CS-Note[*] 19

Bill J. Ellis
September 11, 2002

Raising NuSPADE Issues (Volume 1)

Contents

[*] Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the NuSPADE project (EPSRC GR/R24081).

1 Introduction

NuSPADE is a tight integration between the existing systems SPADE and CLAM. There are a number of issues that are raised from considering the exact nature of this integration. While the importance of these issues vary, each has the potential to impact on the implementation and behaviour of NuSPADE. This note attempts to raise some of these issues and provide reasonable proposals for their solution, with some slight details on implementation.

For the first prototype only the necessary changes will be made to CLAM for integration with SPADE. Rather than try classify and present only those issues deemed necessary, a full collection of issues will be raised. In the conclusion these will be weighted for importance.

Note that in this document the term PLANNER refers to the planner as part of NuSPADE. A more suitable name for this has yet to be determined[1].

2 Expressing Terms in PLANNER

It is necessary for the PLANNER to express terms in a meta-language. The choice of this representation and its subsequent implementation will be a significant task in building the PLANNER.

2.1 CLAM Terms

Though CLAM is built on top of OYSTER the term structure of CLAM is slightly different. CLAM employs a quantified and equality typed first order term structure. For example, an induction hypothesis in CLAM is:

```
v0:ih(b:pnat=>c:pnat=>times(a,times(b,c))=times(times(a,b),c) in pnat)
```

To support rippling [2] the planning term structure is augmented with additional annotations. In CLAM, and other theorem provers that support rippling, these annotations are directly added into the term structure. The induction conclusion for the above takes the following form, Where `\.../` (sinks), ` ``...'' ` (wave fronts) and `{}` (wave holes) are the added annotation:

```
b:pnat=>c:pnat=>
      ``plus({times(a,times(\b/,\c/))},times(\b/,\c/))''<out> =
                     times(``plus({times(a,\b/)},\b/)''<out>,\c/) in pnat
```

2.2 SPADE Terms

SPADE terms exist inside an extended Prolog environment. Inside this environment the syntax of terms can be separated from their semantics. Each of these will be considered separately.

[1] Suggestions already include: NU, BUCKET and LOBSTER. Whatever the choice, it should not be a verb. Otherwise it will need to be preceded by "the", as in "*the* SIMPLIFIER does this" compared to "CLAM does this".

2.2.1 Syntax

The syntax gap between SPADE terms and their existence in Prolog is shown in Figure 1. Each section is summarised below:

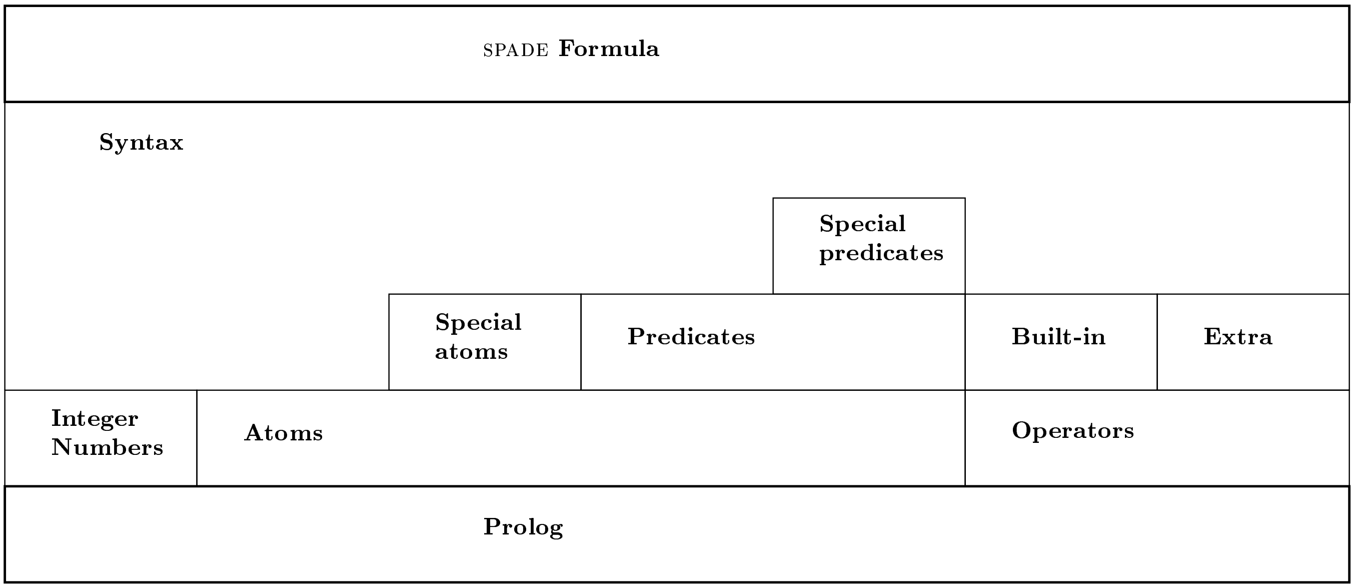

Figure 1: SPADE term syntax on Prolog

- **Integer Numbers**
  SPADE terms make direct use of Prologs built-in integer number support. Real numbers are not used in SPADE. This restriction is overcome by the EXAMINER converting all real numbers into rationales (fractions) before entering SPADE. For example:

```
1
1/2
-1234
1000000000000000000000000000000000000
```

- **Atoms**
  Prolog atoms are used to describe the one word identifiers of a SPADE term. These are the variables or type specifiers of the term. For example:

```
l + 1 <= loop__1__i
u <= max_table_size
not (element(arr, [loop__1__i + 1]) < element(arr, [k]))
for_all (i___1: base_index_type, ((i___1 >= index_type__first) and (i___1 <= index_type__last))
-> ((element( arr, [i___1]) >= contents_type__first) and (element( arr, [i___1]) <=
contents_type__last)))
```

- **Special atoms**
  There are Prolog atoms that do not gain their meaning as regular identifiers of the term. There are special atoms with their own predefined meanings. The following atoms known to have a special meaning in SPADE:

Special atom	Example use in expression	
`true`	`true or true`	This atom is separate to the Prolog atom of the same name.
`false`	`false or false`	
`integer`	`for_all(i__1: integer, ... )`	
`boolean`	`for_all(i__1: boolean, ... )`	
`real`	`for_all(i__1: real, ... )`	

- **Predicates**
  Prolog predicates are used to describe more complex identifiers of a SPADE term. These are the proof functions of a term. For example:

```
perm(table, table)
partitioned(table, 1, loop__1__low, max_table_size)
the_smallest(arr, l, loop__1__i + 1)
```

- **Special predicates**
  There are Prolog predicates that do not gain their meaning through user provided proof functions. There are special predicates with their own predefined meanings. The following predicates known to have a special meaning in SPADE:

Special predicate	Example use in expression
update	update(array,[i],value)
element	element(array,[i])
succ	succ(enum)
pred	pred(enum)
first	first(seq)
last	last(seq)
nonfirst	nonfirst(seq)
nonlast	nonlast(seq)
abs	abs(value)
sqr	sqr(value)
odd	odd(value)
subset_of	subset_of(set)
strict_subset_of	strict_subset_of(set)

There are further special predicates used inside the condition part of a rule. The predicates known to be in this category are:

Special predicate	Example use in expression
not	not(special_predicate_in)
checktype	checktype(expression_in, type_identifier_in)
type	type(identifier_name_in, fdl_type_in)
enumeration	enumeration(identifier_name_in, fdl_enumeration_type_in)
last	last(list_in, element_out)
in_order	in_order(element1_in, element2_in, list_in)
in	?[2]
subset	subset(set_in,set_in)
subst_vbl	subst_vbl(item_find_in,item_replace_in,expression_in,expression_out)
sublist	sublist(list1, list2)
append	append(list1, list2, list3)
set_union	set_union(set_in, set_in, set_out)
set_intersect	set_intersect(set_in, set_in, set_out)
set_lacking	set_lacking(set_in, set_in, set_out)
simplify	simplify(expression_in, expression_out)
integer	integer(expression)
nonvar	nonvar(expression)
var	var(expression)
atom	atom(expression)
atomic	atomic(expression)
length	length(list,length)
iss	iss(expression,value)
genvar	genvar(atom, variable)

- **Operators**
  Prolog operators provide a mechanism for adding new syntax to the Prolog environment. The new syntax is added specifying its fixity and precedence. This is significant as the extra information is required to correctly interpret a SPADE term.

[2]I could not trace this predicate. See also the semantic description of these special predicates.

- **Built-in**
  Prolog has a number of built-in operators. Most of these are related to Prologs number support. The following built-in operators are known to be used in SPADE:

Operator declaration	Example use in expression
`op(21, fx, -)`	`-A`
`op(31, yfx, -)`	`A - B`
`op(31, yfx, +)`	`A + B`
`op(21, yfx, *)`	`A * B`
`op(21, yfx, div)`	`A div B`
`op(21, yfx, /)`	`A / B`
`op(21, yfx, mod)`	`A mod B`
`op(40, xfx, <)`	`A < B`
`op(40, xfx, >)`	`A > B`
`op(40, xfx, >=)`	`A >= B`
`op(40, xfx, =)`	`A = B`

- **Extra**
  Where a suitable Prolog operator does not exist a new operator can be introduced. This is used to add the following extra operators in SPADE:

Operator declaration	Example use in expression	
`op(200,fy,not)`	`not A`	
`op(20,xfx,**)`	`A ** B`	
`op(220,yfx,or)`	`A or B`	
`op(220,yfx,xor)`	`A xor B`	
`op(210,yfx,and)`	`A and B`	
`op(235,yfx,<->)`	`A <-> B`	
`op(230,xfy,->)`	`A -> B`	
`op(40,yfx,<=)`	`A <= B`	Prolog supports less-than as `=<`.
`op(40,yfx,<>)`	`A <> B`	
`op(100,yfx,in)`	`A in B`	
`op(100,yfx,not_in)`	`A not_in B`	
`op(40,fy,set)`	`set A`	
`op(31,yfx,\/)`	`A \/ B`	
`op(21,yfx,)`	`A /\ B`	
`op(31,yfx,\)`	`A \ B`	
`op(35,yfx,@)`	`@`	
`op(20,xf,)̃`	`~`	

2.2.2 Semantics

The semantic gap between SPADE terms and their existence in Prolog is shown in Figure 2. Each section is summarised below:

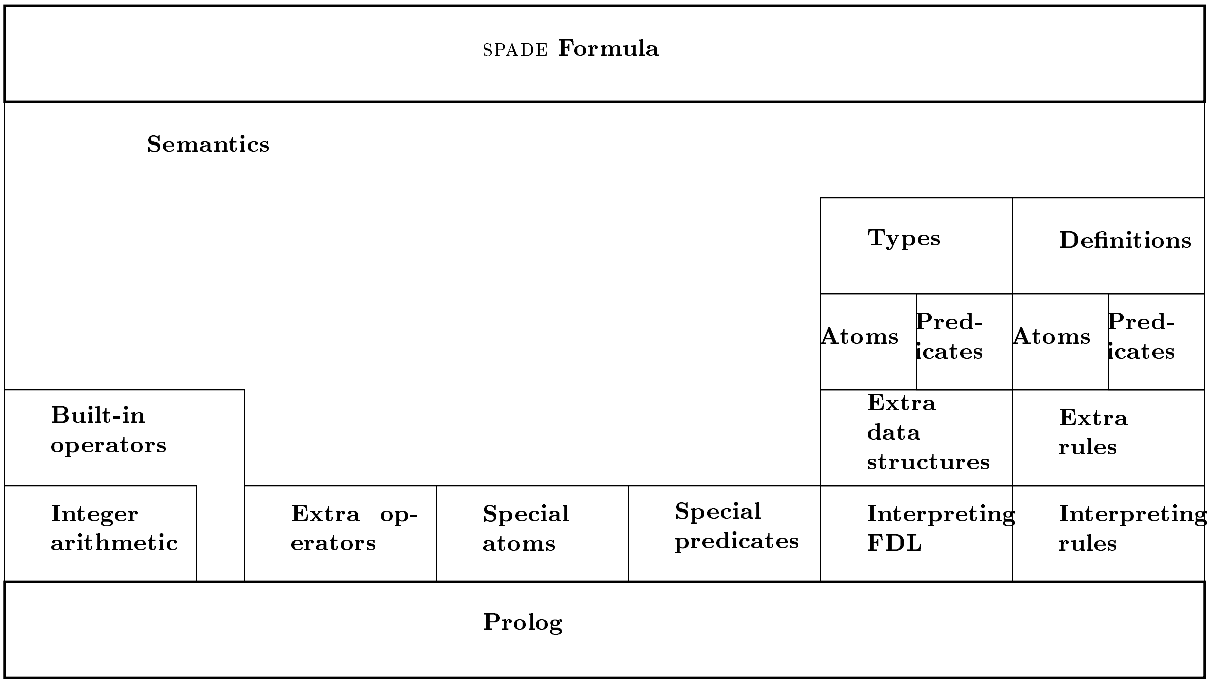

Figure 2: SPADE term semantics on Prolog

- **Integer arithmetic**
  Prolog has built-in support for numerical arithmetic, in particular integer arithmetic. In a few varieties of Prolog (Including Poplog-Prolog and Sicstus) the range of integer numbers is sufficiently large to be considered infinite for all practical purposes. This strong support for integer arithmetic is relied upon in providing the semantics of integer arithmetic operators in SPADE terms.

- **Built-in operators**
  The built-in operators have their semantics provided through a layer above the Prolog. However, where dealing with integers, many of these are implemented almost directly in terms of Prologs integer arithmetic. The following gives an overview of the semantics for each of the built-in operators used in SPADE:

`op(21, fx, -)`	Unary negation.
`op(31, yfx, -)`	Subtraction.
`op(31, yfx, +)`	Addition.
`op(21, yfx, *)`	Multiplication.
`op(21, yfx, div)`	Integer division. (`A div B` is the maximum whole number of `B` in `A`).
`op(21, yfx, /)`	Real division. Note that to avoid real numbers in SPADE this operation can only be evaluated where it is known that the result is an integer.
`op(21, yfx, mod)`	Modulas. Note that, as default, no semantic interpretation is given for modulas. Different programming languages have adopted different definitions, thus SPADE leaves this to be defined by the user through a suitable proof function.
`op(40, xfx, <)`	Less than.
`op(40, xfx, >)`	Greater than.
`op(40, xfx, >=)`	Greater than or equal to.
`op(40, xfx, =)`	Equality.

- **Extra operators**
  The extra operators have their semantics provided through a layer above the Prolog. The following gives an overview of the extra operators used in SPADE:

Operator declaration	Example use in expression
`op(20,xfx,**)`	Exponentiation. (`A ** B` is `A` multiplied by itself `B` times.)
`op(200,fy,not)`	Boolean negation.
`op(220,yfx,or)`	Boolean or.
`op(220,yfx,xor)`	Boolean xor.
`op(210,yfx,and)`	Boolean and.
`op(235,yfx,<->)`	Boolean equivalence.
`op(230,xfy,->)`	Boolean implication.
`op(40,yfx,<=)`	Less than or equal to.
`op(40,yfx,<>)`	Not equal to. (`A<>B` is equivalent to `not(A=B)`.)
`op(100,yfx,in)`	Element in set.
`op(100,yfx,not_in)`	Element not in set.
`op(40,fy,set)`	Denote a set.
`op(31,yfx,\/)`	Set union.
`op(21,yfx,/\)`	Set intersection.
`op(31,yfx,\)`	Set lacking.
`op(35,yfx,@)`	Sequence append.
`op(20,xf,'~')`	Initial value identifier. This is the SPADE transportation of the same entity used in Spark annotations to identify a variables initial value.

- **Special atoms**
  The special atoms have their semantics provided through a layer above the Prolog. The following gives an overview of the special atoms used in SPADE:

Special atom	Example use in expression
`true`	Logic true.
`false`	Logic false.
`integer`	Integer type.
`boolean`	Boolean type.
`real`	Real type. The integer type is a subset of this type.

- **Special predicates**
  The special predicates have their semantics provided through a layer above the Prolog. The meanings for these are defined in [5] and [1].

  The further special predicates, available inside the condition part of a rule, have their meanings defined in [5] section 5.4. Note that there exists instances of these condition predicates accepted by SPADE but not documented here. These missing predicates are listed below:

Special predicate	Example use in expression
`not`	Prolog negation of legal condition predicate.
`type`	Direct access to FDL type predicate.
`enumeration`	Direct access to FDL enumeration predicate.
`in`	Element is in set. Note that the Prolog for this could not be found. A similar operator is declared as infix and manipulated using this definition. However the code for `in` as a predicate could not be found.
`subset`	Is true if the first set is a subset of the second.
`subst_vbl`	Called as `subst_vbl(item_find_in, item_replace_in, expression_in, expression_out)` to substitute all item_find_in with item_replace_in in expression_in to give expression_out.
`set_union`	Set union.
`set_intersect`	Set intersection.
`set_lacking`	Set subtraction.
`nonvar`	Is not a Prolog variable.
`var`	Is a Prolog variable.
`atom`	Is Prolog atomic.
`atomic`	Is atomic.
`iss`	Safe integer expression evaluation.

- **Types**
  The type semantics define the legal types of the items available as regular atoms and predicates.

- **Extra data structures**
  The types of regular atoms and predicates are provided through a collection of further data structures. These structures are loaded from Functional Description Language (FDL) associated with the SPADE terms.

  The FDL is interpreted through a layer above the Prolog. Interpreting the FDL is not a trivial operation. CS-note 4 appendix A.5 covers some of the more common parts of the FDL structure and their meanings.

- **Definitions**
  The definition semantics define the meanings of the items available as regular atoms and predicates.

- **Extra rules**
  The definitions of regular atoms and predicates are provided through a collection of rule files. Each rule will describe what the atom or predicate is specifying. Eventually these definitions should break down into the low level semantics directly supported by SPADE. Where elaborating on an atom or predicate introduces new syntax it may be necessary to load corresponding FDL.

  The rules are interpreted through a layer above the Prolog. The treatment of rules inside spade is covered in CS-note 8.

2.3 Propose: SPADE Formula Notation

A single argument is considered for keeping the terms of the planner not specific to SPADE:

- **Support different front ends -** There are other systems capable of producing VCs. It may be worthwhile to demonstrate that the ideas embedded in the PLANNER translate to the proof of VCs in general, rather than SPADE VCs in specific. Adopting a more general term notation for the PLANNER is likely to make it more straight forward to plan VCs other than those from SPADE.

However, there are a number of reasons for not pursuing such generality:

- NuSPADE **is new** SPADE **-** It is noted that NuSPADE need only operate on the terms of SPADE. While there may be some merit in designing the PLANNER to accept other VCs, its primary use is to support SPADE. Thus introducing a general term notation is far from an essential part of NuSPADE. It is conjectured that proposing such a notation, implementing it as part of the PLANNER and the necessary SPADE coercions is likely to be costly. Further, knowing that the primary target is SPADE, this implementation is likely to be less efficient in normal operation.

- SPADE **terms are viable -** It is hinted in the suggestion of a general term notation that the SPADE notation is somehow not ideal. While the SPADE terms do prescribe a particular framework, this is not necessarily a poor framework for conducting VC proofs.
- SPADE **terms are not necessarily restrictive -** It is likely to be more difficult to convert the VCs generated from another system into SPADE rather than a general term notation. However, it is certainly not the case that this is impossible. Though considered troublesome to automate, it may quite feasible to hand-craft problems for the purposes of demonstration.

Thus the proposal is to adopt the SPADE term notation inside PLANNER.

2.3.1 Note on Implementation

All formula dependent parts of CLAM will need to be considered and if necessary reworked to adopt the SPADE term notation. This is likely to require substantial changes to the methods and critics of CLAM. Note that much of this effort is likely to tie in closely with efforts to deal with type, as discussed in section 3.

2.4 Propose: CLAM Annotations

The choice of where to store annotations is interesting. There is essentially two options:

- **Store outside the term -** The annotations are held alongside the term. This is thought of as attaching mini-blackboards to each term, upon which extra information can be added. This extra information would include, but not necessarily be restricted to, the annotations to support rippling.

  There is many practical disadvantages with this method. Ripping annotations are very closely related to the term structure. Presenting them in isolation from their term would be possible, but less intuitive. Further, there is resource issues in the space consumed by the many mini-blackboards and the extra time taken to analyse the information on these.

- **Store inside the term -** The annotations are directly embedded into the term. This translates very well where the annotations are describing rippling. The representation is both efficient and intuitive.

  However, the annotations are geared toward rippling. It restricts the possibility of recording more unusual information about a term. For example, that subterm at position `[1,1,2]` matches with term `3`. Further, there is a slight conceptual inconsistency in holding annotation and concrete term structure in the same place.

Currently there is no requirement to store information other than for rippling with a term. Thus, as implemented in other systems, the proposal is to insert the annotation directly into the term structure.

2.4.1 Note on Implementation

As the CLAM method of directly adding ripple annotations to the terms is to be maintained the changes to support this will be less. It will be necessary to transform the existing CLAM code to support the annotation of SPADE terms. However, as these routines are based on the progressive deconstruction of Prolog atoms, rather than some CLAM specific notation, the changes required are expected to be minimal.

3 Deal With Types

The proof planner primarily involves the syntactic manipulation of terms. However, to further constrain the proof planning process it is beneficial to make use of types.

3.1 Use of Type in SPADE

The type of every global identifier in a SPADE term is described through Functional Description Language (FDL). This translates to a data structure mapping SPADE identifiers to type descriptors. Some of the concrete details of this FDL data structure are given in CS-note 4 appendix A.5.

3.1.1 Using the FDL

As each new SPADE term is created it is checked to ensure that its type is consistent with the FDL. This is performed internally by deconstructing the term and testing syntax accordingly. The section of code for checking the type of syntax `element/2` is shown in Figure 3.

```
/* ARR2 */   checkhastype(element(A, IL), ET) :-
                checkhastype(A, T),
                (                                                 /*1.4*/
                   type(T, array(ITL, ET))                        /*1.4*/
                ;                                                 /*1.4*/
                   find_core_type(T, TN),                         /*1.4*/
                   type(TN, array(ITL, ET))                       /*1.4*/
                ),                                                /*1.4*/
                checkhastypelist(IL, ITL), !.
```

Figure 3: Checking the type of syntax `element/2` (typecheck5.pl line 370)

Thus there is a two phase process for each SPADE proof step, generating the modified syntax and then ensuring that this syntax satisfies the type checks. As a result of type being checked in this manner the rules in SPADE do not include type information. For example consider the equality based (see CS-note 8) SPADE rule:

```
array(1):  element(update(A,I,X),I) may_be_replaced_by X.
```

Here it must be the case that `A` is of type array, `X` is of the type of the elements of array `A` and `I` is presented in the form `[INDEX]`[3] where `INDEX` is of the type of the index range of `A`.

3.1.2 Type of Quantifiers are Outside the FDL

Not all type in SPADE is made available through FDL. Quantified variables have a local scope, excluding them from the global FDL. Instead their types are included within the syntax. For example:

```
H1:  for_all(i_ : integer, i_ >= indexrange__first and i_ <= indexrange__last
        -> element(flag, [i_]) = red or element(flag, [i_]) = white)
```

Stripping off the quantifiers (`unwrap`) will create a new hypothesis where the variable `i` becomes a global meta-variable ranging over the type of `i`. The result of doing this is shown below. Note how the type of the new variable `int_I__1` is not specified:

```
H3:  int_I__1 >= indexrange__first and int_I__1 <= indexrange__last
        -> element(flag, [int_I__1]) = red or element(flag, [int_I__1])
                   = white
```

[3] The use of [] is not explicitly subscribed inside SPADE, however some routines rely on this form. The following exploits this to make SPADE fall over:

```
CHECK|: infer element(update(a,1,2),1)=2.
Rulename pattern? |: R.
```

It may be necessary to adjust the SPADE rules to explicitly include the [] syntax.

As `int_I__1` is now global its type has been added to the FDL. Examining the FDL data structures reveals:

```
'var_const'('int_I__1', 'integer', 'p').
```

As noted in CS-note 4 the specifier `p` means a variable introduced as a result of unwrapping quantified expressions.

3.1.3 Extendible FDL

Syntax without suitable FDL types for its identifiers can not exist inside SPADE. Although, it is possible to have FDL for identifiers which do not occur in the current problem. Thus, prior to loading any new theory that will introduce new syntax it is necessary to first establish the suitable FDL. It is this situation that BUILDLIB and the RLB files cater for. Alternatively, the user can directly add FDL into SPADE using the `declare` command.

Significantly, this means that at any point during a SPADE session the user can introduce new types. Further, where automatically introducing new syntax into SPADE it is necessary to also automate the generation of corresponding FDL.

3.2 Current Use of Type in CLAM

3.2.1 Equality Type

Equality in CLAM is typed. Through using this single piece of type, CLAM can generally derive most type information, by querying the equality type of definitions and properties. For example:

```
y:pnat=>z:pnat=> mult(0,y,z) = z in pnat
```

Here the equality type is sufficient to determine that the variable `z` and the function `mult(0,y,z)` are of type `pnat`.

3.2.2 Variable Type

Where single variables occur in CLAM their type is also specified. For example:

```
x:pnat
```

3.2.3 Quantified Type

The variables of quantified expressions are typed in CLAM. For example the following CLAM term shows that the type of the universally quantified variables `y` and `z` are both `pnat`:

```
y:pnat=>z:pnat=> mult(0,y,z) = z in pnat
```

In the process of planning, CLAM may strip off a quantifier. At this point a new variable is created of the appropriate type.

3.3 Propose: SPADE Inspired Type System in CLAM

There are a few significant reasons for not adopting the current CLAM type system:

- **Does not translate to SPADE -** The rules of SPADE do not have type information. Instead this is provided explicitly via FDL or implicitly via the internal implementation of the SPADE type checker. In these cases the information required to translate into CLAM typed terms is not directly available.
- **Already removed through SPADE syntax -** As discussed in section 2, the PLANNER will adopt the SPADE term structure. Thus much of the CLAM type information embedded in the term syntax has already been proposed to be removed.

An alternative is to adopt a SPADE inspired type system. Note that inspired need not necessarily mean an exact copy. Although the SPADE type system is well structured[4] it might benefit from being expressed in a planner focused way. For example, to naturally check types inside the method-language and perform this efficiently may require an access layer above the FDL structure or transforming the FDL structure itself. Accepting this, the FDL

[4]The FDL concisely expresses type through a clean nested structure, that closely maps onto the base Ada code.

inside SPADE will need to be accessed by the PLANNER. Either the PLANNER copies over the FDL from SPADE or directly accesses the FDL from within SPADE.

In SPADE the only type information embedded in the syntax will be for quantified expressions. The PLANNER will need to be able to strip off these quantifiers and record their type accordingly. Adding these new types would correspond to extending the FDL. However this can not be done by extending the FDL held inside SPADE as it would violate the core principles outlined in CS-note 6. Thus it will be necessary to hold at least the FDL for striped quantifiers inside the PLANNER.

Accessing the FDL from within SPADE has these advantages:

- **Is more efficient -** Accessing the FDL from within SPADE prevents the copying of FDL. As SPADE and the PLANNER share the same Prolog environment there is no efficiently loss in accessing the conceptually more distant SPADE database.
- **Need not be direct predicate access -** Directly accessing the predicates of SPADE should always be viewed with caution. To avoid this a suitable collection of higher level access predicates could be constructed.
- **Need not track FDL -** At any point during a SPADE session the user can load a RLB file or perform a `declare` command. Both of these have the potential for adding new items to the FDL. Where directly accessing the SPADE database this FDL immediately becomes available. There is no need to monitor the SPADE FDL for changes and copy over the changes when they occur.

Accessing the FDL from outside SPADE has these advantages:

- **Is not clear separation -** Accessing the FDL from within SPADE is not a violation of the core principles outlined in CS-note 6 as it does not modify SPADE. However, the repeated access of the SPADE database (even through a structured interface) does make this separation less apparent.
- **Is not complete -** As noted above, the PLANNER will have to expand and internally record FDL to cater for the unravelling of SPADE quantified expressions. If accessing the FDL in SPADE this results in two separate locations where type information is being held.
- **Prevents restructuring -** Although the SPADE internal representation of the FDL is considered well structured, there may be efficiency or proof gains in restructuring all or part of this information. Any restructured FDL must be held in the PLANNER.
- **Is not large -** The size of the FDL tends to be quite small. This is a result of having one FDL structure to cover an entire collection of VCs. The type information for an identifier need only appear once, regardless of how often the identifier appears in the VCs syntax. It is estimated that this trend continues onto larger examples, though this has not been confirmed.

It is proposed to access SPADE FDL within the PLANNER to perform the type checking. Until it is considered worthwhile to restructure the FDL in some manner, it is considered most straight forward to directly access the FDL of SPADE. Where adding type checks to the PLANNER methods certain patters of type checking may be established and moved into an extended method-language.

3.3.1 Note on Implementation

It is expected that the PLANNER with limited type consideration will be sufficient for many problems. If the need for type information grows, a more sophisticated implementation of type can be considered.

4 Theory Free

It is a general ideal that theorem provers do have any theory built in. Instead the theory should be provided in the form of formal definitions and rules. Following these principles the PLANNER should either be entirely theory free or contain only the minimum amount of necessary theory knowledge.

4.1 CLAM is nearly Theory Free

In the CLAM Proof-Planner manual [3] there is a section explaining why CLAM should be theory free, and how it nearly satisfies this ideal. The points at which CLAM makes use of theory are:

- In the formulation of induction schemes.
- In the formulation of the weak fertilisation method.

Neither of these are core features of CLAM. The CLAM Proof-Planner manual outlines an approach to remove the theory from these into external theory files.

Note that it is likely that CLAM is so theory free because it is built on top of OYSTER. This is a very principled and light weight proof development system, which by design encourages keeping theory external to the system.

4.2 SPADE is not Theory Free

It is not the case that SPADE is theory free, or even nearly theory free. There are a number of properties embedded into the SPADE source code. Further, there are key points where definitions are missing. For example, SPADE is able to prove that `1-2=1` without any formal description of subtraction. The reasoning required for this is achieved by evaluating the expression `1-2` using the underlying Prolog arithmetic functions.

4.3 Propose: Use Embedded Theory With Reluctance

NuSPADE could make explicit the theory that SPADE uses implicitly. This might be achieved by adding new rules to SPADE and for all the plans generated ensure that these rules are used rather than those built in by SPADE. Unfortunately, this is likely to be a rather time consuming effort. As an example consider formalising the subtraction example above. It is first necessary to define what is meant by the numbers **1** and **2**. A possible solution is to take zero as a base case, and define successor and predecessor functions. All expressible integer numbers can be converted into applications of these functions on zero. For example, `-3` is `pred(pred(pred(0)))` while `+3` is `succ(succ(succ(0)))` and 0 is just `0`). Once in function form the numbers can be reasoned about appropriately by adding further rules in terms of `pred` and `succ`. In principle this is sound, but the effort involved is likely to be large.

It is noted that in its normal mode of behaviour the HOL theorem prover [4] avoids negative numbers. It is necessary to load special theorys about integers if negative numbers are required. This is seen as a strong indication of the complexities involved in formalising the SPADE number theory.

Thus the proposal is to grudgingly accept all theory that might be added to the PLANNER. Wherever this is done alternative solutions should be considered. However, this should be done more to keep the PLANNER free from unnecessary detail, rather than as a pursuit of the theory free ideal.

4.3.1 Note on Implementation

Only where the effort is considered small will action be taken to minimise the theory in the PLANNER. Therefore the action required for implementation should be minimal.

References

[1] J. G. P. Barnes. *High Integrity Ada The SPARK Approach.* Addison-Wesley, 1997.

[2] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[3] The DReaM Group. *The Clam proof planner, user manual and programmer manual (version 2.8.1)*, April 1999. Available from ftp://dream.dai.ed.ac.uk/pub/oyster-clam/manual .ps.gz.

[4] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[5] Praxis Critical Systems. *SPADE Proof Checker Version 1.3.* 1998.