Injection Misdemeanors

T. B. Dinesh

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
email: dinesh@cwi.nl

Abstract. ASF+SDF allows injections in the presence of over-loading. The injections are generally perceived as a very useful feature of ASF+SDF. They have, however, troubled many – from implementors to specifiers. We elaborate on some of the problems (misunderstandings) encountered.

Key Words & Phrases: Injections, Sub-sorts, Overloading, Algebraic Specifications

Note: Supported by the Netherlands Organization for Scientific Research (NWO) under the *Generic Tools for Program Analysis and Optimization* project.

1 Introduction

Over-loading of function names in a many-sorted frame work is allowed in ASF+SDF, in the presence of injections. This is different from the case of, say OBJ [GWM+92], where over-loading is allowed in the presence of sub-sorts; in what is known as order-sorted algebraic specifications [GM89].

Over-loading, in the presence of injections, lead to confusing or non-intuitive specifications – since understanding the specification is dependent on understanding the parsing priorities and sometimes the implementation vagaries. Semantics of injections and over-loading should be resolved, independent of efficiency issues, before many implementations appear. ASF+SDF compiler currently behaves differently than the interpreter.

Injections have also confused users by their similarity to sub-sorts (say, as in order-sorted algebra). The combination of over-loading and injections bring about the similarity. For understanding the injections, one has to understand the exact parse of the specification – which is confusing in the presence of over-loading and priority declarations.

Thus, the very nice features of ASF+SDF, such as over-loading, priority declarations and injections are themselves the source of confusion as the need to understand their interactions emerge.

In this document we illustrate, using examples, some of the confusion. The examples and many comments in this document are due to many specifiers who have used the ASF+SDF Meta-environment for writing large specifications, and some from the implementors of the ASF+SDF system.

2 Overloading

A small specification that illustrates the changing parse of the equations, as the signature is extended, is presented here. The reductions surprised Dimitri Naidich, and forced him to think of injections differently than sub-sorts.

Proceedings of ASF+SDF95. A workshop on Generating Tools from Algebraic Specifications. May 11 & 12, 1995, CWI, Amsterdam, M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman & E. Visser (eds.) Technical Report P9504, Programming Research Group, University of Amsterdam

2.1 Overloaded

imports Layout$^{A.2}$
exports
  sorts A B C
  context-free syntax
    a $\rightarrow$ A {**constructor**}
    A $\rightarrow$ B {**constructor**}

    c $\rightarrow$ C {**constructor**}
    overloaded(A) $\rightarrow$ C
    overloaded(B) $\rightarrow$ C

    test(B) $\rightarrow$ C
hiddens
  variables
    $B \rightarrow$ B
equations

[1] $\quad \mathit{overloaded}(\mathrm{a}) \;=\; c$

[2] $\quad \dfrac{\mathit{overloaded}(B) = c}{\mathit{test}(B) = c}$

Using the above, the term "test(a)" does not reduce to "c" as some might expect. However, removing the line

$$\mathit{overloaded}(A) \rightarrow C$$

from the signature will reduce the term "test(a)" to "c".

This can only be understood by understanding the parsed specification.[1] In the first case, the equation [1] is parsed as:

$$\mathit{overloaded}_{[\mathrm{A} \rightarrow \mathrm{C}]}(a) = c_{\mathrm{C}}$$

and the condition part of equation [2] is parsed as

$$\mathit{overloaded}_{[\mathrm{B} \rightarrow \mathrm{C}]}(B) = c_{\mathrm{C}}$$

The $\mathit{overloaded}_{[\mathrm{A} \rightarrow \mathrm{C}]}$ and the $\mathit{overloaded}_{[\mathrm{B} \rightarrow \mathrm{C}]}$ are different functions, and thus the term "$\mathit{overloaded}_{[\mathrm{A} \rightarrow \mathrm{C}]}(a)$" does not match the term "$\mathit{overloaded}_{[\mathrm{B} \rightarrow \mathrm{C}]}(a)$".

Now the answer to the question *how could one add an equation that gives the desired behavior?* is: Add a conditional equation of the form

$$[3] \;\; \mathit{overloaded}(B) = c \textbf{ when } B = a$$

However, when the *overloaded*$(A) \rightarrow C$ is removed from the signature, the equation [1] is parsed as:

$$\mathit{overloaded}_{[\mathrm{B} \rightarrow \mathrm{C}]}(a) = c_{\mathrm{C}}$$

because of the injection $A \rightarrow B$.

[1] We use subscripts to indicate the parse information, and take liberty to abbreviate or not show this information if the context makes it clear.

The above effect can surprise a specifier, when one inadvertently adds a function to the signature, only to notice that the specification behaves different from then on! Thus, the parsing of equations and its behavior is non-monotonic with respect to signature extension.

Over-loading, itself, is taken to an extreme in ASF+SDF – multiple identical definitions are allowed, which certainly lead to ambiguous terms. Considering multiple identical function declarations as one and the same will be useful, but that would not be a general solution, due to the presence of injections. A general solution would take in to account the injections and the over-lapping function declarations to help the user through the specification process. Some of the potentially interesting solutions are not adopted, in ASF+SDF, in order to keep ASF+SDF simple and free of various imposed constraints[2].

2.2 "Type safety" warning

A small specification that illustrates the need for issuing "equation not type-safe" warning is presented here.

imports Layout$^{A.2}$ Booleans$^{A.1}$
exports
  sorts VALUE VENV X
  context-free syntax
    BOOL → VALUE
    VENV → VALUE

    "□" → VENV
    cheat VENV → VENV

    f(VENV) → X
  variables
    VENV → VENV
equations

$$\textit{cheat}\ \text{VENV} = \texttt{tt} \qquad \text{[Cheat]}$$

The term *cheat cheat* □ reduces to *tt*, however *f*(*cheat cheat* □) is already in normal form. This is confusing for many non-expert users of the ASF+SDF Meta-environment. Understanding this, again, requires one to understand the parsing of the equations and the act of lifting the terms whenever possible. The equation [Cheat] is parsed as VALUE = VALUE, i.e.,

$$\textit{cheat}_{\text{VENV[VALUE]}}\text{VENV}_{\text{VENV}} = \textit{tt}_{\text{BOOL[VALUE]}} \qquad \text{[Cheat]}$$

where $\textit{cheat}_{\text{VENV[VALUE]}}$ means that *cheat* was initially parsed as VENV but lifted to VALUE.

Note that the function *cheat* declared to be of sort VENV is parsed as a function of sort VALUE! This is because of the injection function from VENV to VALUE. This function basically tells the parser to parse it in the lifted domain, if it is not able to parse it in the declared domain. Since the right hand side (*tt*) is of sort BOOL and the left hand side is of sort VENV, the only common sort for the two is the sort VALUE.

This does not quite explain why *cheat cheat* □ reduces to *tt*. This happens because the term is parsed in the term editor as a term of sort VENV, i.e.,

$$\textit{cheat}_{\text{VENV}}\ \textit{cheat}_{\text{VENV}}\ \square_{\text{VENV}}$$

[2]Any restriction put on the specifier might become a problem for writing large specifications.

However, the top sort allowed in the term window is any sort (by default) and thus it could be sort VALUE. So there is a chance that the term, although of sort VENV, could have been intended to be of sort VALUE. Thus while looking for a matching equation, this term could be lifted to its "lift-able" sorts (allowed in the context in which it occurs). Thus the term is (effectively) transformed to $cheat_{\mathrm{VENV[VALUE]}}\ cheat_{\mathrm{VENV}}\ \Box_{\mathrm{VENV}}$. Now the left *cheat* matches the left hand side of the equation and the result (in one reduction step) is *tt*.

Another surprise is in store, if one attempts to fix the "problem" by adding to the signature:

cheat VENV $\rightarrow$ VALUE

This does help remove the "`[Cheat] warning: equation not type-safe`" warning. However, now there are two *cheat* functions. One should not forget that these are not related (two different functions with an overloaded name) – see also the previous section. The equation now parses as:

$cheat_{\mathrm{VALUE}}\mathrm{VENV}_{\mathrm{VENV}} = tt_{\mathrm{BOOL[VALUE]}}$ [Cheat]

The result of this being that the term *cheat cheat* $\Box$ is already in normal form (i.e., does not reduce to *tt*). Since the parse of this term is still the same as before and even the lifting does help match this term to the left hand side of the equation [Cheat] (with the new parse).

The lessons of this illustration for a user are:

- Never ignore the "not type-safe" warning – unless you comprehend and desire the (side) effect.
- Be very careful when you overload function names, in the presence of injections.

3 Dynamic check and sub-sorts

In ASF+SDF, when a term is statically type correct, its reduction is always a type-correct term, where as in OBJ the reduced term could be type incorrect (due to retracts). In practice, this is desirable since it guarantees that statically type correct programs will not lead to dynamic errors. The bad side of this is that some desirable terms cannot be built since one cannot guarantee it will not lead to a run time error.

This example illustrates use of injections, to achieve the effect of retracts (of OBJ) – for certain cases.

imports Layout$^{A.2}$
exports
  sorts NAT
  context-free syntax
    "0" $\rightarrow$ NAT
    s(NAT) $\rightarrow$ NAT
  sorts NAT1
  context-free syntax
    NAT $\rightarrow$ NAT1
    exception $\rightarrow$ NAT1
    s(NAT1) $\rightarrow$ NAT1
    p(NAT1) $\rightarrow$ NAT1
  variables
    X $\rightarrow$ NAT
    X1 $\rightarrow$ NAT1

priorities

{s(NAT1) $\rightarrow$ NAT1} < {s(NAT) $\rightarrow$ NAT}

equations

$$
\begin{array}{llcl}
[1] & p(s(X)) & = & X \\
[2] & p(0) & = & exception \\
[3] & s(exception) & = & exception \\
[4] & p(exception) & = & exception
\end{array}
$$

Now s(p(p(s(0)))) reduces to exception where as p(s(s(0))) reduces to s(0), and s(p(s(s(0)))) reduces to s(s(0)) – seems as desired.

However, the parse information is bound tightly with the terms. We can see this effect when we try to reduce p(s(s(p(s(s(0)))))). It reduces to p(s(s(s(0)))), where one might have expected s(s(0)). To get this effect one has to explicitly change the parse information in the term to be rewritten as desired, e.g., by adding the following equation:

$$[5] \quad s(X_1) \;=\; s(X) \quad \textbf{when} \;\; X = X_1$$

Note that, with the above priorities declaration, rule [5] is parsed as

$$s_{\mathrm{NAT1}}(X1_{\mathrm{NAT1}}) = s_{\mathrm{NAT}}(X_{\mathrm{NAT}}) \textbf{ when } X_{\mathrm{NAT}} = X1_{\mathrm{NAT1}}$$

Rule [5] is the effect of retracts in OBJ.

This view of changing the parse of a function makes sense when we are in the context of strong overloading [GD94]. There could be other interpretations, e.g., where one could merge all similar functions and think of an intersection domain – sometimes called weak overloading. See also [CGL95], for the use of extendable sum types for such a purpose.

Although the trick of using the right variables, priorities declaration and the conditional equations seem to provide the desired effect, there are inherent limitations. Some times it is impossible to express the desired parse of the equations using these techniques. Some priorities declaration could make it very hard to build a term that has a lower priority (in the declarations).

3.1 Writing an OBJ module in ASF+SDF

It seems that, in general, the need for retracts in OBJ can be accommodated in ASF+SDF by using conditional equations – with trivial conditions which force a lowering of sort by using a variable of an appropriate sort.

To illuminate the implications, we try to transliterate a gcd specification, given in an OBJ document [GWM$^+$92] into ASF+SDF. This tackles the use of retracts while defining the equations, but not the use of retracts in building terms (in term editors) – for this we would need a way to build conditional terms! For example `Let N = 6/3 in N!` would be another way to write the term `(6/3)!` – which is not type correct, assuming that `/` returns a Real and `!` is defined for Naturals.

This is a specification of (a part of) Nat given in an OBJ report.

imports Booleans$^{A.1}$

exports

sorts Nat NzNat Zero

context-free syntax

Zero $\rightarrow$ Nat

NzNat → Nat

"0" → Zero
s Nat → NzNat

Nat "+" Nat → Nat {**assoc**}
Nat ">" Nat → BOOL
diff(Nat, Nat) → Nat
gcd(NzNat, NzNat) → NzNat
"(" Nat ")" → Nat {**bracket**}

Note that *diff* and *gcd* are declared commutative in OBJ.

hiddens
variables
[*MN*] → Nat
Nz [′]∗ → NzNat

Here we define some short hand symbols; and some functions available for every sort, in OBJ.

exports
lexical syntax
[*1-9*] → NzNat
context-free syntax
Nat "≡" Nat → BOOL
if BOOL *then* NzNat *else* NzNat *fi* → NzNat
if BOOL *then* Nat *else* Nat *fi* → Nat

equations

[n2] $N + 0 \;=\; N$

[n2'] $0 + N \;=\; N$

[n3] $s\,N + s\,M \;=\; s\,s\,N + M$

[n7] $0 > M \;=\; \mathtt{ff}$

[n8] $Nz > 0 \;=\; \mathtt{tt}$

[n9] $s\,N > s\,M \;=\; N > M$

[n10] $\mathit{diff}(0,\, N) \;=\; N$

[n10'] $\mathit{diff}(N,\, 0) \;=\; N$

[n11] $\mathit{diff}(s\,N,\, s\,M) \;=\; \mathit{diff}(N,\, M)$

[n13]
$$\frac{Nz = \mathit{diff}(Nz',\, Nz'')}{\begin{array}{l} \mathit{gcd}(Nz',\, Nz'') = \mathit{if}\ Nz' \equiv Nz'' \\ \qquad\qquad \mathit{then}\ Nz' \\ \qquad\qquad \mathit{else\ if}\ Nz' > Nz'' \\ \qquad\qquad\quad \mathit{then}\ \mathit{gcd}(Nz,\, Nz'') \\ \qquad\qquad\quad \mathit{else}\ \mathit{gcd}(Nz',\, Nz)\ \mathit{fi\ fi} \end{array}}$$

In OBJ this equation is written as:

```
gcd(Nz',Nz'') =
     if Nz' == Nz''
     then Nz'
     else
       if Nz' > Nz''
       then gcd(diff(Nz',Nz''),Nz'')
       else gcd(Nz',diff(Nz',Nz''))
       fi
     fi
```

Note the use of the term *gcd(diff(Nz',Nz"),Nz")* above.

These specifications are almost similar except for the trivial condition used in equation [n13]. The effect of this (harmless looking) condition can be observed when we try to reduce $gcd(1,3)$ (first, without the equation [n13'] below). This reduction gives the result $gcd(s0,s0)$ as the normal form! On careful reading one can observe that the condition fails (prematurely – for our purpose) for the case when *diff* returns 0, which is not a NzNat.

Try gcd again by adding this equation:

[n13'] $gcd(Nz, Nz) \;=\; Nz$

The necessity of this equation indicates that we cannot, in a straight forward manner, transform the "retract" cases of OBJ to "trivial conditions" of ASF+SDF. A straight forward translation would require nested equations. We can, using some form of nested conditional equations, rewrite the equation as:

```
[nn13] gcd(Nz',Nz'') =
           if Nz' == Nz''
           then Nz'
           else
             if Nz' > Nz''
             then gcd(Nz,Nz'') {\bf when} Nz = diff(Nz',Nz'')
             else gcd(Nz',Nz ) {\bf when} Nz = diff(Nz',Nz'')
             fi
           fi
```

Another way to transform could be to use hidden (auxilliary) functions.

Expand short hands:

[s1] $1 \;=\; s\ 0$

[s2] $2 \;=\; s\ s\ 0$

[s3] $3 \;=\; s\ s\ s\ 0$

[s4] $4 \;=\; s\ s\ s\ s\ 0$

[s5] $5 \;=\; s\ s\ s\ s\ s\ 0$

[s6] $6 \;=\; s\ s\ s\ s\ s\ s\ 0$

[s7] $7 \;=\; s\ s\ s\ s\ s\ s\ s\ 0$

[s8] $8 \;=\; s\ s\ s\ s\ s\ s\ s\ s\ 0$

[s9] $9 \;=\; s\ s\ s\ s\ s\ s\ s\ s\ s\ 0$

Some auxiliary functions (avaliable as predefined in OBJ):

[N1] $\textit{if}\ \texttt{tt}\ \textit{then}\ N\ \textit{else}\ M\ \textit{fi}\ = N$

[N2] $\textit{if}\ \texttt{ff}\ \textit{then}\ N\ \textit{else}\ M\ \textit{fi}\ = M$

[N3] $\textit{if}\ \texttt{tt}\ \textit{then}\ Nz\ \textit{else}\ Nz'\ \textit{fi}\ = Nz$

[N4] $\textit{if}\ \texttt{ff}\ \textit{then}\ Nz\ \textit{else}\ Nz'\ \textit{fi}\ = Nz'$

[N5] $N \equiv N\ =\ \texttt{tt}$

[N6] $N \equiv M\ =\ \texttt{ff}$ **when** $N \neq M$

4 Injections and implementations

In the interpretor implementation (using Le-lisp), injections are not preserved in the parse tree and thus nodes are collapsed. Dynamic checks are performed to ensure proper reductions. In the compiler implementation, injection functions are preserved. This leads to a different operational semantics than the Le-lisp implementation. When a term is injected into two different sorts which are injected in another sort, the compiler keeps the injection history in the term, where as the interpreter does not.

4.1 Injection Diamond

imports Layout$^{A.2}$ Booleans$^{A.1}$
exports
  sorts A B C D
  context-free syntax
    a → A
    A → B
    B → D

    A → C
    C → D
    eq(D, D) → BOOL
  variables
    D → D
equations

[1] $eq(D, D)\ =\ \texttt{tt}$

The above equation will always reduce eq(a,a) to tt (with the interpreter), regardless as to whether the term is parsed as

$$eq(a_{\mathrm{B}}, a_{\mathrm{C}})$$

or

$$eq(a_{\mathrm{C}}, a_{\mathrm{B}})$$

However, the compiler treats the two “a”s as different! Thus the current compiler behaves differently than the interpreter. Including such a test would complicate the compiler. A dynamic test would also make the compiler less efficient. However, as Jasper explained, this need not be a dynamic test. The compiler can generate the appropriate equations necessary to keep all the information in the equations themselves. For example, this above equation could generate (or represent) several more specific equations. We use $X_{[[\mathrm{A}]\mathrm{B}]\mathrm{D}}$ to indicate that X matches a term that was initially parsed as A, and then lifted to D via B (by injections). The equations would be of the form:

$$
\begin{array}{lll}
[c1] & eq(X_{[[\mathrm{A}]\mathrm{B}]\mathrm{D}}, X_{[[\mathrm{A}]\mathrm{B}]\mathrm{D}}) & = \mathrm{tt} \\
[c2] & eq(X_{[[\mathrm{A}]\mathrm{B}]\mathrm{D}}, X_{[[\mathrm{A}]\mathrm{C}]\mathrm{D}}) & = \mathrm{tt} \\
[c3] & eq(X_{[[\mathrm{A}]\mathrm{C}]\mathrm{D}}, X_{[[\mathrm{A}]\mathrm{B}]\mathrm{D}}) & = \mathrm{tt} \\
 & \vdots & \\
[ci] & eq(X_{[\mathrm{C}]\mathrm{D}}, X_{[[\mathrm{A}]\mathrm{B}]\mathrm{D}}) & = \mathrm{tt} \\
 & \vdots & \\
[cj] & eq(X_{[\mathrm{C}]\mathrm{D}}, X_{[\mathrm{B}]\mathrm{D}}) & = \mathrm{tt} \\
 & \vdots & \\
[ck] & eq(X_{\mathrm{D}}, X_{\mathrm{D}}) & = \mathrm{tt}
\end{array}
$$

The diamond structure is quite common in specifications and thus this is a serious difference. For example, John Field has used this property to define an operator class structure for his PIM specifications (see Section A.3).

Not only has the forgetting of the injection functions in the parse trees helped the interpreter achieve the nice behavior in the previous section, but also has helped the GSE (the generic syntax editor) achieve the behavior it currently has – in a simple manner. However, it has troubled various other implementation efforts such as the implementation of the incremental rewriting, compiler, pretty printers and even the origin tracking!

4.2 GSE behavior

Currently, GSE has *one* pointer in its abstract syntax tree (AST) pointing at the focus subtree. When injections are left as separate nodes in the AST, GSE will have to maintain *two* pointers. The semantics of a mouse-click on a text part means "change focus to the smallest subtree S of which the corresponding text includes the text part". Furthermore, the focus node yields the focus phylum (sort). That is, GSE knows what kind of language constructs may be inserted at this point.

With injections, the mouse-click yields a node far away from the root, while the language constructs that may be inserted need to "look through injections", i.e., move towards the root. This can be done, but it's going to be harder, e.g. due to SDF's "bracket rules". It has consequences for SEAL (what is the sort of the focus? – do you mean the one pointer or the other?). Also, the meaning of a "move the focus to the parent node" becomes unclear.

The "two pointer approach" will have advantages as well. Currently the "expand" menu gives the information that the thing in focus is an EXP while in fact it is a NAT when using:

$$\mathrm{NAT} \rightarrow \mathrm{INT} \rightarrow \mathrm{REAL} \rightarrow \mathrm{EXP}$$

The "two pointer approach" may supply the user with this information: *You are at a NAT, but you may insert an EXP here.*

4.3 Priorities

Declaration of the proper priorities is crucial to achieve the desired semantics with injections (see Section 3). Using a general heuristic that helps solve many priority conflicts [Kli88, HHKR92, Kli93, KV94] is only desirable if an user can override the defaults, otherwise we defeat the rule of being "non-imposing on the user".

The current implementation of default priorities prefers the shorter injection chain. However, there is no way to specify priorities for injection functions. For example in Section 4.1, one would need to express which chain is preferred while parsing the term "eq(a,a)".

5 Slice problem

Frank Tip encountered a problem with injections during slicing, associated with the forgetfulness of the injection functions.

Consider the following signature:

imports Layout$^{A.2}$ Booleans$^{A.1}$
exports
  sorts INT STRING EXP
  lexical syntax
    [A-Za-z]+ → STRING
    [0-9]+ → INT
  context-free syntax
    INT → EXP
    STRING → EXP
    is-string(EXP) → BOOL
  variables
    INT → INT
    STRING → STRING
equations

[1] *is-string*(INT) = `ff`
[2] *is-string*(STRING) = `tt`

Consider the reduction of a term "is-string(3)". It reduces to "ff".

There are two cases, w.r.t. the injection functions

1. *No explicit functions:* Dependence tracking would result in tracking "ff" to "is-string". Slicing with respect to "ff" would lead to a hole of the form is-string($< ? >$). This implies that $< ? >$ can be replaced by either an INT or a STRING! This is wrong since $< ? >$ should not be replaced by a STRING.

2. *With explicit functions:* Dependence tracking would result in tracking "ff" to both "is-string" and the injection function that injects INT into EXP. Lets call this injection function "ItoE". Then the hole would be of the form is-string(ItoE($< ? >$)). Thus, implying that $< ? >$ can only be replaced by an INT. Clearly this is the desired effect.

6 Sub-sorts or Injections?

Do we need sub-sorts then? Do we need to sacrifice the "be non-imposing" rule to impose sub-sorts? Are sub-sorts "imposing" to an typical ASF+SDF user? What exactly do we mean when we say sub-sorts? For example [Mos92, GD94, Ste95] discuss many different interpretations for sub-sorts.

Peter Mosses prefers trees not to contain injection functions at all. He suggests that A → B be considered as sub-sort inclusion, rather than an invisible injection function. Thus, a branch of sort A could occur wherever one of sort B is allowed. He claims that one can then safely disregard ambiguities that yield the same tree, due to different sub-sort inclusions/injections. In particular, he seems to be quite annoyed by the disambiguator asking him to choose between identical looking things[3]. He also claims that preferring minimum chains of injections can give nasty surprises which are difficult to circumvent. This suggestion, however, takes the situation close to his unified algebras, and thus provide obvious advantages for building the Action Semantics Definition tools.

The above suggestion can be regarded as the other extreme of injections. Provided an user can obtain the above interpretation, by appropriately defining how the injections should behave, current notion of injections appears to be more general. A similar argument would view having invisible injection functions as a compromise to flexibility, when compared to OBJ's more flexible (and visible if necessary) retraction functions.

Arie van Deursen notes in his thesis [vD94, Section 5.7] that the SDF reference manual [HHKR89, Section 5.4] suggests that injection functions be treated as sub-sorts. The plus-lists and star-lists provided in SDF behaves both in the interpreter version and the compiler version as sub-sorts – i.e., there are no injection functions and lifting of sorts that one has to worry about (from plus-lists to star-lists). If there was a way to define (represent) an empty-list, then there would be cases where star-lists are defined using the plus-list and the empty-list.

What are the advantages of injections over that of sub-sorts? Using injections it is possible to have two constants to be considered different as, say Intcons, but same as Expressions – when Intcons are injected into Expressions. This is a more general notion than sub-sorts, in a way!

We hope that after the discussions during the workshop, we can conclude on what the ASF+SDF users need.

In any case, we need to forget the different ways injections have been implemented and consider a discussion of what we need from injections. It appears like the first semantics (the Le-Lisp implementation) came about since there were essentially two basic tools (the equation manager and the GSE) that had to work with injections; and both these tools found it easier not to have explicit injections in the abstract syntax trees. It was not until much later, during the implementation of the incremental rewriting (Emma van der Meulen) and the compiler (Jasper Kamperman and Pum Walters), that the missing injections were considered a handicap. Thus, it is essential that the abstract syntax tree keeps the injection information. However, we need to settle the desired semantics.

7 Conclusions

Injections, with strong over-loading appear to have a lot to desire. In combination with default and declared priorities, these become harder to understand – as it forces one to have precise parse information to follow the specification; and sometimes hard to manipulate to get the desired results. The invisibility of the injection functions is a big advantage for the specifier most of the time, but it compromises flexibility by making it possible for the injection to escape simple and straight forward manipulations.

Acknowledgements I would like to thank Dimitri Naidich, John Field, Peter Mosses, Wilco Koorn and Frank Tip for contribution of examples and experiences. The various discussions with members of the ASF+SDF group has made it possible

[3]He suggests that in general SDF should allow repetition of sorts and syntax rules.

to understand the vagaries of injections. Any error in this document is of course attributed to my yet-to-understand features of the injection semantics in the current ASF+SDF Meta-environment. Finally, the reviewer comments have been of good help in preparing this document.

References

[CGL95] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995. A preliminary version has been presented at the *1992 ACM Conference on LISP and Functional Programming*, San Francisco, June 1992.

[GD94] Joseph Goguen and Răzvan Diaconescu. An oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4(3):363–392, September 1994.

[GM89] J.A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions, and partial operations. SRI-CSL-89-10, SRI international, Menlo Park, 1989.

[GWM+92] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.P. Jouannaud. Introducing OBJ. In J.A. Goguen, D. Coleman, and R. Gallimore, editors, *Applications of Algebraic Specification Using OBJ.* Cambridge University Press, 1992. To Appear.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[HHKR92] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. *The syntax definition formalism SDF - reference manual,* version 6 December, 1992. Earlier version in *SIGPLAN Notices*, 24(11):43-75, 1989. Available by *ftp* from ftp.cwi.nl:/pub/gipe/reports as SDFmanual.ps.Z.

[Kli88] Paul Klint. Definitie van prioriteiten in sdf. Unpublished document, July 1988.

[Kli93] P. Klint, editor. *The ASF+SDF Meta-environment User's Guide,* version 26 February. Centrum voor Wiskunde en Informatica (CWI), 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe/reports as SysManual.ps.Z.

[KV94] Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy, October 1994. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano. Also as TR P9426, Programming Research Group, University of Amsterdam, ftp://ftp.fwi.uva.nl /pub/programming-research/ reports/1994/ P9426.ps.Z.

[Mos92] Peter Mosses. The use of sorts in algebraic specifications. In *Proceedings of the Eighth Workshop on Specification of Abstract Data Types, Dourdan, France, 26-30 August 1991*, Lecture Notes in Computer Science. Springer-Verlag, 1992. Also available as Arhus University technical report: DAIMI PB -416; September 1992.

[Ste95] John Stell. Order-sortde theories. email: john@cs.keele.ac.uk, April 1995.

[vD94] Arie van Deursen. *Executable Language Definitions.* PhD thesis, Unversity of Amsterdam, 1994.

A Some Auxiliary modules

A.1 Module Booleans

imports Layout$^{A.2}$
exports
  sorts BOOL-CON
  lexical syntax
    $tt \rightarrow$ BOOL-CON
    $f\!f \rightarrow$ BOOL-CON
  variables
    Bool-con [*1-9*]$* \rightarrow$ BOOL-CON
exports
  sorts BOOL
  context-free syntax

BOOL-CON	$\rightarrow$ BOOL	
BOOL "|" BOOL	$\rightarrow$ BOOL	{**assoc**}
BOOL "&" BOOL	$\rightarrow$ BOOL	{**assoc**}
"*not*" BOOL	$\rightarrow$ BOOL	
"(" BOOL ")"	$\rightarrow$ BOOL	{**bracket**}
BOOL "≡" BOOL	$\rightarrow$ BOOL	
if BOOL *then* BOOL else BOOL fi	$\rightarrow$ BOOL	

  variables
    Bool [*1-9'*]$* \rightarrow$ BOOL
  priorities
    BOOL "|"BOOL → BOOL < BOOL "&"BOOL → BOOL <
    "*not*"BOOL → BOOL
equations

[1] $\texttt{tt} \mid Bool = \texttt{tt}$

[2] $\texttt{ff} \mid Bool = Bool$

[3] $\texttt{tt}\ \&\ Bool = Bool$

[4] $\texttt{ff}\ \&\ Bool = \texttt{ff}$

[5] $not\,\texttt{tt} = \texttt{ff}$

[6] $not\,\texttt{ff} = \texttt{tt}$

[7] $Bool \equiv Bool = \texttt{tt}$

[8] $Bool_1 \equiv Bool_2 = \texttt{ff}$ **when** $Bool_1 \neq Bool_2$

[9] $if\,\texttt{tt}\ then\ Bool_1\ else\ Bool_2\ fi = Bool_1$

[10] $if\,\texttt{ff}\ then\ Bool_1\ else\ Bool_2\ fi = Bool_2$

A.2 Module Layout

exports
lexical syntax

```
"%%"~[\n]*[\n] → LAYOUT
[␣\t\n]        → LAYOUT
```

A.3 Operator Class Structure

John Field found injections to be convenient for organising the specification by defining a "lattice of types". This is an illustration of how he defined simple classes of algebraic substructures, that utilizes the fact that equations may apply to terms of any sort below a given point in the lattice – by using variables whose sort corresponds to a given point.

imports Layout$^{A.2}$
exports
sorts AOp ACOp BOp PredOp Op AnyOp

AOp is a monoid – associative binary operations with identity element, ACOp is a commutative monoid – where the binary operation is commutative, BOp is the Boolean conjuction and disjunction operations, PredOp has the operator that computes a Boolean value, Op has the n-ary operators on base values, and AnyOp – all operator types, with the addition of a special "dummy".

Operator classes and their relationships are (note the injection diamond):

context-free syntax

```
PredOp      → Op
AOp         → Op
ACOp        → AOp
BOp         → ACOp
BOp         → PredOp
Op          → AnyOp
"DummyOp"   → AnyOp
```

In the following, IdElt (monoid id element) and AnnElt (annihilator) are algebraic meta-operators, AnyOp $< \cdots >$ is any operator followed by its arguments and L is the list constructor binary operator (L<hd,tl>).

context-free syntax

```
"IdElt(" AOp ")"            → Val
"AnnElt(" AOp ")"           → Val

AnyOp "<" {Val ","}* ">"    → Val
"~"                         → PredOp
"&"                         → BOp
"|"                         → BOp
"BOpComp(" BOp ")"          → BOp

"≡"                         → PredOp
"BoolVal?"                  → PredOp
"ErrVal?"                   → PredOp
"L"                         → Op

"Hd"                        → Op
"Tl"                        → Op
"App"                       → AOp
"Null?"                     → PredOp
"+"                         → ACOp
"−−"                        → Op
```

"−" → Op
"∗" → ACOp
"/" → Op
"*LTInt*" → PredOp

sorts Val BoolVal

context-free syntax

"*T*" → BoolVal
"*F*" → BoolVal
BoolVal → Val

variables

"_" *AOp* → AOp
"_" *ACOp* → ACOp
"_" *BOp* → BOp
"_" *PredOp* → PredOp
"_" *Op* → Op
"_" *AnyOp* → AnyOp
"_" *BoolVal* [*0-9*]∗ → BoolVal
"_" *Val* [*0-9*]∗ → Val
"_" *ValList* [*0-9*]∗"∗" → {Val ","}∗
"_" *ValList* [*0-9*]∗"+" → {Val ","}+

equations

Generic rules for monoids. We use conventions that Aop<_Val> is equivalent to _Val. Empty list will represent identity element for the operator

$$\begin{aligned}&\_AOp < \_ValList_1^*, \_AOp < \_ValList_2^+ >, \_ValList_3^* > = \\ &\_AOp < \_ValList_1^*, \_ValList_2^+, \_ValList_3^* >\end{aligned} \qquad \text{[OpFlatten1]}$$

$$\_AOp < \_Val > \; = \; \_Val \qquad \text{[OpFlatten2]}$$

$$\_AOp < \; > \; = \; IdElt(\;\_AOp) \qquad \text{[OpIdElt]}$$

$$\frac{\_Val = IdElt(\;\_AOp)}{\begin{aligned}&\_AOp < \_ValList_1^*, \_Val, \_ValList_2^* > = \\ &\_AOp < \_ValList_1^*, \_ValList_2^* >\end{aligned}} \qquad \text{[OpIdElim]}$$

$$\frac{\_Val = AnnElt(\;\_AOp)}{\_AOp < \_ValList_1^*, \_Val, \_ValList_2^* > = \_Val} \qquad \text{[OpAnn]}$$

Generic equality rules for associative operators

$$\frac{\begin{aligned}&\equiv < \_Val_1, \_Val_2 > = T, \\ &\equiv < \_AOp < \_ValList_1^* >, \_AOp < \_ValList_2^* > > = T\end{aligned}}{\begin{aligned}\equiv < &\_AOp < \_Val_1, \_ValList_1^* >, \\ &\_AOp < \_Val_2, \_ValList_2^* > > = T\end{aligned}} \qquad \text{[AOpEqT1]}$$

Generic equality rules for AC operators

$$\frac{\begin{aligned}&\equiv < \_Val_1, \_Val_2 > = T, \\ &\equiv < \_ACOp < \_ValList_1^*, \_ValList_2^* >, \\ &\qquad \_ACOp < \_ValList_3^*, \_ValList_4^* > > = T\end{aligned}}{\begin{aligned}\equiv < &\_ACOp < \_ValList_1^*, \_Val_1, \_ValList_2^* >, \\ &\_ACOp < \_ValList_3^*, \_Val_2, \_ValList_4^* > > = T\end{aligned}} \qquad \text{[ACOpEqT1]}$$

Equations for Booleans

$$
\begin{array}{lll}
BOpComp(\ \&) & = \ | & \text{[BOpCompConj]} \\
BOpComp(\ |) & = \ \& & \text{[BOpCompDisj]} \\
IdElt(\ \&) & = \ T & \text{[BConjId]} \\
IdElt(\ |) & = \ F & \text{[BDisjId]} \\
AnnElt(\ \&) & = \ F & \text{[BConjAnn]} \\
AnnElt(\ |) & = \ T & \text{[BDisjAnn]} \\
\sim < T > & = \ F & \text{[BTrueNeg]} \\
\sim < F > & = \ T & \text{[BFalseNeg]} \\
\sim < \sim < \_BoolVal > > & = \ \_BoolVal & \text{[BNegNeg]}
\end{array}
$$

$$
\begin{array}{ll}
\sim < \_BOp < \_BoolVal, \_ValList^{+} > > = & \text{[BDeMorgan]} \\
\quad BOpComp(\ \_BOp) < \sim < \_BoolVal >, \sim < \_BOp < \_ValList^{+} > > > &
\end{array}
$$

Note that many Boolean laws are instances of generic laws for algebraic structures above (in particular the [OpAnn] and [OpIdElim]). Also note that (some of) the algebraic laws above apply equally well to lists, integer operators, and Booleans, and that they can be expressed succinctly without repeated instances for each subclass.