n$ are expressions, $t_1, \dots, t_n$ are tags of a sum domain, and $v_1, \dots, v_n$ 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

$$\gamma v_1\ \dots\ v_n$$

where $n \geq 1$, γ is a constructor, and $v_1, \dots, v_n$ are bindings.

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

$$prefix\colon\ \alpha \rightarrow \alpha^* \rightarrow \alpha^*$$
$$pair\colon\ \alpha \rightarrow \beta \rightarrow \alpha \times \beta.$$

Constructors *prefix* and *pair* are *polymorphic* in the sense that they are applicable to values of any types α and β. We will shortly discuss about polymorphism in our typechecking algorithm.

The case expression is another extension to conventional lambda notation. It tests a value of a sum domain and binds it to variables in the binding of corresponding summand. To gain a better understanding of the usage of the case expression, consider the primitive operations over sum domains in Ref. 5). For a sum domain

$$D = t_1[D_1] + \ldots + t_n[D_n],$$

there are primitive functions

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

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

and

$$inD_i\ d = d \text{ in } D \qquad \text{(Injection)}$$

Note that these functions have types

$$isD_i\colon D \to Bool$$
$$outD_i\colon D \to D_i$$
$$inD_i\colon D_i \to D.$$

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

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

respectively. As will be shown in following examples, the case notation is more useful than primitive operations like tests and projections.

As far as typechecking is concerned, special forms like infix notation for arithmetic operations are not relevant and are excluded from SDL. The conditional expression

$$\textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2$$

could be defined by a function

$$if\colon Bool \to \alpha \to \alpha \to \alpha$$

or

$$if\colon (\alpha \times \alpha) \to Bool \to \alpha$$

as appropriate. Polymorphic functions appear again. In fact polymorphism eliminates the need to consider special functions one by one, and makes our typechecking algorithm versatile and independent of the implementation language of the evaluator. Polymorphic values are, however, not included in SDL domains. That is, functions like *prefix* and *pair* can be used to describe semantics such a way that they appear at the function position of combinations and

the constructor position of structured bindings. We can say that polymorphic functions are second class objects in the sense that they are not allowed to become arguments of functions.

We now extend the domain expression by adding

(D0) Type variables α, β, ...

for specifying types of polymorphic functions.

The expression definition is a system of recursive equations

$$E_1 : T_1 = F_1[E_1, \dots, E_n]$$
$$\dots$$
$$E_n : T_n = F_n[E_1, \dots, E_n]$$

where each E_i is a variable, T_i is a domain expression constructed using (D0)-(D4), and each F_i is an expression built from (E1)-(E5). Note that the domain construction rule (D5) is not allowed here. This effectively eliminates the possibility of defining sum domains of which summands contain type variables, i. e., polymorphic sum domains.

Expression $F_i[E_1, \dots, E_n]$ may be omitted; in this case E_i: T_i simply states that E_i is of type T_i. Otherwise, E_i is defined by the right-hand side of the equation. The type may be polymorphic in the first case, but must not be polymorphic in the second case.

§3 Types and Typechecking

In order to ensure consistent treatment of the type, we follow a clear exposition of polymorphic typechecking scheme by Cardelli.[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 rather straightforward. Types are *structures* given by domain definitions. Given a domain equation

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

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

A type can be either a type variable or a type operator. Type variable stands for an arbitrary type. Type operator corresponds to one of the primitive domains or the domain constructors. An operator standing for primitive domains like *Int, Bool* or *?*, or an abstract domain is nullary. Parametric operators like $\rightarrow$, $\times$, $*$, and $+$ take one or more types as arguments. Types containing type variables are *polymorphic* and called *polytypes*. Other types are *monomorphic* and called *monotypes*. Recall that types corresponding to domain expressions in the domain definition are monomorphic. In SDL, only a function can be polymorphic.

Thus, a type is either

(T1) an atomic type,
  (T1.1) a type variable,
  (T1.2) a primitive type operator among **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 $T_1, T_2, \ldots, T_n$ 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 $t_1, t_2, \ldots, t_n$ are tags, and $T_1, T_2, \ldots, T_n$ are monotypes.

As mentioned earlier, constituent types of a sum type must be monomorphic.

The correspondence between domains and types is straightforward. For example, for a set of domain equations

$$D_0 = t_1[D_1] + t_2[D_2]$$
$$D_1 = Int \times D_0$$
$$D_2 = Bool$$

a type σ corresponding to D_0 is $\sigma = t_1[\mathbf{int} \times \sigma] + t_2[\mathbf{bool}]$ where **int** and **bool** are primitive types corresponding to the primitive domains *Int* and *Bool*.

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 f be of type $\sigma_1 \rightarrow \sigma$, and e of type σ'_1. Then what condition should be satisfied for the combination $(f\ e)$ being meaningful? A sufficient condition would be $\sigma_1 = \sigma'_1$, getting the type σ for $(f\ e)$. In another case where σ_1 is a sum type and σ'_1 is its summand with tag t_1, we can transform the term into acceptable one $(f(t_1\ e))$ using injection operator $t_1: \sigma'_1 \rightarrow \sigma_1$. 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 previous section. Although naming convention such as e for a variable of type *Exp* might be applied, there is no declaration of types for locally declared variables. This is very contrast to conventional typed language 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 under the condition that each E_j has type T_j for $j = 1, \ldots, n$. Note that T_i should be monomorphic in the definition of the form

$$E_i: T_i = F_1[E_1, \ldots, E_n].$$

No types for local variables are specified in SDL.*

Our typechecker does not only check the consistency of types, but also inserts necessary injection operations to SDL specifications. Let **P** be the typechecking procedure producing SDL specifications with injection operations inserted for any SDL specifications. Then, for any x, $x'=\mathbf{P}(x)$ is also an SDL specification and $x'=\mathbf{P}(x')$ is expected. That is, **P** has the idempotent property $\mathbf{P}^2=\mathbf{P}$.

3.3 Typechecking Algorithm

Our typechecking algorithm partly relies on the polymorphic type system of ML.[2,7] It is different, however, in that ours determines types of constituents of an expression from the type of that expression, while the type of an ML expression is inferred from known types of its constituents. That is, types are propagated downwards from an expression to its constituent subexpressions in our algorithm. This enables us to insert injection operations into subexpressions properly to keep the type of the larger expression unaffected. We now introduce some notations for describing the algorithm.

We use the notation

$$[\sigma'\rightarrow\sigma]$$

to represent an injection operation. For monotypes σ' and σ other than $?$, $[\sigma'\rightarrow\sigma]$ is the function that injects every element of type σ' into one of type σ. In particular, it is simply the identity function if σ' and σ are identical. If σ' or σ is polymorphic, or either of them is $?$, it stands for the function $\lambda x.\mathbf{?}$ which maps any value into the nonsensical element **?** of type $?$.

The *environment ρ for variables* associates variables with their types. The initial environment ρ_0 for the definitions

$$E_i : T_i=F_1[E_1, \ldots, E_n] \text{ for } i=1, 2, \ldots, n$$

is

$$\rho_0=\{E_1\rightarrow T_1; \ldots ; E_n\rightarrow T_n\}$$

that represents the function

$$\rho_0 x=\begin{cases} T_i & \text{if } x=E_i \\ \mathit{undefined} & \text{otherwise}\end{cases}$$

An environment ρ is updated by $\{x\rightarrow\sigma\}$ to yield a new environment $\rho+\{x\rightarrow\sigma\}$:

* The accepted usage of global declarations, e. g., e:*Exp* for variable e and decorated variables e_1, e', etc. being of type *Exp*, may help typechecking. We do not adopt such declarations in SDL because the types of local variables can be inferred from the context as shown in the algorithm. However, there remains the possibility of using such global declarations to locate erroneous specifications of types.

$$(\rho+\{x\rightarrow\sigma\})y=\begin{cases}\sigma & \text{if } x=y\\ \rho\ y & \text{otherwise}\end{cases}$$

Similarly, the *environment* π *for type variables* maps type variables to monotypes. Substitution τ/π of type variables in a type expression τ with monotypes under the type environment π is defined as

(1) For an atomic type
  (1.1) a type variable α, $\alpha/\pi=\pi\alpha$
  (1.2) a primitive type operator o, $o/\pi=o$
(2) For a functional type $\tau_1\rightarrow\tau_2$, $(\tau_1\rightarrow\tau_2)/\pi=(\tau_1/\pi)\rightarrow(\tau_2/\pi)$
(3) For a product type $\tau_1\times\tau_2\times\ \ldots\ \times\tau_n$,
  $(\tau_1\times\tau_2\times\ \ldots\ \times\tau_n)/\pi=(\tau_1/\pi)\times(\tau_2/\pi)\times\ \ldots\ \times(\tau_n/\pi)$
(4) For a sequence type τ^*, $(\tau^*)/\pi=(\tau/\pi)^*$
(5) For a sum type σ, which should be a monotype, $\sigma/\pi=\sigma$.

Finally, we denote the expression obtained by inserting injection operations to the expression e in the context requiring type σ under the environment ρ by

$$\langle e : \sigma \mid \rho\rangle$$

where σ is a monotype.

The basic algorithm **P** can be described as follows. We use σ, σ', σ'', σ_1, ... to represent monotypes, and τ, τ_1, ... to represent polytypes (including monotypes).

(A1) If a constant i of type **int** appears in the context requiring type σ, injection function ψ: **int**$\rightarrow\sigma$, if any, is inserted to obtain $(\psi\ i)$ of type σ. That is,

$$\langle i : \sigma \mid \rho\rangle=[\mathbf{int}\rightarrow\sigma]\,i$$

Similarly for a constant b of type **bool**,

$$\langle b : \sigma \mid \rho\rangle=[\mathbf{bool}\rightarrow\sigma]\,b$$

The type *?* is considered to be universal ; it is consistent with any type.

$$\langle \mathbf{?} : \sigma \mid \rho\rangle=\mathbf{?}$$

(A2) If a variable x of type σ' appears in the context requiring type σ, injection function ψ: $\sigma'\rightarrow\sigma$, if any, is inserted to obtain $(\psi\ x)$ of type σ. Types of variables are assigned by expression definitions or bindings in expressions of the form (E4) or (E5). The type of each variable is maintained as an environment ρ.

$$\langle x : \sigma \mid \rho\rangle=[\rho x\rightarrow\sigma]\,x$$

(A3) For a combination $(f\ e_1\ \ldots\ e_n)$ in the context requiring type σ, assume

that variable f is assigned a functional type $\rho f = \tau_1 \to \ldots \to \tau_n \to \tau$, which can be polymorphic with type variables $\alpha_1, \ldots, \alpha_m$. Find a type environment π for type variables α_j by unifying τ with σ or its summands. Then replace type variables α_j in $\tau_1 \to \ldots \to \tau_n \to \tau$ by monotypes using the type environment π just obtained to yield a monotype $\sigma'_1 \to \ldots \to \sigma'_n \to \sigma'$. That is, $\sigma' = \tau/\pi$ and $\sigma'_i = \tau_i/\pi$ for $i=1, \ldots, n$. Finally make a new combination

$$\langle (f\ e_1 \ldots e_n) : \sigma \mid \rho \rangle = [\sigma' \to \sigma](f\ e'_1 \ldots e'_n)$$

where each e'_i is a transformed expression of e_i under the context of required type σ'_i, i. e., $e'_i = \langle e_i : \sigma'_i \mid \rho \rangle$.

If f does not have a functional type of the above form,

$$\langle (f\ e_1 \ldots e_n) : \sigma \mid \rho \rangle = \mathbf{?}$$

(A4) For a lambda expression $\lambda v.e$ in the context requiring type σ, find a functional type $\sigma' \to \sigma''$ from σ itself or its summands with injection function ψ: $(\sigma' \to \sigma'') \to \sigma$. Update the environment ρ to reflect the lambda variables in binding v of type σ' as $\rho + \{\sigma/v\}$ (See below). Then check and transform e in the context of required type σ'' with the new environment to obtain e'. Finally make a combination of ψ and a new lambda term $\lambda v.e'$ getting $\psi(\lambda v.e')$. That is,

$$\langle \lambda v.e : \sigma \mid \rho \rangle = [(\sigma' \to \sigma'') \to \sigma](\lambda v. \langle e : \sigma'' \mid \rho + \{\sigma'/v\} \rangle)$$

If no functional type is found from σ or its summands,

$$\langle \lambda v.e : \sigma \mid \rho \rangle = \mathbf{?}$$

(A5) For a case expression

$$e = \mathbf{case}\ e_0 \{ t_1[v_1].e_1 \mid \ldots \mid t_n[v_n].e_n \}$$

in the context requiring type σ, find a functional type $\sigma' \to \sigma''$ from σ itself or its summands with injection function $\psi:(\sigma' \to \sigma'') \to \sigma$. Assume that σ' is a sum type

$$\sigma' = t_1[\sigma_1] + \ldots + t_n[\sigma_n].$$

Transform e_0 into e'_0 of type σ':

$$e'_0 = \langle e_0 : \sigma' \mid \rho \rangle$$

For each e_i, find the new environment $\rho + \{\sigma_i/v_i\}$ for variables which reflects binding v_i of type σ_i, and transform e_i with respect to type σ'' to obtain e'_i under that environment:

$$e_i = \langle e_i : \sigma'' \mid \rho + \{\sigma_i/v_i\} \rangle \text{ for } i = 1, 2, \ldots, n$$

Then make a new expression

$$\langle e : \sigma \mid \rho \rangle = [(\sigma' \rightarrow \sigma'') \rightarrow \sigma](\textbf{case } e'_0 \{ t_1[v_1].e'_1 \mid \ldots \mid t_n[v_n].e'_n \})$$

of type σ.

If no functional type is found from σ or its summands, or if the source type σ' of the functional type is not a sum type,

$$\langle e : \sigma \mid \rho \rangle = \textbf{?}$$

The injection function may well be the identity function. The typechecking procedure **P** will report type inconsistencies to make the term be reduced to **?** if the conditions mentioned in each case are not satisfied. It should be noted that polymorphic types are allowed only for the function f in (A3).

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

(AB1) If the binding is a variable x and the type given to it is σ, then the new environment is one updated as x has type σ. That is,

$$\rho + \{\sigma/x\} = \rho + \{x \rightarrow \sigma\}$$

(AB2) If the binding is of the form

$$\gamma \ v_1 \ \ldots \ v_n$$

and the given type is σ, assume that the constructor γ has type $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$, which can be polymorphic with type variables $\alpha_1, \ldots, \alpha_m$. Note that σ must be monomorphic. Then find type environment π for type variables α_j by unifying τ with σ. Finally replace type variables α_j in $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$ by monotypes using the type environment π to obtain $\sigma'_1 \rightarrow \ldots \rightarrow \sigma'_n \rightarrow \sigma$. That is, $\sigma = \tau/\pi$, and $\sigma'_i = \tau_i/\pi$ for $i = 1, \ldots, n$. The new environment is obtained by applying this algorithm recursively:

$$\rho + \{\sigma/(\gamma v_1 \ \ldots \ v_n)\} = \rho + \{\sigma'_1/v_1\} + \ \ldots \ + \{\sigma'_n/v_n\}$$

If the constructor γ does not have the type as above,

$$\rho + \{\sigma/(\gamma v_1 \ \ldots \ v_n)\} = \rho + \{v_1 \rightarrow ?\} + \ \ldots \ + \{v_n \rightarrow ?\}$$

The unification algorithm[11] is used in stages (A3) and (AB2) to find particular instances of polymorphic types.

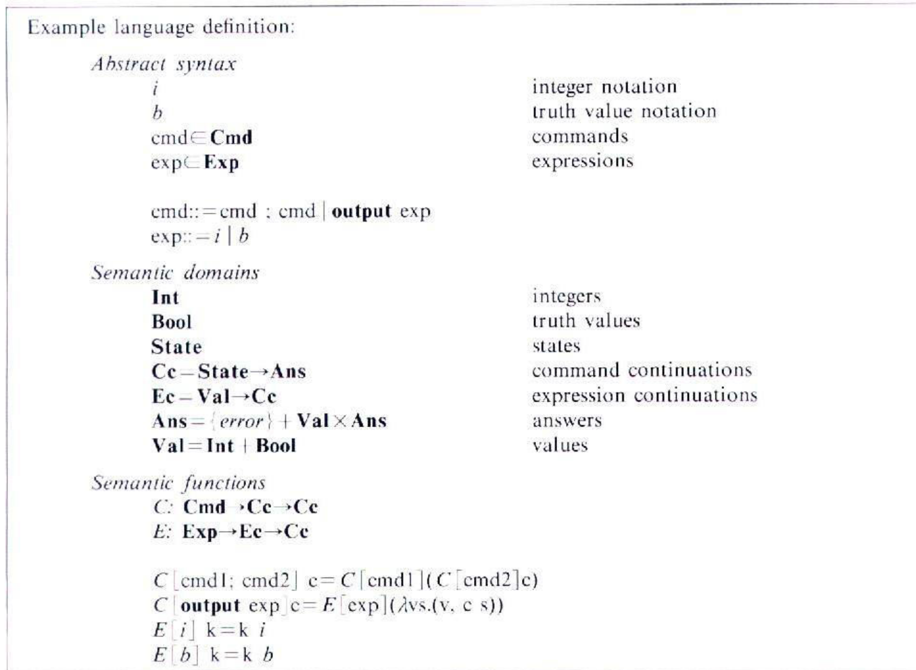
To illustrate how the types of subexpressions are determined, consider a simple term $(pair\ x\ y)$ in the context of required type $Val = val_1[\textbf{int}] + val_2[Num \times \textbf{bool}]$. Assume that $Num = num_1[\textbf{int}] + num_2[?]$, and the environment for variables gives type **int** and **bool** to x and y, respectively, i. e., $\rho = \{x \rightarrow \textbf{int}; y \rightarrow \textbf{bool}\}$. To begin with, Rule (A3) is applied to $(pair\ x\ y)$. The type of $pair$ is polymorphic, i. e., $\alpha \rightarrow \beta \rightarrow \alpha \times \beta$. By unifying $\alpha \times \beta$ with the second summand of Val, a type environment $\pi = \{\alpha \rightarrow Num; \beta \rightarrow \textbf{bool}\}$ is obtained. And $\sigma' = (\alpha \times \beta)/\pi = Num \times \textbf{bool}$. Then the injection operation for that term is $[\sigma' \rightarrow Val] = val_2$. The next step is to check types of subexpressions x and y, which must be

transformed to have type *Num* and **bool**, respectively. We can apply Rule (A2) to both terms. From the environment ρ, $\rho x = \mathbf{int}$, which is injected into *Num* by num_1, i. e., $[\rho x \to \mathbf{int}] = num_1$. And y remains as it is. Thus, we obtain a term

$$\langle (pair\ x\ y) : Val \mid \rho \rangle = (val_2(pair(num_1\ x) y))$$

of type *Val*.

A small but complete example of typechecking a specification in Fig. 1 is shown in the Appendix. The specification is written in the form of S-expression of Lisp, which should be considered as an internal form of SDL. A more sophisticated style of specifications will be allowed in the final definition of SDL.



Example language definition:

Abstract syntax

i	integer notation
b	truth value notation
cmd ∈ **Cmd**	commands
exp ∈ **Exp**	expressions

cmd::= cmd ; cmd | **output** exp
exp::= i | b

Semantic domains

Int	integers
Bool	truth values
State	states
Cc = State → Ans	command continuations
Ec = Val → Cc	expression continuations
Ans = {*error*} + **Val × Ans**	answers
Val = Int + Bool	values

Semantic functions

C: **Cmd → Cc → Cc**
E: **Exp → Ec → Cc**

C[cmd1; cmd2] c = C[cmd1](C[cmd2]c)
C[**output** exp] c = E[exp](λvs.⟨v, c s⟩)
E[i] k = k i
E[b] k = k b

Fig. 1

§4 Reservation

We have not specified how to choose a summand from possible candidates in the clauses "by unifying τ with σ or its summands" in (A3), and "from σ itself or its summands" in (A4) and (A5). The most general solution would be one to try *all* the possible cases using backtracking. However, it would be impractical for sizable specifications. Since the main purpose of our algorithm is still guaranteeing the type consistency of specifications, no backtracking is implemented in our typechecker; actually only the first possible summand is

taken. The validity of inserting injection operations could be confirmed to some degree by applying the procedure **P** to the resutling specification again to check whether the idempotent property is satisfied. Of course, it does not follow that the transformation is correct even if that property holds. In either case, the resulting specification has to be inspected whether it is or is not the intended one. If not, the necessary injection operation should be inserted to make the specification unambiguous.

To find out the cause of ambiguity, the dependency graph for domains would be helpful. The dependency graph consists of nodes containing type operators, and of arcs directed to constituent types. Each node is associated with a type of which structure is given by the maximal subgraph containing that node. The domains defined in the specification correspond to types associated with the dependency graph. Fig. 2 illustrates the graph for domains defined in Fig. 1. Summands with the identical structure do not exist in this case. If several descendants of a +node have the same kind of operators among →, ×, *, or primitive types, care should be taken to make the specificaton unambiguous.

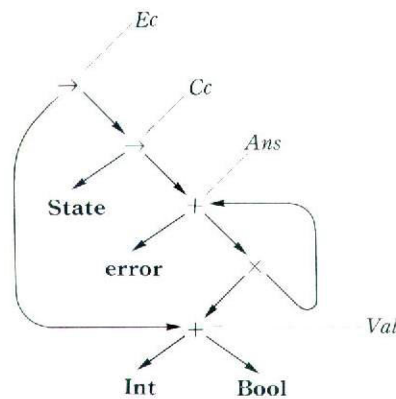


Fig. 2

Although there is a limitation of our typechecker, injection insertion turns out to be useful for practical problems. In fact, there is little chance of failure in injection by the above procedure.

§5 Conclusion

We have devised an algorithm for inserting injection operations to denotational specifications. Although local binding mechanisms such as "**let** ... **in** ...", or "... **where** ..." are not included in SDL described in this paper, extension of our 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 to

be no practical problems. If more general form of combination $(f\ e)$ had been allowed, the types of f and e could not be deduced from the type of $(f\ e)$ alone ; a functional type should be produced for $f=\lambda v.e'$ from little knowledge of the type of f. This can be done if we simply check the type consistency as in ML, but it is not the case for our purpose. From this observation, we have chosen a slightly restricted class of expressions for our specification language.

Mitchell[8] deals with polymorphic typechecking 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.

As the final remark, we have to state about the limitation of our algorithm. Although the value for practical usage has been demonstrated in this paper, we do not claim that our algorithm is complete or optimum. On the contrary, it fails easily for artificial problems as described in the previous section.

It is hoped, however, that this research has made a step toward a complete design of SDL and a more sophisticated semantics implementation system based on SDL. Another area of research is in extending this work to coercion insertion in functional languages. The applicability of our algorithm and the relations to other approaches, e. g., Ref. 4), should be studied further.

Acknowledgements

The author would like to thank NGC referees for many constructive criticism and suggestions on an earlier draft of the paper. Their advice have been invaluable in improving both the technical content and the presentation.

References

1) Bodwin, J., Bradley, L., Kanda, K., Litle, D. and Pleban, U., "Experience with an Experimental Compiler Generator Based on Denotational Semantics," *Proc. 1982 ACM Symp. on Compiler Construction, SIGPLAN Notices, Vol. 17, No. 6*, pp. 216-229, 1982.
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 Inf. Proc. Japan*, pp. 427-428, 1984, [in Japanese].
Also Cardelli, L., *Pascal VAX-Unix Version of Edinburgh ML*, converted from VMS by Nobuo Saito.
4) Futatsugi, K., Goguen, J. A., Jouannaud, J.-P. and Meseguer, J., "Principles of OBJ2." *Proc. 12th ACM Symp. on Principles of Prog. Lang.*, pp. 52-66, 1984.
5) Gordon, M. J. C., *The Denotational Description of Programming Languages*, Springer-Verlag, 1979.
6) Gordon, M. J., Milner, R. and Wadsworth, C. P., *Edinburgh LCF*, LNCS 78, Springer-Verlag, 1979.
7) Milner R., "A Theory of Type Polymorphism," *J. Comput. Syst. Sci., Vol. 17*, pp. 348-375, 1978.
8) Mitchell, J. C., "Coercion and Type Inference (Summary)," *Proc. 11th ACM Symp. on Principles of Prog. Lang.*, pp. 175-185, 1983.

9) Mosses, P., *SIS—Semantic Implementation System: Reference Manual and User Guide, DIAMI MD-30*, Dept. of Computer Science, University of Aarhus, Denmark, 1979.
10) Ohira, T. and Takeichi, M., "A Language Development System," *Proc. 28th Symp. of Inf. Proc., Japan*, pp. 329-330, 1984, [in Japanese].
11) Robinson, J. A., "A Machine-Oriented Logic Based on the Resolution Principle," *J. ACM, Vol. 12*, pp. 23-49, 1965.
12) Stoy, J. E., *Denotational Semantics*, MIT Press, 1977.
13) Wand, M., "A Semantic Prototyping System," *Proc. 1984 ACM Symp. on Compiler Construction, SIGPLAN Notices, Vol. 19, No. 6*, pp. 213-221, 1984.

Appendix

SDL specificaton

```
; Domains
((?(?))                                           ; ?={?}
 (Int(Int))                                       ; Standard domains Int
 (Bool(Bool))                                     ; and Bool
 (State(State))                                   ; Abstract domain State
 (Cmd(+(cmd-seq(* Cmd Cmd))(cmd-output Exp)))     ; Cmd
 (Exp(+(exp-int Int)(exp-bool Bool)))             ; Exp
 (Cc(->State Ans))                                ; Command continuations
 (Ec(->Val Cc))                                   ; Expression continuations
 (Ans(+("error")((* Val Ans))))                   ; Answers
 (Val(+(Int)(Bool))))                             ; Values
; Functions
((pair(->(0)(->(1)(* (0)(1)))))                   ; pair: α→β→α×β
 (C(-> Cmd(-> Cc Cc))                             ; Semantic function C
  (lambda(cmd c)
    (case cmd
      (cmd-seq((pair cmd1 cmd2))(C cmd1(C cmd2 c)))
      (cmd-output(exp)(E exp(lambda(v s)
                              (pair v(c s)))
                          ))))
  )
 (E(-> Exp(-> Ec Cc))                             ; Semantic function E
  (lambda(exp k)
    (case exp(exp-int(e)(k e))(exp-bool(e)(k e))))
  ))
```

Transformed SDL specification

```
((?(?))
 (Int(Int))
 (Bool(Bool))
 (State(State))
 (Cmd(+(cmd-seq(* CmdCmd))(cmd-output Exp)))
 (Exp(+(exp-int Int)(exp-bool Bool)))
 (Cc(-> State Ans))
 (Ec(-> Val Cc))
```

```
(Ans(+(Ans1(error))(Ans2(* Val Ans))))          ; Summand tags Ans1, Ans2,
(Val(+(Val1 Int)(Val2 Bool))))                  ; Val1, and Val2 have been
((pair(-> (0)(-> (1)(* (0)(1)))))               ; generated.
(C(-> Cmd(-> Cc Cc))
  (lambda(cmd)                                  ; Lambda-binding has been
    (lambda(c)                                  ; normalized.
      (case cmd
        (cmd-seq((pair cmd1 cmd2))(C cmd1(C cmd2 c)))
        (cmd-output(exp)
            (E exp
              (lambda(v)
                (lambda(s)
                  (Ans2                         ; Ans2: Val×Ans→Ans
                  (pair v(c s))))
                )
              ))))
      )
  )
(E(-> Exp(-> Ec Cc))
  (lambda(exp)
    (lambda(k)
      (case exp
        (exp-int(e)(k(Val1 e)))                 ; Val1: Int→Val
        (exp-bool(e)(k(Val2 e)))))              ; Val2: Bool→Val
      )
  ))
```