HEWLETT
PACKARD

The HP-SL Type Model

Patrick Goldsack
Software Engineering Department
HP Laboratories Bristol
HPL-91-73
August 1991

Formal specification,
type systems

Abstract

The report provides a detailed account of the semantics of the type system of HP-SL. It does this in two ways. The first is informally by example, it provides an outline of the two forms of identifying types and discusses their meaning. The second way is to show that the types of HP-SL form a term algebra, with every type mapping to a specific term of that algebra. Every HP-SL construct to do with types is then given an interpretation by means of the mapping to the algebra and thus provides a semantics for this part of HP-SL. The notion of type correctness is also explained in terms of the mapping to the algebra.

Contents

1 Background

The aim of this report is semantic; to describe in some detail the type model used within HP-SL. This should be sufficient for users of the language to gain an intuitive understanding of the various language constructs, and in particular the relationship between the two forms of type definition — the 'type' and the 'syntype' definitions.

It is hoped that the semantic descriptions are given in a way that is not too theoretical, yet completely unambiguous.

2 Informal Description

Every type in HP-SL can be referenced using a type expression. Type expressions have two different forms.

- the name of a type
- the application of a type constructor to one or more type expressions.

Note that subtype expressions are not being considered at this point.

To understand the following description of the type model more fully, a few concepts need to be defined

- simple types — the fundamental building blocks of the type model; the simple types are the only ones not built by the use of type constructors
- type constructors — functions which create a type from other types
- type synonyms — an association of names with types, not creating new types in the type model.

2.1 Simple Types

The best way of describing a 'simple type' in HP-SL is to show how they are defined and how they relate to existing types. The syntax is as follows (other more complicated forms available in HP-SL can all be mapped into this form)

```
type typename
```

In the above, the terminal 'type' is a keyword introducing the simple type. The nonterminal 'typename' represents the name of the type.

This declaration has two consequences. Firstly it introduces a new simple type; that is a type different from all other types be they pre-defined, defined using a simple type declaration, or created by the application of a type constructor to any type.

Secondly, it introduces a name by which this type may be denoted; in this case 'typename'. Thus all simple types are named and this is the only way (bar the introduction of a synonym) by which this type may be referenced.

Consider the following examples

```
type Colour

type Words
```

Each type is distinct; thus expressions of the form

```
x:Colour = y:Words
```

are ill-typed.

Note that nothing in the type declarations ensures that the types represent values in the domain implied by their names. Thus nothing as yet insists that the only values of type 'Colour' are indeed colours. HP-SL provides an assertional style of specification which enables such constraints to be placed but this is not the concern of this report.

2.2 Constructed Types

A type constructor is a function which 'creates' or perhaps more accurately 'gives reference to' another type when applied to one or more type parameters.

Note that every type constructor

- produces types different from all other type constructors and simple types
- produces a different type for each different set of type parameters,

Thus every type is denoted by exactly one term constructed out of the simple types and the type constructors.

The syntax for these type constructors is as follows

```
type (| typevar, typevar |) typename
```

The terminal 'type' introduces the type constructor declaration; the difference between the simple type and the constructor being the 'typevar', these are formal parameter names given to the formal type parameters of the constructor.

Consider the following examples

```
type (|Element|) Stack

type (|Key, Data|) Assoc_list
```

The first is a function from types (e.g. Colour) to the types 'Stack' of that type (e.g. Stack Colour).

In a similar way, the second is a function from two types to a type 'Assoc_list of' those types. For example

$$Assoc\_list\ Student\ Grade$$

which could be a type representing the mapping between students and examination grades.

As with simple types, these constructed types carry no inherent properties to ensure that the types behave as one might expect from their names. Nothing in the type declaration ensures that 'Stack Colour' is indeed a stack of colours. This fact is of no importance to the type model, however, as the model makes no attempt to capture the values or properties of the values in each type — this being a feature of the mapping between the type model and the models of the specification.

2.3 Naming Types — Synonym Types

From the above it is clear that some types do not have an identifier; their only representation is via a type expression. However, there are several reasons why it is useful to have a name for these types

- to give a useful hint as to how the type is being used in a particular specification
- to shorten the repeated use of a complex type expression

This is done via the synonym type declaration. The general form of such a declaration is

$$\textbf{syntype}\ syntypename \triangleq type\_expression$$

The terminal 'syntype' introduces the declaration which binds the name 'syntypename' to the type denoted by 'type_expression'.

This does not introduce a new type to the type model, it merely adds a name for an existing type into an environment of syntypes (a mapping between syntype names and the types which they denote).

HP-SL also allows the provision of synonym constructors, constructors for whole classes of synonyms. Thus the class of binary relations could be defined as

$$\textbf{syntype}\ (\!|A,B|\!)\ Relation \triangleq Set(A \times B)$$

The application of the synonym produces a type equivalent to the substitution of the type variables for the type parameters in the constructor application. Thus

$$Relation\ Int\ Bool = Set(Int \times Bool)$$

3 Models and Type Models

Before embarking upon the description of the HP-SL type model, a distinction between the notion of 'model' and the notion of 'type model' is required to avoid confusion.

Semantic descriptions of languages are often described in terms of 'models' of a 'syntactic term' of the language. This is particularly true of specification languages where loose specification allows the possibility of having many models, whereas typically in programming languages each program denotes a single model.

A specification in HP-SL is

- a set of types
- a set of functions and constants defined on these types

In outline, the notion of model in HP-SL is

- a set of sets (sometimes called 'domains'), one for each type in a specification,
- a set of functions and constants operating on the appropriate 'domains' denoting the functions or constants defined in that specification.

There is no restriction within a model on the relationship between the sets representing the types. Thus the set representing the type 'Int' and the set representing the type 'Char' may be related in some way — perhaps by encoding the characters (eg ASCII or EBCDIC) within the integers.

In a type model, on the other hand, these two types would be kept completely separate, considered as distinct members of the type model. The type model is a reflection of the type rules of the language — a direct mapping between the terms of the language and the concepts they denote.

The mapping between the syntax and the type model captures the notion of type equivalence and through this the notion of type correctness (more of this later).

The mapping between the syntax and the type model is described in this report. The next step of defining the set of models denoted by the specification is left to a definition of the language semantics. Thus the type model describes the set of types available to a user of the language; those types that a user is capable of denoting by using valid terms of HP-SL.

Intuitively, an HP-SL type is a collection of 'terms[1]' However, these collections must obey certain simple properties

- every valid term is the member of some type
- no two types have terms in common.

These two conditions between them state that the universe of values (as represented by valid terms) in HP-SL is *partitioned* into a set of types. This property makes the HP-SL type model extremely simple and

[1] roughly a syntactic expression within the grammar of the notation

makes the notion of type correctness a simple matter. Note that the property of partitioning is lost by the time the models of HP-SL are concerned; it is only a property of the type model.

A particular feature of HP-SL that makes the type model so simple is the lack of recursive type equations. A roughly equivalent effect may be obtained by the use of types and functions. For example

```
type Tree
type Node

fn mk_tree: Node × Node → Tree
fn left_node: Tree → Node
fn right_node: Tree → Node
```

4 Notation

First the terminology needs to be explained.

A type model is a pair, an environment of declared synonyms (Env) and a type universe (TU).

4.1 Type Universe

A type universe also consists of a pair, a generator set and the set of *sorts* generated by that set (sort universe). The term *sort* is used in the algebraic sense; put simply a sort is a 'place-holder' for a set of values in a model of a specification.

The symbol 'ρ' will be used to represent a typical member of the set of type universes; 'ρ_s' will represent the associated sort universe and 'ρ_g' the associated generator set.

We define a sort as a finite term of the following grammar.

```
1       <Sort>  ::=  <Id>
2               |    ( <Sort> <Sort> )
```

Where 'Id' is taken from a set of identifiers.

The construction of the type universe is as follows.

The generator set 'ρ_g' is an indexed set of maps 'ρ_i, i $\geq$ 0', where each 'ρ_i' is the set of type constructors with 'i' type parameters; the special case 'ρ_0' is the map of simple types to their associated sorts.

Thus

ρ_0 = [Complex $\mapsto$ Complex, Bool $\mapsto$ Bool, ...]
ρ_1 = [Set $\mapsto$ λ T. (Set T), Seq $\mapsto$ λ T. (Seq T), ...]
ρ_2 = [Pair $\mapsto$ λ T. λ S. ((Pair T) S), ...]

Let 'U_n, $n \geq 0$' be an indexed set of sorts defined by

U_0 = rng ρ_0
$U_n = \bigcup_i$ { reduce(...(reduce(ρ_i(c), t_1)...),t_i) | c $\in$ dom ρ_i, t_1, ..., $t_i \in U_{n-1}$ }

where 'reduce' is the λ-calculus β-reduction.

Thus

U_0 = { Complex, Bool, ... }
$U_1 = U_0 \cup$ { (Set Complex), (Set Bool), ((Pair Complex) Bool), ... }
$U_2 = U_1 \cup$ { (Set (Seq Complex)), (Seq ((Pair Bool) Complex)), ... }
etc.

Let the sort universe 'ρ_s' be defined as

$\rho_s = \bigcup_n U_n$

The pair '(ρ_g, ρ_s)' is the type universe; the indexed set of simple types and type constructors ρ_g, and the set of sorts, the sort universe generated by this set of constructors. The operation which converts an indexed set of constructors to the set of sorts is 'closure'.

ρ_s = closure(ρ_g)

4.2 Environment

An environment is a mapping between

- syntype identifiers and their definition, namely 'Id $\mapsto$ Sort'.
- syntype constructor identifiers and their definition, namely 'Id $\mapsto$ λ 'Tid' . Sort' (where 'Sort' may contain references to 'Tid').

The set of valid environments is called 'Env' and the symbol 'σ' will be used to represent such an environment.

4.3 Other Notation

The brackets '[ ]' denote the 'meaning' of the syntactic term within them. They will always be written as follows

[*syntax*]other_param = meaning

where 'other_param' will typically be an environment, a type universe or both. Note that the syntactic term will be in italic font to distinguish it from other text in the equation.

5 Scope Rules

HP-SL is a normal block-structured language with constructs which introduce new scopes. For the convenience of this paper, it is assumed that the specification has undergone 'α-conversion'; that all names have been made unique and all references have been resolved. This simplifies the notion of environment as the nested scopes need not be modelled using a stack of environments — rather a single flat environment may be assumed.

The α-conversion also enables other simplifying assumptions in the sequel. These are indicated wherever relevant. The alpha-conversion rules are defined in [1].

6 Semantics

The meaning of various HP-SL syntactic terms can now be analysed. In particular, the following can now be described in terms of effects or dependencies on the type model

1. type expressions
2. type declarations
3. syntype declarations

6.1 Type Expressions

The type of the meaning function for type expressions is

Type_expr $\rightarrow$ (Env $\times$ TU) $\rightarrow$ Sort[2]

The semantics of type expressions are the simple conversion from the HP-SL syntactic terms, to the sort in the obvious way.

Dealing first with identifiers,

$[\![id]\!](\sigma, \rho) = \sigma(id)$ if id $\in$ dom σ.

$[\![id]\!](\sigma, \rho) = \rho_i(id)$ if id $\in$ dom ρ_i

$[\![id]\!](\sigma, \rho) = id$ otherwise

[2] the notion of Sort used here is extended to include free non-underlined type identifiers to represent uninstantiated type variables. This is done to allow the meaning function for type expressions to be used on the body of type constructors.

The last case is used to represent the type variables in an expression, these remain for later binding by a lambda (see also footnote 1).

Next, the semantics of the application of type and synonym constructors must be defined,

$[\![type\_expr_1\ \ type\_expr_2]\!](\sigma,\rho) = \text{reduce}([\![type\_expr_1]\!](\sigma,\rho), [\![type\_expr_2]\!](\sigma,\rho))$

where 'reduce' is the lambda-calculus β-reduction, with the first argument being the lambda. If reduction is not possible (i.e. if $type\_expr_1$ does not represent a constructor), the composition of the two type expressions is not well formed.

Furthermore, a type expression type identifiers not within the scope of a lambda binding does not represent a term in the sort term algebra and so does not represent a type. It may safely be assumed that the process of α-conversion has rejected such illegally scoped specifications.

6.2 Type Declarations

In HP-SL, the notion of type declaration covers all forms of syntax starting with the keyword 'type' — including all those followed by injector and projector definitions. These are merely derived forms of the simplest case considered here and so are fully defined by the semantics given here.

The type of the meaning function for type declarations is

Type_decl $\rightarrow$ (Env $\times$ TU) $\rightarrow$ (Env $\times$ TU)

There are two cases to consider, simple types and type constructors.

$$
\begin{array}{l}
[\![type\ id\ ]\!](\sigma,(\rho_g,\rho_s)) = \\
\quad (\sigma, \\
\quad (\rho_g \dagger (\rho_0 \cup [id \mapsto \underline{id}]), \\
\quad\ \text{closure}(\rho_g \dagger (\rho_0 \cup [id \mapsto \underline{id}])) \\
\quad) \\
\quad)
\end{array}
$$

$$
\begin{array}{l}
[\![\text{type } (|\ \text{id}_1, ..., \text{id}_n\ |)\ \text{id}\ ]\!](\sigma,(\rho_g,\rho_s)) = \\
\quad (\sigma, \\
\quad (\rho_g \dagger (\rho_n \cup [id \mapsto \lambda id_1 .... \lambda id_n .(...(\underline{id}\ id_1), ...id_n)]), \\
\quad \text{closure}(\rho_g \dagger (\rho_n \cup [id \mapsto \lambda id_1 .... \lambda id_n .(...(\underline{id}\ id_1), ...id_n)])) \\
\quad) \\
\quad)
\end{array}
$$

Note that the environment of syntypes is not changed, whilst the closure of the valid sorts is reset to take account of the additional type or type constructor.

Because of α-conversion we may ignore issues of scope and the existence of the definitions of the various identifiers.

6.3 Type Synonym Declarations

The type of the meaning function for syntype declarations is

Syntype_decl $\rightarrow$ (Env $\times$ TU) $\rightarrow$ (Env $\times$ TU)

There are two cases to consider, synonyms and synonym functions.

$[\![\mathit{syntype}\ \mathit{id} \triangleq\ \mathit{type\_expr}\ ]\!](\sigma, \rho) = (\sigma\dagger[\mathit{id} \rightarrow [\![\mathit{type\_expr}\ ]\!](\sigma, \rho)],\ \rho)$

$[\![\mathit{syntype}\ (|\ \mathit{id}_1,\ ...,\ \mathit{id}_n\ |)\ \mathit{id} \triangleq\ \mathit{type\_expr}]\!](\sigma, \rho) = (\sigma\dagger[\mathit{id} \rightarrow \lambda\ \mathrm{id}_1.\ ...\ .\ \lambda\ \mathrm{id}_n\ .[\![\mathit{type\_expr}\ ]\!](\sigma, \rho)],\ \rho)$

The evaluation of the meaning of the *type_expr* will ensure that the type variables will remain untouched. The process of α-conversion guarantees that the type variables will all be uniquely named and hence there is no need to alter the environment and type model to cater for these when using them on the right-hand side of the equation.

Note also how recursion is effectively barred from syntypes as it is impossible to refer to the newly introduced syntype in the type expression.

7 Type Compatibility

Two type expressions are 'compatible' if, and only if, they denote the same sort within the sort universe.

Thus a specification is type correct if, and only if, the expected types (for function parameters and for value bindings) and the actual types are compatible.

Note that no mention of sub-types has been made. This is because type expressions with invariants denote (in the models) sets of values entirely taken from the base type. Thus the 'sort' of these values is the same as that for the base type.

It might be possible to move to a scheme with ordered sorts to handle sub-types. However the current way of defining sub-types through predicates makes the ordering undecidable, so the benefit for type-checking seems limited — so complicating the type model is unnecessary if the specification is viewed as a definition of the decidable type rules.

The notion of type compatibility therefore takes no account of the sub-type orderings.

8 References

[1] Patrick Goldsack. HP-SL type and scope rules. Technical report, Hewlett-Packard Laboratories. In production.