HEWLETT
PACKARD

HP-SL Abstract Type Syntax (Records and Unions)

Patrick Goldsack
Software Engineering Department
HP Laboratories Bristol
HPL-91-69
August 1991

Formal specification

Abstract

The report outlines the derived syntax provided within the HP-SL specification language to enable the definition of record and union types. It describes the notation at two levels. First it describes the requirements for such a notation by way of an example the informally defines the derived short-hand notation. Secondly it provides a detailed definition by way of axiom schemas of the translation between the derived notation and the base HP-SL notation.

Contents

1 Background

Early versions of HP-SL had type constructors for records and unions in a similar way to traditional VDM notations. However, these are strictly not necessary in a language with abstract types as both are easily codified within this concept.

As a result, the current version of HP-SL has dropped these to simplify the language semantically. This paper aims to show how HP-SL now deals with concepts such as record and union types by way of its abstract types, and the derived syntax that is introduced to make this easier.

2 Abstract Types

HP-SL provides the capability of defining abstract types: types which have no internal structure other than that implied by the functions operating on that type. It is also able to define type constructors; functions which generate families of types — one for each type parameter. For further explanation of this aspect of the language, see [1] and [2].

2.1 A running example

As an example which will be carried forward to the remainder of the paper, consider the family of 'trees' generated by the type constructor

```
type (|T|) Tree
```

In other notations, one might state that a tree is the union of three alternatives; an empty tree, a leaf containing a value of type 'T', or a node containing two sub-trees. In an abstract type language, however, there is no union construction merely the ability to define functions and values. Thus the representation in HP-SL would be

```
val (|T|) empty : Tree T
fn (|T|) leaf: T → Tree T
fn (|T|) node: (Tree T) × (Tree T) → Tree T
```

The three types of tree are constructed using two different functions and a value.

2.2 Constructor functions

Every abstract type has a set of values and function, known as 'constructors', which construct all possible values of the abstract type. Clearly any function which returns values of the abstract type could be included, so the choice of a particular set as the constructors is a matter of instinct, taste or even arbitrary decision. To act as a guide, consider the following examples:

- The tree type has three constructors, 'empty', 'leaf' and 'node'. A function to return the left subtree of a node would probably not be considered a constructor as that subtree could be expressed in terms of the other three; yet adding it would not allow the removal of one of the other three constructors.
- For lists, the set of constructors would probably be 'empty' and 'cons'. Again, 'append' would not be a suitable constructor. An alternative to 'cons' might be an 'add_element_to_back_of_list' function.

2.3 Projector functions

For every constructor function it is possible to define a set of functions to 'pull apart' the constructed value — typically to return each of the components of the value injected into the type. Thus, for the tree example,

the constructor function 'leaf' might be associated with a function 'leaf_val' to extract the value injected into the tree by 'leaf'. Similarly, 'node' might be accompanied by two projectors, one to extract the left subtree and one to extract the right subtree of a node.

So for this example,

```
fn ⦇T⦈ leaf_val: Tree T → T is
  leaf_val t
    pre
      ∃ v:T· t = leaf v
    return v
    post
      t = leaf v

fn ⦇T⦈ left_subtree: Tree T → Tree T is
  left_subtree t
    pre
      ∃ (t1: Tree T, t2: Tree T)·
        t = node(t1, t2)
    return t1
    post
      ∃ (t2: Tree T)·
        t = node(t1, t2)
```

and so on for all functions projecting component values of the tree type.

Generally a projector is defined to be a function which extracts one of the parameters that was used to construct the value in the first place. This relationship is easy to give directly using an assertion.

```
assert ⦇T⦈ constructor_axioms ≜
  (∀ v: T· leaf_val(leaf(v)) = v)
  ∧
  (∀ t1 and t2: Tree T·
      left_subtree(node(t1, t2)) = t1
      ∧
      right_subtree(node(t1, t2)) = t2
  )
```

2.4 Predicates — 'is' functions

Each of the projector functions above are defined with a pre-condition to ensure that the value to which it is applied was built (or could have been built) using the appropriate constructor.

These pre-conditions could easily be converted into predicates, such as

```
fn ⦇T⦈ is_leaf: Tree T → Bool is
```

$$is\_leaf\ t \triangleq (\exists\ v{:}T \cdot\ t = leaf\ v)$$

Throughout the remainder of this document functions such as these are termed 'is' functions.

2.5 Updators — 'set' functions

For every projector, one could define a function which changes just the value of that projector when applied to a value of the type (appropriately constructed), leaving the values of the other projectors unchanged.

For example

fn $(\!|T|\!)$ *set_left_subtree: Tree T → Tree T → Tree T* is
set_left_subtree t t_1
pre *is_node t_1*
return t_2
post
left_subnode t_2 = t ∧
right_subnode t_2 = right_subnode t_1

This function takes two trees (curried) and sets the left subtree of the second tree to be equal to the first tree.

The reason for the currying of the parameters is that one can compose partially applied set functions to produce a function which 'sets' several projectors simultaneously. This leads to a cleaner and clearer presentation of multiple 'sets'. For example

$$(set\_proj1\ new\_p1\ \circ\ set\_proj2\ newp2)\ r$$

This generalises to as many 'sets' as required.

2.6 Axioms

There are certain common properties of constructor functions that still need to be established. Currently there is no statement that

- all trees may be constructed using these functions and values
- the constructor functions and values generate different trees, thus it is possible for 'leaf(4) = empty' to be true

These two axioms are now discussed more fully, plus another which is important when reasoning about abstract types.

2.6.1 No Junk

When defining an abstract type with a given set of constructor functions, it is necessary to say that the chosen set is 'complete', that all elements of the type may be finitely constructed with that set of constructor functions.

In the case of the 'Tree' type constructor, the constructor functions are 'empty', 'leaf' and 'node'. If these are the only ones, the following axiom should be added

$$
\begin{array}{l}
\textbf{assert } (\!|T|\!)\ no\_tree\_junk \triangleq \\
\quad (\forall\ t\text{: } Tree\ T\cdot \\
\qquad is\_empty\ t \vee \\
\qquad is\_leaf\ t \vee \\
\qquad is\_node\ t)
\end{array}
$$

using the 'is_' functions defined as for the 'is_leaf' definition in section 2.4.

2.6.2 Induction

In addition we should provide the structural induction axiom to capture the notion of 'finitely generated' and to enable the use of structural induction as a proof technique when reasoning about a type (i.e. case analysis proofs). Note that the no junk axiom is implied by the induction axiom, which insists in addition that the values of the abstract type are 'finitely generated'.

$$
\begin{array}{l}
\textbf{assert } (\!|T|\!)\ tree\_induction \triangleq \\
\quad (\forall\ p\text{:}Tree \rightarrow Bool\cdot \\
\quad ((p\ empty) \\
\qquad \wedge\ (\forall\ v\text{:}T\cdot\ p(leaf\ v)) \\
\qquad \wedge\ (\forall\ t_1 \textbf{ and } t_2\text{:}Tree\ T\cdot\ p(t_1) \wedge p(t_2) \Rightarrow p(node\ (t_1, t_2)))) \\
\quad \Rightarrow \\
\quad (\forall\ t\text{: } Tree\ T\cdot\ p\ t) \\
\quad)
\end{array}
$$

2.6.3 Equalities and No Confusion

The final set of axioms which should be considered are those regarding the equalities between two constructions — in this case between two trees. The most common equality axiom is that no two terms of the same type constructed using the constructor functions denote the same value (in the models) unless the two terms were themselves identical. Identical terms must *always* be equal (assuming that they are fully specified).

$$
\begin{array}{l}
\textbf{assert } (\!|T|\!)\ no\_tree\_confusion \triangleq \\
\quad (\forall\ t\text{: } Tree\ T\cdot \\
\qquad (is\_empty\ t \Rightarrow \neg\ (is\_leaf\ t \vee is\_node\ t)) \\
\qquad \wedge \\
\qquad (is\_leaf\ t \Rightarrow \neg\ (is\_empty\ t \vee is\_node\ t)) \\
\qquad \wedge
\end{array}
$$

$$(is\_node\ t \Rightarrow \neg\ (is\_empty\ t \vee is\_leaf\ t))$$
)

3 Short-hand Description

The examples of section 2 demonstrated that defining the properties of an abstract type requires several functions and axioms to be written on that type. However two forms of abstract type definition are so common and their properties so well understood that a syntactical shorthand is provided which allows the simultaneous definition of such abstract types and their associated functions and axioms.

3.1 Constructor short-hand

HP-SL provides a syntactic mechanism by which this may be done. Consider

$$\begin{array}{l}\textbf{type}\ (\!|T|\!)\ \mathit{Tree} \triangleq \\ \quad [\mathit{empty}] \mid \\ \quad [\mathit{leaf}\colon T] \mid \\ \quad [\mathit{node}\colon \mathit{Tree}\ T \times \mathit{Tree}\ T]\end{array}$$

The syntax defines the following

- that there are three constructors, *empty*, *leaf* and *node*; each set of bracketing [ ] introduces a constructor, and each of these is seperated by a |.
- *empty* is a constant value; if no type is defined after the constructor name, the constructor is a constant
- *leaf* is a constructor function which takes a value of type 'T' into *Tree T*

  if a type is given after the name of the constructor, this is the type of the parameters it injects into the abstract type
- *node* is a constructor function which takes a pair of *Tree T* into a *Tree T*
- the functions *is_empty*, *is_leaf* and *is_node* are defined with the appropriate properties (for details see section A.2.4)
- the no junk and no confusion axioms are automatically defined when appropriate (for details see section A.2.6)

This syntax has clearly shrunk a large multi-line specification into a single definition.

3.2 Projector short-hand

There are several projectors that could be defined (see 2.3)

- 'leaf_val' — which delivers the value of type 'T' used to construct the leaf
- 'left_subtree' — which delivers the left subtree used to construct the node
- 'right_subtree' — which delivers the right subtree used to construct the node

- a whole set of compound projectors, such as one which delivers both subtrees of a node; in this example such projectors will not be defined

This may easily be accomodated within the syntactic form outlined above.

$$
\begin{array}{l}
\textbf{type}\ (\!|T|\!)\ \mathit{Tree} \triangleq \\
\quad [\![\mathit{empty}]\!] \mid \\
\quad [\![\mathit{leaf} \triangleright \mathit{leaf\_val}\colon T]\!] \mid \\
\quad [\![\mathit{node} \triangleright \\
\quad\quad (\mathit{left\_subtree}\colon \mathit{Tree}\ T, \\
\quad\quad\ \mathit{right\_subtree}\colon \mathit{Tree}\ T)\ ]\!]
\end{array}
$$

The syntax can be explained as follows

- if there is no type or no '$\triangleright$' symbol, the constructor is a constant (e.g. 'empty')
- if the constructor is followed by a type, it is a function as before (ie 'leaf')
- if the constructor is followed by '$\triangleright$', the constructor is a function which takes values matching the pattern which immediately follows the '$\triangleright$'. Thus 'node' is a function of type $\mathit{Tree}\ T \times \mathit{Tree}\ T \rightarrow \mathit{Tree}\ T$, and 'leaf' a function of type $T \rightarrow \mathit{Tree}\ T$
- every identifer in the pattern following a '$\triangleright$' is a projector function, projecting the appropriate component from the abstract type, in this case 'Tree T'. Note that the existance of *any* projector (even an anonymous one) ensures that different values for the parameter of the constructor produces different values of the type. Thus

$$
\begin{array}{l}
\textbf{assert}\ (\!|T|\!)\ \mathit{surjective\_node} \triangleq \\
\quad (\forall\ t_1\ \textbf{and}\ t_2\ \textbf{and}\ t_3\ \textbf{and}\ t_4\colon \mathit{Tree}\ T \cdot \\
\quad\quad \mathit{node}(t_1, t_2) = \mathit{node}(t_3, t_4) \\
\quad\quad\quad \Leftrightarrow \\
\quad\quad t_1 = t_3 \wedge t_2 = t_4 \\
\quad)
\end{array}
$$

- every constructor function without user-defined projectors (determined by the lack of a $\triangleright$ in the definition), merely have an constructor. There is therefore no axiom relating the projectors and constructors thus impling that the constructor may 'confuse' two parameters. Thus had the type Tree been written

$$
\begin{array}{l}
\textbf{type}\ (\!|T|\!)\ \mathit{Tree} \triangleq \\
\quad [\![\mathit{leaf}\ :\ T]\!] \mid \ldots
\end{array}
$$

  it would have been legal for the function 'leaf' to take two different vlaues of type T to the same leaf-tree.

- every (non anonymous) projector has a 'set' function associated with it, to modify just the one projector leaving the other untouched. Its name is obtained by adding 'set_' in front of every projector. The 'set' functions are curried; thus 'set_left_subtree' has type '$T \rightarrow \mathit{Tree}\ T \rightarrow \mathit{Tree}\ T$ '.
- 'empty' does not have a projector

3.3 Layering projectors

Since a pattern is used after the '▷', layered patterns may be used to provide projectors that extract values from the abstract type in various ways; thus for the 'Tree' example, one could have written

```
type ⦇T⦈ Tree ≜
        ... |
        [ node ▷
            whole as
              ( left_subtree : Tree T,
                right_subtree: Tree T ) ]
```

Here the function 'whole' returns the pair of values from a node-constructed tree. As before, the 'set_whole' function is also defined, although in this case it is equivalent to the 'node' function itself.

The signatures of the three projector functions are

```
fn ⦇T⦈ whole: Tree T → ( Tree T × Tree T )
fn ⦇T⦈ left_subtree: Tree T → Tree T
fn ⦇T⦈ right_subtree: Tree T → Tree T
```

and they satisfy the obvious axiom

```
assert ⦇T⦈ layering_axiom ≜
    (∀ t: Tree T·
       whole t = ( left_subtree t, right_subtree t ))
```

3.4 Pre-conditions for constructors

The syntax for abstract types uses a pattern after the ' ▷ '. These patterns may be sub-typed using any of 'inv', 'sat' or an infix predicate such as '∈'.

For example

```
type Limits ≜ [ limits ▷ ( lower:Int, upper: Int ) sat lower < upper ]
```

Informally, such typings may be seen as pre-conditions on the constructor function, thus in this example

```
fn limits: Int × Int → Limits is
   limits (l,u)
      pre l < u
```

Use of pattern typings in this way also has an effect on the use of set functions. The predicates also generate a pre-condition on the set function that the appropriate resultant value is in the type. If not, the set function's behaviour is unspecified. Thus in this example

```
fn set_lower: Int → Limits → Limits is
   set_lower l limit
      pre l < upper(limit)
```

This limits the use of set function when many projectors are related by invariant. The type declaration should be grouped and layered to provide simultaneous set functions, or a parent type defined with each projector independant, then the actual type declared as a sub-type of it. For example,

```
type Limits_base ≜ [[ limits ▷ (lower: Int, upper: Int) ]]
syntype Limits ≜ Limits_base inv l· lower l < upper l
```

It is worth noting the following fact about the limit example. The pattern which defines the projectors provides identifiers with two distinct scopes and in these scopes the identifiers have distinct types.

The first scope is that of the pattern itself, including any 'sat' clause that may be attached to the pattern. Within this scope the identifiers have the type defined in the pattern, hence the example

```
type Limits ≜ [[ limits ▷ (lower:Int, upper: Int) sat lower < upper ]]
```

compares lower and upper as two integers within the 'sat' clause.

The second scope is that of the externally visible identifiers that are defined for use within the remainder of the specification. Thus in other definitions the identifiers

```
fn lower: Limit → Int
fn upper: Limit → Int
```

are available.

4 Axiom Control

4.1 Allowing Junk

At times it is convenient to provide only part of the definition of an abstract type; supplying the remainder at a later stage. This is enabled by the removal of the 'no junk' axiom mentioned in section 2.6.1.

To do this, the syntax is extended by allowing the sequence of alternative constructors to include '...'. For example

$$\textbf{type}\ \mathit{State} \triangleq [\![\mathit{stacks}\colon \mathit{Stack\_id} \xrightarrow{m} \mathit{Stack}]\!] \mid \ldots$$

The meaning of the '...' is 'and others, not yet specified'.

Note that the no confusion axiom still holds; that the alternatives so far provided are not confused with each other. No guarantee is given regarding the alternatives not provided.

Thus the following definition of 'Colour' may be given

$$\textbf{type}\ \mathit{Colour} \triangleq [\![\mathit{red}]\!] \mid [\![\mathit{green}]\!] \mid [\![\mathit{black}]\!] \mid \ldots$$

and each of the three colours are different. However, the definition can be extended by

$$\textbf{type}\ \mathit{Colour} \triangleq [\![\mathit{rouge}]\!] \mid [\![\mathit{vert}]\!] \mid [\![\mathit{noir}]\!] \mid \ldots$$

These are again different from each other, but not necessarily from the first three colours.

4.2 Allowing Confusion

In addition to allowing junk, at times it might be useful to allow confusion between the various constructors. Take, for example, sets. Sets have no specific constructor function — perhaps the nearest equivalent is

$$\textbf{type}\ (\!|T|\!)\ \mathit{Set} \triangleq [\![\mathit{empty}]\!] \mid [\![\mathit{singleton}\colon T]\!] \mid [\![\mathit{union} : \mathit{Set}\ T \times \mathit{Set}\ T]\!]$$

The problem is that 'union' can produce any set including any of the singleton sets and the empty set. This means that the construction defined above is wrong because the no confusion axiom is present.

(Note that we already have that two different terms containing the 'union' constructor could denote the same set, thus the symmetric nature of 'union' is allowed.)

To enable the removal of the 'no confusion' axiom, the keyword 'overlapping' may be used directly after the '$\triangleq$'. Thus the correct formulation for the set definition is

$$\textbf{type}\ (\!|T|\!)\ \mathit{Set} \triangleq \textbf{overlapping}\ [\![\mathit{empty}]\!] \mid [\![\mathit{singleton}\colon T]\!] \mid [\![\mathit{union} : \mathit{Set}\ T \times \mathit{Set}\ T]\!]$$

and the equalities and inequalities have to be added explicitly.

Note that the formulation of the 'is' function is such that

$$is\_singleton(\, union(\, empty,\ singleton\ 4\,)\,)$$

would be true (assuming that the correct definition of equality had been provided) even though the top-level constructor is not 'singleton'.

4.3 Eliminating Constructor Functions

HP-SL provides the ability to only partially specify the set of projectors associated with a constructor function or to not specify the form in which they occur in the domain type of that constructor function.

As an example, consider the following

```
type (| T |) Tree ≜
    [[ empty ]] |
    [[ leaf: T ]] |
    [[ node
       ▷ left_subnode: Tree T
       ▷ right_subnode: Tree T
     ]]
```

Notice that there are now two projector patterns for the constructor node, one for each of the subnodes. This form underspecifies the type of the constructor 'node'; the domain type is such that the two projected values are precisely represented in some simple form (the type is isomorphic to the 2-tuple type), but the precise type is not known. This in turn means that 'node' may not be used as a function directly — its type is not known. In fact the constructor function 'node' is not defined at all, only the predicate 'is_node' is defined by such a declaration. This still allows indirect forms of defining values where 'is-node' would be true.

If in addition one wishes to state that other projectors might exist one of the patterns following a '▷' may be '...'. This indicates that more may be present. For example

```
type (| T |) Tree ≜
      [[ empty ]] |
      [[ leaf: T ]] |
      [[ node
           ▷ left_subnode : Tree T
           ▷ ...
       ]]
```

states that 'node' may have projectors in addition to the function 'left_subnode'.

5 Expressions

The values and functions defined in parallel to the abstract type may be used as normal — thus the value 'empty' may be used freely in expressions; as may the function 'leaf', 'project_leaf', is_leaf' 'node', and so on. Their types are deduced from the type declaration as described above. For example all of the following are allowed

```
val a ≙ empty

val b ≙ leaf 4

val c ≙ if project_leaf b = 4
            then if is_empty a
                     then node(empty, empty)
                     else node(empty, node(leaf 5, leaf 6))
            else empty
        endif
```

However, a slightly different style of defining a value of the type is possible. Consider the following

```
let
  val t: Tree Int sat
      is_node t ∧
      left_subtree t = empty ∧
      right_subtree t = leaf 4
in
  t
endlet
```

Note that the constructor is not used directly; it is only known to be the constructor that *must* have been used. This makes the style suitable for those cases where the type of the constructor function is not known, either because the order of parameters was undefined or because an incomplete set was defined. It is also a suitable way of constructing an underspecified member of the type — possibly leaving one or more projected values unspecified.

This form of specification is of such utility that a short form is provided in HP-SL. For example

```
[node
     ▷ right_subtree ≙ leaf 4
     ▷ left_subtree ≙ empty
]
```

Note that the syntax closely follows that of the type declaration. The syntax is interpreted as follows — the initial use of the 'node' constructor implies that 'is_node' would be true if applied to the expression. Then, following each of the '▷' symbols is an binding between a projector on the left side and the value which it should project on the right side.

Any projector associated with the constructor, but which is not bound in the expression, is considered

underspecified; but tools are free to issue warnings if this occurs.

6 Pattern Matching

Under certain (syntactically determinable) conditions, HP-SL provides a mechanism for pattern matching over terms of the abstract types declared using the constructor/projector notation.

Basically the conditions are that no 'confusion' may occur in the patterns, thus if

- the **overlapping** keyword is used, or
- one of the constructors is defined using the form

  $[\![\ c : T\ ]\!]$

then pattern matching is not fully provided. In the first case no pattern matching is provided. In the scond case, pattern matching is provided for all other constructors (unless they are similarly defined).

For the other cases, the syntax follows that introduced for the types and repeated in the expressions. As an example consider the following

```
cases t of
  case [[empty]] then ...
  case [[leaf ▷ v]] then ...
  case [[node
          ▷ t1 ≙ left_subnode
          ▷ t2 ≙ right_subnode
          ▷ both_trees ]] then ...
endcases
```

The pattern for a constant constructor is just that constructor within the brackets.

With a function constructor, one of two situations occurs.

1. If the whole of the constructor's input value is to be matched, a pattern may be given on its own after a ' ▷ '. There are two examples above, the first being the 'leaf' case, the second being the matching of 'both_trees' to the pair of subtrees.
2. If a pattern is to be matched with just the result of a single projector, then that the pattern must be accompanied by ' ≙ *projector_id* '. The two examples of this are the pattern matching of 't_1' and 't_2' in the 'node' example.

Note that

- not all projectors need be mentioned.
- pattern matching is only allowed on types that have *not* been declared as 'overlapping'
- whole input matching is only allowed in cases where a constructor's domain type is known (equivalently, that the constructor function is defined).

7 References

[1] Patrick Goldsack. The HP-SL model of polymorphism — an informal description. Technical Report HPL-91-71, Hewlett-Packard Laboratories, Bristol, June 1991.

[2] Patrick Goldsack. The HP-SL type model. Technical Report HPL-91-73, Hewlett-Packard Laboratories, Bristol, June 1991.

[3] Patrick Goldsack. Module combinators for specification languages. Technical Report HPL-91-70, Hewlett-Packard Laboratories, Bristol, 1991. June.

A Semantics of Type Declarations

There are two issues to consider when giving a description of the semantics of the abstract type syntax

1. the names that are introduced, and their type
2. the axioms that are included by virtue of the way in which the syntax is used.

A.1 Declarations

Given

$$\textbf{type}\ Tid \triangleq \ldots \mid constructor_i \mid \ldots$$

From this we get the definition of the type

$$\textbf{type}\ Tid$$

and the definitions generated by each of the constructor descriptions.

For constructors, there are several possibilities

- constants

  $$[\ id\ ]$$

  gives

  $$\begin{array}{l}\textbf{val}\ id\ :\ Tid\\ \textbf{val}\ is\_id\ :\ Tid \rightarrow Bool\end{array}$$

- constructor function only

  $$[\ id\ :\ Texpr\ ]$$

  gives

  $$\begin{array}{l}\textbf{val}\ id\ :\ Texpr \rightarrow Tid\\ \textbf{val}\ is\_id\ :\ Tid \rightarrow Bool\end{array}$$

- single projector pattern

  $$[\ id \triangleright pattern\ ]$$

  gives

$$\textbf{val } id : typeof(pattern) \rightarrow Tid$$
$$\textbf{val } is\_id : Tid \rightarrow Bool$$

and for every $name \in namesof(pattern)$

$$\textbf{val } name : Tid \rightarrow typeof(name, pattern)$$
$$\textbf{fn } set\_name : typeof(name, pattern) \rightarrow Tid \rightarrow Tid \textbf{ is}$$
$$set\_name\ x\ t$$
$$\textbf{pre } predicateof\_name(x,t)$$

where the predicate *predicateof_name* is the check of any sub-typing restriction on *pattern* with x replacing the n part within the pattern.

- multiple projector patterns

  $$[\ id \triangleright pattern_1 \ldots \triangleright pattern_n\ ]$$

  gives

  $$\textbf{val } is\_id : Tid \rightarrow Bool$$

  and for each $i \in \{1\ ..\ n\}$, for every $name \in namesof(pattern_i)$

  $$\textbf{val } name : Tid \rightarrow typeof(name, pattern_i)$$
  $$\textbf{fn } set\_name : typeof(n, pattern_i) \rightarrow Tid \rightarrow Tid \textbf{ is}$$
  $$set\_name\ x\ t$$
  $$\textbf{pre } predicateof\_name(x,t)$$

- $\triangleright \ldots$

  The '...' may occur alone or with any number of patterns.

  It does not introduce any identifiers, but ensures that the definition of the constructor function does not appear.

A.2 Axioms

The axioms may be divided up into several classes

1. those relating the *is_* functions to the constructors
2. those relating the constructor and the projectors
3. those relating the *set_* functions to the projectors
4. those relating the complete set of *is_* functions
5. those relating the abstract type to the domain types of the constructors
6. induction

These axioms are not independant, some of them may be deduced from the others. However, by giving them explicitly in this way a better understanding of the semantics is obtained.

A.2.1 'is_' axioms

The axioms relating the *is_* functions and constructor functions (or, of course, constructor constants) clearly only exist if both the functions are declared by the abstract type definition. Thus only the constructor syntaxes

$$[\ c_1\ ]$$
$$[\ c_2\ :\ T_c\ ]$$
$$[\ c_3 \triangleright p\ ]$$

where p is any pattern and T_c any type actually generate the axioms.

For the first case we have

$$(\forall\ t\colon T\cdot\ is\_c_1\ t \Leftrightarrow t = c_1)$$

For the second case, the difference is that the *is_* function is true for all posible parameters to the constructor

$$(\forall\ t\colon T\cdot\ is\_c_2\ t \Leftrightarrow (\exists\ t_c\colon T_c\cdot\ c_2\ t_c = t))$$

In the third case, the difference is in obtaining the domain type of the constructor function.

$$(\forall\ t\colon T\cdot\ is\_c_3\ t \Leftrightarrow (\exists\ t_c\colon typeof(p)\cdot\ c_3\ t_c = t))$$

A.2.2 projector/constructor axioms

These axioms exist only when both the constructor and the projector functions exits. Thus only the case of a constructor defined by

$$[\ c \triangleright p\ ]$$

need be considered.

An axiom of the following kind is generated for each $n \in namesof(p)$

$$\begin{array}{l}(\forall\ v\colon typeof(p)\cdot\\ \quad n\ (c\ v) =\\ \quad\quad \textbf{let}\\ \quad\quad\quad \textbf{val}\ p \triangleq v\\ \quad\quad \textbf{in}\\ \quad\quad\quad n\end{array}$$

endlet)

This axiom relies on the fact that n exists in the pattern p, and so by matching the pattern against the parameter value of the constructor, the appropriate component part is obtained.

A.2.3 'set_' axioms

The *set_* axioms are defined whenever both projectors and set functions are defined. Thus the following constructor syntaxes need to be considered

$[\ c_1 \triangleright p\ ]$
$[\ c_2 \triangleright p_1 \triangleright p_2\ ]$ /* *and greater numbers of patterns* */
$[\ c_3 \triangleright p \triangleright \ldots\ ]$ /* *and with greater numbers of patterns* */

The axioms are complicated by three factors

1. the patterns may have anonymous components, using the pattern place-holder '_'
2. the projectors and setter functions have dependencies provided by the layering of patterns — thus two pattern identifiers, one on each side of an 'as', may be related in some way
3. not all the projectors are known when the constructor is provided in the third of the syntax forms given above

The first is handled by considering every _ as a unique identifier and allowing their use in the axioms.

The second is handled by the use of a semantic predicate *dependant* which, given two pattern variables and a set of patterns in which they may occur, is true when the two variables are dependant through layering in one or more of the patterns.

The third is impossible to take account of, so the axioms state the relationship between the setter functions and projectors *defined* in the constructor definition. They do not guarantee that these relationships exist with those projectors and setters yet to be defined. This make use of the $\triangleright\ldots$ notation rather dangerous.

The axioms state two things

1. that having set the value, the associated projector delivers that value
2. that all other independant projectors deliver the same values as they did before.

Thus for each $name \in names(p_1) \cup \ldots \cup names(p_n)$ we get the axiom

$(\forall\ m \in names(p_1) \cup \ldots \cup names(p_n)$
  sat $\neg$ $dependant(name, m, \{p_1, \ldots, p_n\})\cdot$
  $(\forall\ e{:}\ T$ sat $is\_c_i\ e\cdot$
    $(\forall\ x : typeof(name, \{p_1, \ldots, p_n\})\cdot$
      $m(set\_name\ x\ e) = m\ e$

```
      )
   )
)
```

covering the 'rest don't change' part, and

$$
\begin{array}{l}
(\forall\ e{:}\ T\ \textbf{sat}\ is_c\ e\cdot \\
\quad (\forall\ x\ :\ typeof(\ n\ \in\ \{p_1,\ \ldots,\ p_n\})\cdot \\
\qquad n(set_n\ x\ e)\ =\ x \\
\quad) \\
)
\end{array}
$$

to deal with the behaviour of the setter on the related projector.

All dependant projectors are indirectly catered for by the above two axioms.

A.2.4 relating 'is_' functions

There are two kinds of axioms related to two pieces of syntax in the 'union' part of the language.

The first of these is the no junk axiom. The following axiom is added

$$(\forall\ t{:}\ T\cdot\ is\_c_0\ t\ \vee\ is\_c_1\ t\ \vee\ \ldots\ \vee\ is\_c_n\ t)$$

assuming the set of constructors to be the set $\{c_1,\ \ldots,\ c_n\}$ and assuming that the |... syntax is not used.

The second of the axioms is the no confusion axiom which is included if the **overlapping** keyword is not used.

$$
\begin{array}{l}
(\forall\ t{:}\ T\cdot \\
\quad (is\_c_0\ t\ \ \Rightarrow\ \neg\ (\ is\_c_1\ t\ \vee\ \ldots\ \vee\ is\_c_n\ t)) \\
\quad \wedge \\
\quad (is\_c_1\ t\ \ \Rightarrow\ \neg\ (\ is\_c_0\ t\ \vee\ \ldots\ \vee\ is\_c_n\ t)) \\
\quad \wedge \\
\quad \ldots \\
)
\end{array}
$$

If the |... syntax is used, the no confusion axiom covers only those constructor functions defined.

A.2.5 relating constructor function domain types

For the constructor definitions which do not include ▷..., one can define relations between the type of the constructor and the abstract type.

For the constant constructor

$$[\ c_1\ ]$$

we have

$$(\forall\ t{:}\ T\ \textbf{sat}\ is\_c_1\ t\cdot\ t = c_1)$$

For the case when there are no projectors

$$[\ c_2\ :\ T_2\ ]$$

we have

$$\begin{array}{l}(\forall\ t{:}\ T\ \textbf{sat}\ is\_c_2\ t\cdot\\ \quad (\exists\ t_2{:}\ T_2\cdot\ c_2\ t_2 = t)\\ )\end{array}$$

For the cases where there are projectors

$$\begin{array}{l}[\ c_3 \triangleright p\ ]\\ [\ c_3 \triangleright p_1 \triangleright p_2\ ]\\ etc\end{array}$$

we have (in a form where we assume n patterns)

$$\begin{array}{l}(\exists\ f{:}\ typeof(p_1) \times \ldots \times typeof(p_n)\cdot\\ \quad (\forall\ t{:}\ T\ \textbf{sat}\ is\_c_3\ t\cdot\\ \qquad (\exists\ s{:}\ typeof(p_1) \times \ldots \times typeof(p_n)\cdot f\ s = t)\\ \quad)\\ \quad \wedge\\ \quad (\forall\ t_1\ \textbf{and}\ t_2\ :\ typeof(p_1) \times \ldots \times typeof(p_n)\cdot\\ \qquad f\ t_1 = f\ t_2 \Leftrightarrow t_1 = t_2\\ \quad)\\ )\end{array}$$

In the case when there is a single pattern, this axiom may be re-written with the f replaced by the constructor function c_3. In the case where there is more than one pattern, it in effect states the potential existance of a constructor function.

A.2.6 Induction

Structural induction is a vital proof technique for types described in the form above. However, for this to be valid we need two properties

1. that all constructor functions and constants are known

2. that the type is finitely generated, all values of the type are constructed from a finite number of applications of the constructor functions

The former can be characterised by a simple syntactic restriction, the second requires the addition of another axiom — the induction axiom.

Thus, when the type is defined using only one of the three forms of constructor definition which give a constructor function, and does not use |..., then the induction axiom is provided.

The valid forms of constructor definition are

$[\ c\ ]$
$[\ c\colon T\ ]$
$[\ c \rhd p\ ]$

Note that the lack of the form

$[\ c \rhd p_1 \rhd p_2\ ]$

gives a degree of control over whether the induction axiom is indeed provided.

The form of the axiom is

$$
\begin{array}{l}
(\forall\ p\colon T \to Bool\cdot \\
\quad (\\
\qquad /*\ for\ each\ constant\ constructor\ c_i\ */ \\
\qquad p\ c_i \\
\qquad \wedge \\
\qquad\ \\
\qquad /*\ for\ each\ constructor\ function\ d_i\colon V \to T\ where\ V\ does\ not\ use\ T\ */ \\
\qquad (\forall\ v\colon V\cdot\ p(d_i\ v)) \\
\qquad \wedge \\
\qquad\ \\
\qquad /*\ for\ each\ constructor\ function\ e_i\colon V \to T\ where\ V\ mentions\ T\ */ \\
\qquad /*\ (for\ simplicity\ in\ presentation\ here\ assume\ V = T)\ */ \\
\qquad (\forall\ v\colon V\cdot\ p(v) \Rightarrow p(e_i\ v)) \\
\quad) \\
\qquad\ \\
\qquad \Rightarrow \\
\qquad\ \\
\quad (\ \forall\ t\colon T\cdot\ p\ t\ ) \\
)
\end{array}
$$

B HP-SL Modules and Abstract Types

HP-SL modules are defined to be independant of the base language, in that it could be viewed via an operational semantics on the name space and on the text. Thus it has no effects on the properties of the underlying flat language.

It was pointed out in [3] that for the '+' module operator to work in merging definitions, a base language has to have a notion of multiple definition for each kind of entity.

The following subsections breifly examine the effect on the abstract types of HP-SL.

HP-SL allows multiple definition of all its entity kinds, so long as these are compatible. It is up to the tools to warn of such multiple definitions if appropriate.

Basically, multiple definition is valid if the declarations and axioms introduced by each of the type definitions do not contradict each other.

However, additional constraints are imposed to ensure that consistent naming of constructors and projectors is achieved.

A type definition is 'complete' if the symbol '...' does not occur in the sequence of constructors, and is 'partial' otherwise. A constructor is 'fully specified' if at most one '▷' occurs in the constructor definition and that this does not preceed a '...'.

Two definitions may only exist if they are compatible, i.e. if the following conditions hold

- two complete type definitions for the same type may only exist if they have the same set of constructors.
- if one of the constructors is not complete, and the other is, then the complete one must have a super-set of the constructors of the partial one
- if one of the two constructors is fully specified, and the other not, then the fully specified constructor must contain a super-set of the projectors in the not fully-specified definition.
- axioms implied by one of the definitions, but not the other, take precedence (i.e. the lack of such axioms does not imply denial)

Thus for example one could write

$$\textbf{type } (\!|T|\!)\ Set \triangleq [\![empty]\!] \mid [\![singleton\colon T]\!] \mid \ldots$$
$$\textbf{type } (\!|T|\!)\ Set \triangleq \textbf{overlapping } [\![empty]\!] \mid [\![singleton\colon T]\!] \mid [\![union\ :\ Set\ T \times Set\ T]\!]$$

implying that the complete set of constructors was 'empty', 'singleton' and 'union', that values constructed by 'empty' and 'singleton' are always different, but that values constructed by 'union' may be equal to either a singleton set or an empty set.