H-PILoT User Manual
Version 1.95

Carsten Ihlemann

October 21, 2010

Abstract

H-PILoT (Hierarchical Proving by Instantiation in Local Theory extensions) is a program that employs hierarchical reasoning for a distinct class of theory extensions. It serves as a front end which produces a reduction of a set of clauses to a set of clauses of the base theory and then calls a dedicated solver for the base theory. The extensions for which these reductions are possible are called *local extensions.* They include many examples of theories found in real-life software verification.

Contents

1 Introduction

H-PILoT (Hierarchical Proving by Instantiation in Local Theory extensions) reduces queries over an extended theory to an equivalent query over the base theory. This reduction is always possible in the case of so-called *local extensions* [4]. This approach is quite relevant for real-life verification tasks where one typically faces queries over a mixed bag of (multi-sorted) theories, e.g lists or arrays with integers as indices and some type of entries. Further, particularly in recent years, very efficient solvers for prominent background theories have been developed (e.g. real number and integers). However, to take advantage of these, one needs to find a way of using these provers as background theory solvers in extended cases. Even in the case of extending a theory such as the real numbers with some free (uninterpreted) functions this is not a trivial task.

To deal with these types of problems, the theory of local theory extensions was developed and H-PILoT is an implementation of this approach. H-PILoT will carry out a reduction and will hand over the transformed problem to a dedicated prover such as Yices or CVC.

Examples of local extension appearing in verification include:

- Extensions by free/uninterpreted functions.
- Extensions by monotone functions.
- Definitional extensions (case distinctions).
- Fragments of the theory of arrays.
- The theory of pointer structures.

2 Background

The theory of *locality* was developed by Harald Ganzinger and Viorica Sofronie-Stokkermans [2, 4, 5, 7]. The main idea is to limit the search space for counterexamples (hence the name). Suppose we have a set of universal clauses $\forall\mathcal{K}$ and some ground clause G and we want to check whether $\forall\mathcal{K} \models G$ or equivalently if $\forall\mathcal{K}, \neg G \models \perp$.

If $\mathcal{K}$ is a local set of (Horn) clauses we need not consider the whole universe of a model but only certain (ground) instances of $\mathcal{K}$, viz. those (ground) instances of $\mathcal{K}$ in which each term was already a ground term of $\mathcal{K}$ or G. Let us denote this set of instances of $\mathcal{K}$ by $\mathcal{K}[G]$. A theory, then, is *local* if we have for an arbitrary ground Horn clause G that $\forall\mathcal{K},\ G \models \perp \Leftrightarrow \mathcal{K}[G], G \models \perp$.

The same idea works for *theory extensions.* Here we start with a base theory T_0 and extend it by a set of clauses $\mathcal{K}$. $\mathcal{K}$ contains new function symbols (and possibly new sorts). We change the definition of $\mathcal{K}[G]$ slightly. We consider again instances of $\mathcal{K}$, but now we only care about those subterms which start with an extension function. Only of those we demand that they already appear as a ground term in $\mathcal{K}$ or G.

Now the extension $T_0 \subseteq T_0 \cup \mathcal{K}$ is called *local* if we have for each set of ground clauses G[1] that

$$T_0, \forall\mathcal{K}, G \models \bot \Leftrightarrow T_0, K[G], G \models^* \bot.$$

There is an additional twist, however, due to the fact that elements of $\mathcal{K}[G]$ need no longer be ground. We compensate by considering models with *partial functions* and *weak (partial) semantics* [4]. That is, by $T_0, K[G], G \models^* \bot$ we mean that there is no (total) model of T_0 with partial extension functions in which all ground subterms of $\mathcal{K}$ and G are defined and which satisfies $\mathcal{K}[G] \cup G$.

Not only can we get rid of universal quantifiers in a local theory extension by substituting dedicated instances but we can also carry out a full-fledged reduction to the base theory. We do this in the following manner. First we purify $T_0, K[G], G$ by introducing fresh names for all the extension terms. Call these unit clauses D. Outside D, we only use these new names. This gives us a set T_0, K_0, G_0, D where only D contains extension terms now. In a final step, we eliminate even those by replacing each definition of an extension term by a certain congruence axiom which no longer contains extension functions.

In this way, we have gotten rid of all extension terms and face now a satisfiability problem over the base theory T_0. Further, we need no longer worry about partial models. Provided we have an efficient prover for this base theory, we have now an efficient way of dealing with a large class of theory extensions. H-PILoT is a program carrying out this very reduction and handing over to such a prover.

Let us start with an easy example.

3 Examples

As a first example let us consider monotone functions over some partially ordered set. We consider two monotone functions f and g as extension over the base theory of a partial order. That is our base theory consists of the axioms for reflexivity, transitivity and anti-symmetry.

(1) $\forall x.\ x \leq x.$

(2) $\forall x, y.\ x \leq y \wedge y \leq x \rightarrow x = y.$

(3) $\forall x, y, z.\ x \leq y \wedge y \leq z \rightarrow x \leq z.$

Our extension consists of the two new function symbols together with the clauses expressing monotonicity.

(1) $\forall x, y.\ x \leq y \rightarrow f(x) \leq f(y).$

(2) $\forall x, y.\ x \leq y \rightarrow g(x) \leq g(y).$

We will always call our base theory T_0 and the extension clauses $\mathcal{K}$. Our query will usually be denoted by G. That is, we want to know whether $T_0 \cup \mathcal{K}, G \models \bot$.

[1] G may contain new constants.

In this case, we want to show that $c_0 \leq c_1 \leq d_1 \wedge c_2 \leq d_1 \wedge d_2 \leq c_3 \wedge d_2 \leq c_4 \wedge f(d_1) \leq g(d_2)$ implies $f(c_0) \leq g(c_4)$. Expressed as a satisfiability problem, this gives us the following query G:

$$c_0 \leq c_1 \leq d_1 \wedge c_2 \leq d_1 \wedge d_2 \leq c_3 \wedge d_2 \leq c_4 \wedge f(d_1) \leq g(d_2) \wedge \neg f(c_0) \leq g(c_4).$$

As an input file for H-PILoT this looks as follows (we use "R" as order relation because $\leq$ is reserved).

```
% Two monotone functions over a poset.
% Status: unsatisfiable

Base_functions:={}
Extension_functions:={(f, 1), (g, 1)}
Relations:={(R, 2)}

% R is partial order
Base := (FORALL x).     R[x, x];
        (FORALL x,y,z). R[x, y], R[y, z] --> R[x, z];
        (FORALL x,y).   R[x, y], R[y, x] --> x = y;

Clauses := (FORALL x,y). R[x, y] --> R[f(x), f(y)];
           (FORALL x,y). R[x, y] --> R[g(x), g(y)];

Query := R[c0, c1];
         R[c1, d1];
         R[c2, d1];
         R[d2, c3];
         R[d2, c4];
         R[f(d1), g(d2)];
         NOT(R[f(c0), g(c4)]);
```

In this case we have no function symbols at all in our base theory and two functions symbols f and g of arity 1 in our extension clauses. This is expressed by:

```
Base_functions:={}
Extension_functions:={(f, 1), (g, 1)}
```

We have only one relation in our (base) clauses, viz. 'R'. It has arity 2 of course. This we express by

```
Relations:={(R, 2)}
```

Relations require square brackets as seen above. The symbols $<=, <, >=$ and $>$ are reserved for arithmetic over the integers or over the reals. They may be written infix and there are provers (Yices, CVC) that "understand" arithmetic and orderings. We wouldn't have needed to axiomatize '$\leq$' at all. It is already built into Yices and CVC.

However, the above problem is more general. It concerns every partial order not only numbers. By default, H-PILoT calls SPASS. SPASS has no in-built understanding of orderings and, thus, '<=' would be just any old symbol. For clarity we used the letter 'R'.

As for the syntax of clauses, one should note that each clause must end with a semicolon, be in prenex normal form and all names meant to be (universal) variables must be explicitly quantified. *Every name which is not explicitly quantified will be considered a constant!* As we can see in our query:

```
Query := R[c0, c1];
         R[c1, d1];
         R[c2, d1];
         R[d2, c3];
         R[d2, c4];
         R[f(d1), g(d2)];
         NOT(R[f(c0), g(c4)]);
```

Note further that because the background theory, extension theory and the query must all be clauses[2], we need to break up the conjunction in our original query into a set of unit clauses. A non-unit clause may be written as above in a "sorted" manner $\varphi_1, ..., \varphi_n \rightarrow \psi_1, ..., \psi_k$ for $\varphi_1 \wedge ... \wedge \varphi_n \rightarrow \psi_1 \vee ... \vee \psi_k$, i.e. as an implication with the negated atoms in the antecedent and the (positive) atoms in the consequent, or as an arbitrary disjunctions of literals.

The name of the input files to H-PILoT can be freely chosen, although it is customary to have them have the suffix ".loc". We run H-PILoT by calling

```
hpilot.opt mono_for_poset.loc
```

H-PILoT will parse the input file, carry out the reduction and then will hand over the reduced problem to SPASS (using the same name but with the suffix ".dfg"). SPASS terminates quickly with the result that a proof exists

```
SPASS beiseite: Proof found.
Problem: mono_for_poset.dfg
SPASS derived 35 clauses, backtracked 0 clauses and kept 41 clauses.
SPASS allocated 496 KBytes.
SPASS spent     0:00:02.32 on the problem.
                0:00:00.00 for the input.
                0:00:00.00 for the FLOTTER CNF translation.
                0:00:00.00 for inferences.
                0:00:00.00 for the backtracking.
                0:00:00.10 for the reduction.

--------------------------SPASS-STOP-------------------------
```

[2]In fact, the extension clauses might be more general as we will see later.

meaning that the set of clauses is inconsistent. Just what we wanted to show.

If you would like to see more of the reduction process use the option `-prClauses`. For more on the rationale of the reduction please refer to [4].

For a more complicated example let us consider an algorithm for inserting an element x into a sorted array a with the bounds l and u. We want to check that the algorithm is correct, i.e. that the updated array a' remains sorted. This could be an invariant being checked in a verification task. There are three different cases. First, x could be smaller than any element in a (equivalently, $x < a[l]$), x could be greater than any element of a ($x > a[u]$) or, thirdly, there is a position p ($l < p < u$) such that $a[p-1] < x$ and $x \leq a[p]$. In the first two cases we put x at the first resp. last position of the array. In the third case, we insert x at position p and shift the other elements to t he right, i.e. $a'[i+1] = a[i]$ for $i > p$. We have to take care to cover aberrant cases where the array contains only 1 or 2 elements. As input it will look as follows.

```
Clauses :=
  % case 1
  (FORALL i). i = l, x <= a(i) --> a'(i) = x;
  (FORALL i). x <= a(l), l < i, i <= u + _1 --> a'(i) = a(i - _1);

  % case 2
  (FORALL i). i = u, a(i) <= x --> x <= a(l), a'(i + _1) = x;
  (FORALL i). a(u) <= x, l - _1 <= i, i < u
                       --> x <= a(l), a'(i + _1) = a(i + _1);

 % case 3
  (FORALL i). x < a(u), l <= i, i < u, a(i) < x, x <= a(i + _1)
                 --> a'(i + _1) = x;
  (FORALL i). a(l) < x, x < a(u), l <= i, i < u, x <= a(i),
                 x <= a(i + _1) --> a'(i + _1) = a(i);
  (FORALL i). a(l) < x, x < a(u), i = u + _1 --> a'(i) = a(i - _1);
  (FORALL i). a(l) < x, x < a(u), l - _1 <= i, i < u, a(i + _1) < x
                 --> a'(i + _1) = a(i + _1);

  (FORALL i,j). l <= i, i <= j, j <= u --> a(i) <= a(j);
```

with the last clause saying that a was sorted at the beginning.

There are several things to note. Most importantly, we now have a two step extension. First, an array can be simply seen as a partial function. This gives us the first extension $T_0 \subseteq T_1$. T_0 here is the theory of indices (integers, say) which we extend by the function a and the axiom for monotonicity of a. Now we update a, giving us a second extension $T_2 \supseteq T_1$ where our extension clauses $\mathcal{K}_2$ are given by the three cases above.

Of course, we need to make sure that the last extension is also local. This is easy to establish, however, because $\mathcal{K}_2$ is a definitional extension or case distinction, cf. [8]. A definitional extension is one where extension functions f only appear in the form $\varphi_i(\bar{x}) \rightarrow f(\bar{x}) = t_i(\bar{x})$ with t_i being a base theory term and the φ_i are mutually exclusive base theory clauses. This is the reason that we have written $\forall i. i = l, x \leq a(i) \rightarrow a'(i) = x$ instead of the shorter $x \leq a(l) \rightarrow a'(l) = x$: to ensure that the antecedents of the clauses are all mutually exclusive. Now we know that we are dealing with a definitional and therefore local extension. (Remember that for assessing if $T_2 \supseteq T_1$ is a local extension, T_1 is our base theory. That T_1 is itself an extension is of no moment.)

We need to tell the program that we are dealing with a chain of extensions instead of a single one. We do this by the simple expedient of declaring to which level of the chain an extension function belongs, like thus $(f, arity, level)$.

In our example that would be

```
Extension_functions:={(a', 1, 2), (a, 1, 1)}
```

The program will now automatically determine the level of an extension clause. In our example, an extension clause will have level 2 iff a' occurs in it and level 1 otherwise (level 0 means base clause).

Also note that numerals (names for integers) must be preceded by an underscore such as `_1` and that $+$ and $-$ may be written infix for readability; $(=, +, -, *, /)$ are the only functions for which this is allowed[3].
Our declarations, therefore should look like this.

```
Base_functions:={(+,2), (-, 2)}
Extension_functions:={(a', 1, 2), (a, 1, 1)}
Relations:={(<=, 2), (<, 2)}
```

All that is left to do now is add our query - the negation of
$\forall i, j.\ l \leq i \leq j \leq u \rightarrow a'(i) \leq a'(j)$ - to the mix and hand it over.
The entire file looks like this.

```
% ai.loc
% Arrays for definitional extensions;

Base_functions:={(+,2), (-, 2)}
Extension_functions:={(a', 1, 2), (a, 1, 1)}
Relations:={(<=, 2), (<, 2)}

% K
Clauses :=
  % case 1
  (FORALL i). i = l, x <= a(i) --> a'(i) = x;
  (FORALL i). x <= a(l), l < i, i <= u + _1 --> a'(i) = a(i - _1);

  % case 2
```

[3]When using SPASS they may also be written infix, but nevertheless they are just free functions for SPASS.

```
  (FORALL i). i = u, a(i) <= x --> x <= a(l), a'(i + _1) = x;
  (FORALL i). a(u) <= x, l - _1 <= i, i < u
                        --> x <= a(l), a'(i + _1) = a(i + _1);

 % case 3
  (FORALL i). x < a(u), l <= i, i < u, a(i) < x, x <= a(i + _1)
                --> a'(i + _1) = x;
  (FORALL i). a(l) < x, x < a(u), l <= i, i < u, x <= a(i),
                x <= a(i + _1) --> a'(i + _1) = a(i);
  (FORALL i). a(l) < x, x < a(u), i = u + _1 --> a'(i) = a(i - _1);
  (FORALL i). a(l) < x, x < a(u), l - _1 <= i, i < u, a(i + _1) < x
                --> a'(i + _1) = a(i + _1);

  (FORALL i,j). l <= i, i <= j, j <= u --> a(i) <= a(j);

Query := l <= m;
         m <= n;
         n <= u + _1;
         NOT( a'(m) <= a'(n) );
```

We do not need to declare a base theory here because we will be using Yices and Yices already knows arithmetic. We call H-PILoT thus.

```
hpilot.opt -yices -preprocess ai.loc
```

H-PILoT will produce a reduction, put it in a file name *ai.ys* and pass it over to Yices which will say `unsat` or `sat`. A note on the flag `-preprocess`. Establishing that some extension is local, presupposes that the extension clauses $\mathcal{K}$ be *flat* and *linear*. Flatness simply means that we have no nesting of functions. Linearity means first, that we have no variable occurring twice in any extension term and, further, that if any variable occurs in two extension terms, the terms are the same. In this example, we have non-flat clauses such as

```
(FORALL i). i = u, a(i) <= x --> x <= a(l), a'(i + _1) = x;
```

We rectify matters by a *flattening* operation - rewriting the above clause to

```
(FORALL i,j). j = i + _1, i = u, a(i) <= x --> x <= a(l), a'(j) = x;
```

This will not affect consistency of any query w.r.t. $\mathcal{K}$. It hardly improves readability however. Therefore the program will do it for you if you tell it to `-preprocess` the input.

4 Advanced features

4.1 Arrays

In [1] Bradley, Manna and Sipma showed a large fragment of the theory of arrays to be decidable, the so called *array property fragment.* This is a $\exists\forall$-fragment of the theory of arrays that imposes syntactical restrictions. In a nutshell these restrictions are:

(1) An *Index guard* is a positive Boolean combination of atoms of the form $t \leq u$ or $t = u$ where t and u are either a variable or a ground term of linear arithmetic.

(2) A universal formula of the form $(\forall \bar{x})(\varphi_I(\bar{x}) \rightarrow \varphi_V(\bar{x}))$ is an *array property* if it is flat, if φ_I is an index guard and if all occurrences of the variables are shielded by extension functions in φ_V, i.e. variables x only occur as direct array reads $a[x]$ in φ_V.

In this section we will only consider extensions by clauses of the above form. Our base theory will always be linear integer arithmetic (Presburger). In the paper [1], it was shown that a limited set of instances is sufficient to decide any query over the fragment. This is not quite locality since the set of instances is slightly larger. This led us to the generalization of *minimal locality*, see [9] for details. In H-PILoT minimal locality is also implemented.
Call

```
hpilot.opt -arrays k.loc
```

to use this feature.

Let us (again) consider the example of inserting into a sorted array a. Let d be just like a but for position k which is w and let e be just like d except maybe at position l where we have written x and similarly for c,b and a.

Our set $\mathcal{K}$ of extension clauses looks as follows.

$$\left.\begin{array}{ll}
(\forall i,j)(0 \leq i \leq j \leq n-1 \rightarrow c[i] \leq c[j]), & (1)\\
(\forall i,j)(0 \leq i \leq j \leq n-1 \rightarrow e[i] \leq e[j]), & (2)\\
(\forall i)(i \neq l \rightarrow b[i] = c[i]), & (3)\\
(\forall i)(i \neq k \rightarrow a[i] = b[i]), & (4)\\
(\forall i)(i \neq l \rightarrow d[i] = e[i]), & (5)\\
(\forall i)(i \neq k \rightarrow a[i] = d[i]). & (6)
\end{array}\right\} \mathcal{K}$$

Our query (with additional constraints) is

$$\left.\begin{array}{l} w < x < y < z, \\ 0 < k < l < n, \\ k+3 < l, \\ c[l] = x, \\ b[k] = w, \\ e[l] = z, \\ d[k] = y. \end{array}\right\} \quad G$$

The input file looks just like that (with $+$ and $-$ written prefix).

```
% Arrays for minimal locality
Base_functions:={(plus,2), (minus, 2)}
Extension_functions:={(a, 1), (b, 1), (c, 1), (d, 1), (e, 1)}
Relations:={(<=, 2)}

% K
Clauses := (FORALL i,j). _0 <= i, i <= j, j <= minus(n, _1)
                                                --> c(i) <= c(j);
           (FORALL i,j). _0 <= i, i <= j, j <= minus(n, _1)
                                                --> e(i) <= e(j);
           (FORALL i).  --> i=l, b(i) =  c(i);
           (FORALL i).  --> i=k, a(i) =  b(i);
           (FORALL i).  --> i=l, d(i) =  e(i);
           (FORALL i).  --> i=k, a(i) =  d(i);


Query := plus(w, _1)  <= x;
         plus(x, _1)  <= y;
         plus(y, _1)  <= z;
         plus(_0, _1) <= k;
         plus(k, _1)  <= l;
         plus(l, _1)  <= n;
         plus(k, _3)  <= l;
         c(l) = x;
         b(k) = w;
         e(l) = z;
         d(k) = y;
```

We need to put the requirement that e and c should be sorted in $\mathcal{K}$ because G must be ground. $\mathcal{K}$ does not yet fulfill the syntactic requirements (index guards must be positive!). We must rewrite $\mathcal{K}$ in a suitable fashion. We change an expression $i \neq l$ where i is the (universally quantified) variable to $i \leq l-1 \vee l+1 \leq i$. We have to rewrite it like this because the universally quantified variable i

must appear unshielded in the index guard. This gives us the following set of clauses.

$$\left.\begin{array}{lr}
(\forall i,j)(0 \le i \le j \le n-1 \rightarrow c[i] \le c[j]), & (1)\\
(\forall i,j)(0 \le i \le j \le n-1 \rightarrow e[i] \le e[j]), & (2)\\
(\forall i)(i \le l-1 \rightarrow b[i] = c[i]), & (3)\\
(\forall i)(l+1 \le i \rightarrow b[i] = c[i]), & (4)\\
(\forall i)(i \le k-1 \rightarrow a[i] = b[i]), & (5)\\
(\forall i)(k+1 \le i \rightarrow a[i] = b[i]), & (6)\\
(\forall i)(i \le l-1 \rightarrow d[i] = e[i]), & (7)\\
(\forall i)(l+1 \le i \rightarrow d[i] = e[i]), & (8)\\
(\forall i)(i \le k-1 \rightarrow a[i] = d[i]), & (9)\\
(\forall i)(k+1 \le i \rightarrow a[i] = d[i]). & (10)
\end{array}\right\} \mathcal{K}'$$

Also, $\mathcal{K}$ is not linear, this must also be taken care of. H-PILoT will carry out all these necessary rewrite steps for the user. He can simply input the above file to deal with this example.

You can also use a "write" function when specifying array problems. For example, use write(a, i, x) to denote a new array which is just like a except (possibly) at position i where the value of the new array is set to x. In this way we could have specified our problem above as

```
% Arrays for minimal locality with 'write'function.
Base_functions:={(+, 2), (-, 2)}
Extension_functions:={(a, 1)}
Relations:={(<=, 2)}

% K
Clauses :=
   (FORALL i,j). _0 <= i, i <= j, j <= n - _1 -->
       write(write(a,k,w), l, x)(i) <= write(write(a,k,w), l, x)(j);

   (FORALL i,j). _0 <= i, i <= j, j <= n - _1 -->
       write(write(a,k,y), l, z)(i) <= write(write(a,k,y), l, z)(j);

Query := w + _1  <= x;
         x + _1  <= y;
         y + _1  <= z;
         _0 + _1 <= k;
         k + _1  <= l;
         l + _1  <= n;
         k + _3  <= l;
```

As above, H-PILoT will also automatically split on disequations in the antecedent. Note also that since we assume that indices of arrays are integers, it makes no difference whether we write `w + _1` or `plus(w, _1)` in the input file. Linear integer arithmetic will be used (Yices is default).

4.2 Global constraints[4]

Sometimes we want to restrict the domain of the problem, e.g. we want to consider natural numbers instead of integers or we are interested in a real interval $[a, b]$ only. Yices and CVC support the definition of subtypes. When using one of these it is possible to state a global constraint on the domain of the models in the preamble like thus.

```
Interval := 0 <= x <= 1;
```

This will restrict the domain of the models of the theory to the unit interval $[0, 1]$. It is equivalent to adding the antecedent $0 \leq x \wedge x \leq 1$ for every variable x to each formula in the clauses and the query.

The bounds of the interval can also be exclusive or mixed as in

```
Interval := 0 < x <= 1;
```

or one-sided as in

```
Interval := 2 <= x;
```

As an example, consider the case of multiple-valued logic [8]. The class $\mathcal{MV}$ of all MV-algebras is the quasi-variety generated by the real unit interval $[0, 1]$ with the Łukasiewicz connectives $\{\vee, \wedge, \circ, \Rightarrow\}$, i.e. the algebra $[0, 1]_L = ([0, 1], \vee, \wedge, \circ, \Rightarrow)$. Further the Łukasiewicz connectives can be defined in terms of the real '$+, -$' and '$\leq$' giving us a local extension over the real unit interval.

Therefore, the following are equivalent:

(1) $\mathcal{MV} \models \forall \overline{x} \bigwedge_{i=1}^{n} s_i(\overline{x}) = t_i(\overline{x}) \rightarrow s(\overline{x}) = t(\overline{x})$

(2) $[0, 1]_L \models \forall \overline{x} \bigwedge_{i=1}^{n} s_i(\overline{x}) = t_i(\overline{x}) \rightarrow s(\overline{x}) = t(\overline{x})$

(3) $\mathcal{T}_0 \cup \mathsf{Def}_L \wedge \bigwedge_{i=1}^{n} s_i(\overline{c}) = t_i(\overline{c}) \wedge s(\overline{c}) \neq t(\overline{c}) \models \perp$,

where $\mathcal{T}_0$ consists of the real unit interval $[0, 1]$ with the operations $+, -$ and predicate symbol $\leq$.

For instance, we might want to establish that linearity $x \Rightarrow y. \vee .y \Rightarrow x = 1$ follows from the axioms. As an input file for H-PILoT it looks like this.

```
% file mv1.loc
% Example for MV-algebras
% The Lukasiewicz connectives can be defined
```

[4]This feature is not supported for SMT/Z3 settings.

```
% in terms of the (real) connectives +,-,<=

Base_functions:={(+, 2), (-, 2)}
Extension_functions:={(V, 1), (M, 1), (o, 1), (r, 1)}
Relations:={(<=, 2), (<, 2), (>, 2), (>=, 2)}

Interval := 0 <= x <= 1;

% K
Clauses := % definition of \/
           (FORALL x,y). x <= y --> V(x, y) = y;
           (FORALL x,y). x > y  --> V(x, y) = x;

           % definition of /\
           (FORALL x,y). x <= y --> M(x, y) = x;
           (FORALL x,y). x > y  --> M(x, y) = y;

           % definition of o
           (FORALL x,y). x + y <  _1 --> o(x, y) = _0;
           (FORALL x,y). x + y >= _1 --> o(x, y) = (x + y) - _1;

           % definition of =>
           (FORALL x,y). x <= y --> r(x, y) = _1;
           (FORALL x,y). x > y  --> r(x, y) = (_1 - x) + y;

Query := % linearity: x => y . \/ . y => x = 1
         NOT(V(r(a, b), r(b, a)) = _1);
```

4.3 Types

In default mode, using SPASS, H-PILoT hands over a set of general first-order formulas without types. However, H-PILoT also provides support for the standard types `int, real, bool` and for free types. When using CVC or Yices the default type is `int`, for REDLOG it is `real`. The default type does not need to be specified in the input file. You can also use the `-real` flag to set the default type to real for Yices and CVC [5].

Free types are specified as `free#`$i, i = 1, 2..$ or simply as `free` if there is only one free type. When using free types the flag `-freeType` must be set. Only Yices and CVC are able to handle free types (Yices is default).

When using mixed type in one input file the types of the functions and the constants need to be declared. H-PILoT does not try to deduce types. If the domain of a function is the same as the range it is enough to specify the domain as in

$(foo, arity, level, domainType)$

if they differ say

$(foo, arity, level, domainType, rangeType)$.

Constants are simply declared as

$(name, type)$.

We offer the following example taken from [6].

```
% Pointers
% status unsatisfiable
Base_functions:={(+,2), (-, 2)}
Extension_functions:={(next, 1, 1, free#1), (prev, 1, 1, free#1),
                      (priority, 1, 1, free#1, real),
                         (state, 1, 1, free#1, free#2)}
Relations:={(>=, 2)}
Constants:={(null, free#1), (eps, real), (a, free#1), (b, free#1),
             (RUN, free#2), (WAIT, free#2), (IDLE, free#2)}

% K
Clauses :=
  (FORALL x). OR(state(x) = RUN, state(x) = WAIT, state(x) = IDLE);
  % prev and next are inverse
  (FORALL p). OR(p = null, prev(next(p)) = null, prev(next(p)) = p);
  (FORALL p). --> p = null, next(prev(p)) = null, next(prev(p)) = p;
  (FORALL p, q). next(q) = next(p) --> p = null, q = null, p = q;
  (FORALL p, q). prev(q) = prev(p) --> p = null, q = null, p = q;
  (FORALL p). --> p = null, next(p) = null, state(p) = IDLE,
                  state(next(p)) = IDLE, state(p) = state(next(p));
  (FORALL p). OR(p = null, next(p) = null, NOT(state(p) = RUN),
                    priority(p) >= priority(next(p)));
```

[5]But recall that they are only able to handle linear arithmetic for reals.

```
Query := NOT(eps = _5);
         NOT(eps = _6);
         priority(a) = _5;
         priority(b) = _6;
         a = prev(b);
         state(a) = RUN;
         NOT(next(a) = null);
         NOT(a = null);
         NOT(b = null);
```

4.4 Pointers

We consider pointer problems over a two-typed language. One of which is the type `pointer` and the other is some scalar type. The scalar type can be concrete like `real`, say, or kept abstract in which case it is written as `scalar`. In [3] Necula and McPeak investigate reasoning about pointers in this two-typed language with the additional features that there are only two function types, viz. `pointer` $\rightarrow$ `pointer` and `pointer` $\rightarrow$ *scalar*, where *scalar* is either a concrete scalar type (e.g. `real`) or the abstract scalar type.

H-PILoT allows the user to specify multiple pointer types $\mathsf{p}_1, \ldots, \mathsf{p}_n$, $n \geq 1$ as `pointer#1`,...,`pointer#n` (or simply as `pointer` if $n = 1$). There are two types of pointer functions. Functions Σ^p_{ij} of signature $\mathsf{p}_i \rightarrow \mathsf{p}_j$ from one pointer type to another and functions Σ^s_i of signature $\mathsf{p}_i \rightarrow \mathsf{s}$ from a pointer type into scalars. A constant null_i of sort $\mathsf{p_i}$ exists for each i, $1 \leq i \leq n$. The only predicate of sort $\mathsf{p_i}$, $1 \leq i \leq n$ is equality; predicates of sort s are allowed and can have any arity.

The axioms considered are all of the form

$$\forall \bar{p}.\ \mathcal{E} \vee \mathcal{C}$$

where $\bar{p}$ is a set of pointer variables containing all the pointer variables occurring in $\mathcal{E} \vee \mathcal{C}$, $\mathcal{E}$ contains disjunctions of pointer equalities and $\mathcal{C}$ is a disjunction of scalar constraints (i.e. literals of scalar type). $\mathcal{E} \vee \mathcal{C}$ may also contain free variables of scalar type or, equivalently, free scalar constants.

In order to rule out null pointer errors, Necula and McPeak demanded that pointer terms appearing below a function should not be null. That is for all pointer terms $f_1(f_2(\ldots f_n(p)))$, $f_i \in \Sigma^p \cup \Sigma^s$, $i = 1,..,n$, occurring in the axiom, the axiom also contains the disjunction $p = \mathsf{null}_{j_1} \vee f_n(p) = \mathsf{null}_{j_2} \vee \cdots \vee f_2(\ldots f_n(p)) = \mathsf{null}_{j_n}$ where the null_{j_k} are of the appropriate type.

We will call pointer/scalar formulas complying with this restriction *nullable.* It was shown in [9] that nullable pointer/scalar axioms of the above form are (ψ) stably local. This result allows the integration of pointer reasoning with the above features into H-PILoT .

As an example (cf. [3]) consider doubly linked lists of decreasing priorities.

```
Base_functions:={(+,2), (-, 2)}
Extension_functions:={(next, 1, 1, pointer),
                      (prev, 1, 1, pointer),
                      (priority, 1, 1, pointer, real)}
Relations:={(>=, 2)}
Constants:={(a, pointer), (b, pointer)}

Clauses :=
     (FORALL p). prev(next(p)) = p;
     (FORALL p). --> next(prev(p)) = p;
     (FORALL p). --> q = null, priority(p) >= priority(next(p));
```

```
Query := priority(a) = _5;
         priority(b) = _6;
         a = prev(b);
         NOT(a = null);
         NOT(b = null);
```

H-PILoT can be called without any parameters because the keyword `pointer` is present. This will set H-PILoT into pointer mode so that it will add all the nullable terms to the axioms and use the specific ψ-locality required.

Because the scalar type is concrete here (`real`) H-PILoT will use Yices as the prover (its default for arithmetic). If we want to leave the scalar type abstract we could write something like

```
%psiPointers.scalar.loc
Base_functions:={}
Extension_functions:={(next, 1, 1, pointer),
                      (prev, 1, 1, pointer),
                      (priority, 1, 1, pointer, scalar)}
Relations:={}
Constants:={(a, pointer), (b, pointer), (c5, scalar), (c6, scalar)}

Clauses := (FORALL p). prev(next(p)) = p;
           (FORALL p). next(prev(p)) = p;
           (FORALL p). NOT(priority(p) = priority(next(p)));


Query := priority(a) = c5;
         priority(b) = c6;
         a = prev(b);
         c5 = c6;
         NOT(a = null);
         NOT(b = null);
```

We again can simply type `hpilot.opt -preprocess psiPointers.scalar.loc` without any parameters. H-PILoT will again recognize this as a pointer problem and use Yices as default, this time because of the free type `scalar`.

Pointer extensions can be fused with other types of extensions in a hierarchy. However, due to the different types of locality that need to be employed, the user must specify which levels are pointer extensions. He does this by using the keyword `Stable`[6].

For example the header of a more complicated verification task which mixes different pointer types might look like this.

```
Base_functions:={(-, 2), (+, 2)}
Extension_functions:=
```

[6]Which stands for stable locality which is the type of locality used for pointer extensions.

```
   { % level 4
     (next',1,4, pointer#2,pointer#2), (pos',1,4,pointer#2,real)
     % level 3
     (next,1,3,pointer#2,pointer#2), (pos,1,3,pointer#2,real),
     (spd,1,3,pointer#2,real), (segm,1,3,pointer#2,pointer#1),
     % level 2
     (bd,1,2,real,real),
     % level 1
     (lmax,1,1,pointer#1,real), (length,1,1,pointer#1,real),
     (nexts,1,1,pointer#1,pointer#1), (alloc,1,1,pointer#1,int)}

Relations :={(<=, 2), (>=, 2), (>, 2), (<, 2)}

Constants:= {(t3,pointer#2), (t2,pointer#2), (t1,pointer#2),
              (d,real), (State0,int), (s,pointer#1), (State1,int)}

Stable := 1, 3;
```

Note that the type `pointer#2` must be declared higher than `pointer#1` because `pointer#2` refers to `pointer#1` but not vice versa.

Here is an example of a header with multiple pointer types at one level.

```
Base_functions:={(-, 2), (+, 2)}
Extension_functions:={
(bd,1,1,real,real),

% pointer type 1
(lmax,1,2,pointer#1,real),(length,1,2,pointer#1,real),
(alloc,1,2,pointer#1,int), (nexts,1,2,pointer#1,pointer#1),

% pointer type 2
(next,1,2,pointer#2,pointer#2),(pos,1,2,pointer#2,real),
(spd,1,2,pointer#2,real), (segm,1,2,pointer#2,pointer#1),

% third extension: updated functions
(next',1,3,pointer#2,pointer#2),
}

Relations:={(<= , 2), (>=, 2), (>, 2), (<, 2)}

Constants:={
(t7,pointer#2), (t6,pointer#2), (t5,pointer#2),(t4,pointer#2)}

Stable := 2;
```

4.5 Extended locality

For some applications we would like that the extension clauses $\mathcal{K}$ were allowed to be more complicated, say being inductive ($\forall\exists$) instead of universal. Consider the following example, taken from [9].

Consider a parametric number m of processes. The priorities associated with the processes (non-negative real numbers) are stored in an array p. The states of the process – enabled (1) or disabled (0) are stored in an array a. At each step only the process with maximal priority is enabled, its priority is set to x and the priorities of the waiting processes are increased by y. This can be expressed with the following set of axioms which we denote by $\mathsf{Update}(p, p', a, a')$

$$\begin{array}{l}
\forall i(1 \leq i \leq m \wedge \quad (\forall j(1 \leq j \leq m \wedge j{\neq}i \rightarrow p(i){>}p(j))) \rightarrow a'(i) = 1) \\
\forall i(1 \leq i \leq m \wedge \quad (\forall j(1 \leq j \leq m \wedge j{\neq}i \rightarrow p(i){>}p(j))) \rightarrow p'(i){=}x) \\
\forall i(1 \leq i \leq m \wedge \neg(\forall j(1 \leq j \leq m \wedge j{\neq}i \rightarrow p(i){>}p(j))) \rightarrow a'(i){=}0) \\
\forall i(1 \leq i \leq m \wedge \neg(\forall j(1 \leq j \leq m \wedge j{\neq}i \rightarrow p(i){>}p(j))) \rightarrow p'(i){=}p(i){+}y)
\end{array}$$

where x and y are considered to be parameters. We may need to check whether if at the beginning the priority list is injective, i.e. formula $(\mathsf{Inj})(p)$ holds:

$$(\mathsf{Inj})(\mathsf{p}) \quad \forall i, j(1 \leq i \leq m \wedge 1 \leq j \leq m \wedge i \neq j \rightarrow p(i) \neq p(j))$$

then it remains injective after the update, i.e. check whether:

$$(\mathsf{Inj})(p) \wedge \mathsf{Update}(p, p', a, a') \wedge (1 \leq c \leq m \wedge 1 \leq d \leq m \wedge c \neq d \wedge p'(c) = p'(d)) \models \perp.$$

The problem here is that we need to deal with alternations of quantifiers in the extension. To deal with cases like this, the notion of *locality* needs to be extended. That is exactly what was done [4, 7]. In *extended locality* the extension formulas $\mathcal{K}$ may now have the form $\forall x_1, ..., x_n.\ (\Phi(x_1, \ldots, x_n) \vee C(x_1, \ldots, x_n))$, where $\Phi(x_1, \ldots, x_n)$ is an *arbitrary first-order formula* in the base signature with free variables $x_1, \ldots, x_n$ and $C(x_1, \ldots, x_n)$ is a *clause* in the extended signature.

In H-PILoT extended locality is also implemented. Extended clauses may be either written as $\forall \bar{x}.\ (\Phi(\bar{x}) \vee C(\bar{x}))$ or as $\forall \bar{x}.\ (\Phi(\bar{x}) \rightarrow C'(\bar{x}))$. The input file for H-PILoT simply looks like this.

```
% Updating of priorities of processes
% File update_AE.loc
Base_functions:={(+,2), (-, 2)}
Extension_functions:={(a’, 1, 2, bool), (a, 1, 1, bool),
                      (p’, 1, 2, real), (p, 1, 1, real)}
Relations:={(<=, 2), (<, 2), (>, 2)}
Constants:={(x, real), (y, real)}

% K
Clauses :=
   (FORALL i). _1 <= i, i <= m --> _0 <= p(i);
   (FORALL i). { AND(_1 <= i, i <= m,
      (FORALL j). (AND(_1 <= j, j <= m, NOT(j = i)) --> p(i) > p(j)))}
           --> a’(i) = _1;
   (FORALL i). { AND(_1 <= i, i <= m,
      (FORALL j). (AND(_1 <= j, j <= m, NOT(j = i)) --> p(i) > p(j)))}
            --> p'(i) = x;
   (FORALL i). { AND(_1 <= i, i <= m,
      NOT((FORALL j,i).(AND(_1 <= j, j <= m, NOT(j = i))
                                                  --> p(i) > p(j))))}
            --> a'(i) = _0;
   (FORALL i). { AND(_1 <= i, i <= m,
      NOT((FORALL j).(AND(_1 <= j, j <= m, NOT(j = i))
                                                  --> p(i) > p(j))))}
            --> p'(i) = p(i) + y;

   (FORALL i,j). _1 <= i, i <= m, _1 <= j, j <= m, p(i) = p(j) --> i = j;

Query := _1 <= c;
         c <= m;
         _1 <= d;
         d <= m;
         x <= _0;
         y > _0;
         NOT(c=d);
         p'(c) = p'(d);
```

The curly braces '{', '}' are required to demarcate the beginning and the end of the base formula Φ.

4.6 Clausification

H-PILoT also provides a simple clausifier for ease of use. First-order formulas can be given as input (with the syntax of the above subsection) and H-PILoT will translate them into clausal normal form (CNF). The CNF-translator does not provide the full functionality of FLOTTER[7] but should be quite powerful enough for most applications.

As a simple example consider the following.

```
% cnf.fol

Base_functions:={(delta, 2), (abs, 1), (-, 2)}
Extension_functions:={(f, 1)}
Relations:={}



Formulas :=
    (FORALL eps, a, x). (_0 < eps -->
             AND( _0 < delta(eps, a),
                 (abs(x - a) < delta(eps, a) --> abs(f(x) - f(a)) < eps)));

Query :=
```

H-PILoT will clausify the `Formulas` for us. To see the output let us use

```
     hpilot.opt -preprocess -prClauses cnf.fol
```

We will see an output like

```
!-Adding formula:
   (FORALL eps, a, x).
        (_0 < eps --> AND( _0 < delta(eps, a), (abs(-(x, a)) < delta(eps, a)
           --> abs(-(f(x), f(a))) < eps)))
!-add_formulas
!-We have 1 levels.
!-done
!-Our base theory is:
!-empty.
!-Clausifying formulas...
!-(FORALL z_1, z_3). OR( _0 < delta(z_1, z_3), NOT(_0 < z_1))
!-(FORALL z_1, z_2, z_3). OR( NOT(abs(-(z_2, z_3)) < delta(z_1, z_3)),
                              abs(-(f(z_2), f(z_3))) < z_1, NOT(_0 < z_1))
!-Yielding 2 new clauses:
!-[z_1, z_2, z_3] abs(-(z_2, z_3)) < delta(z_1, z_3), _0 < z_1
                         ---> abs(-(f(z_2), f(z_3))) < z_1  L: 1;
!-[z_1, z_3] _0 < z_1 ---> _0 < delta(z_1, z_3)  L: 0;
```

[7]It only provides standard formula renaming and standard Skolemization.

```
!-After rewriting we have as clauses K:
!-[z_1, z_2, z_3] abs(-(z_2, z_3)) < delta(z_1, z_3), _0 < z_1
                            ---> abs(-(f(z_2), f(z_3))) < z_1  L: 1;
!-[z_1, z_3] _0 < z_1 ---> _0 < delta(z_1, z_3)  L: 0;
```

telling us that the above formula resulted in two new clauses (in addition to those outright given under `Clauses`, viz.

$$\forall z_1, z_3.\ \ 0 < delta(z_1, z_3) \vee \neg(0 < z_1)$$

and

$$\forall z_1, z_2, z_3.\ \ \neg(abs(z_2 - z_3) < delta(z_1, z_3)) \vee abs(f(z_2) - f(z_3)) < z_1 \vee \neg(0 < z_1).$$

In this case no ground clause resulted and H-PILoT stops.

5 Parameters of H-PILoT

H-PILoT has several input parameters controlling its behavior. They can be listed by calling `hpilot.opt -help`.

-min	Use minimal Locality. This is only relevant for the array property fragment right now.
-prClauses	Produce output: print all the clauses calculated and used.
-noProver	Do not hand over to prover, just produce output.
-yices	Use Yices as background solver. arithmetics: 'plus', '+' etc. are predefined as are the order relations $\leq, \geq, <, >$. Numbers can also be given in the input. ($\_i$ is translated to i). Numbers are integers by default (use '-real' for real numbers).
-cvc	Use CVC as background solver. Arithmetic is predefined as with '-yices'
-z3	Use Z3 as background solver. Arithmetic is predefined as with '-yices'
-flatten	Flatten clauses first.
-linearize	Linearize clauses first.
-flattenQuery	Flattens the query first.
-preprocess	Preprocess input: flatten/linearize clauses, flatten query. In array-context: split clauses which contain inequalities like $i \neq j, ...$ into two clauses.
-noSeparation	Stop at calculating instances $\mathcal{K}[G]$. Don't introduce names for extension terms and don't reduce to base theory. See [4] for background.
-unPseudofy	Eliminate pseudo-quantifiers like $\forall i.i = 3...$ before handing over to a prover[8].
-noProcessing	No computation. Just translate into prover syntax and hand over. Overrides '-preprocess' and turns off clausification. When using this flag one should provide the domains of functions too. When used in combination with CVC there may arise problems with boolean types.[9]
-clausification	(true/false). Turns clausification of general formulas on or off. The default is 'true'. Implies '-noProcessing'.
-real	Use real instead of integer linear arithmetic as the default type.
-redlog	Call Redlog for base prover. Assumes '-real'.
-version	Print version number.
-freeType	Enables the use of an unspecified type 'free' in addition to 'real' and 'int'. Only CVC, Z3 and Yices accept free types. Yices is default.

[9]This is automatically carried out if we have a multiple-step extension. This is because the next step can only be carried out if the current step resulted in ground clauses.

[9]This is because CVC only provides booleans as bit-vectors of length 1. The type 'BOOLEAN' is the type of formulas.

-arrays	Use settings for array. This combines -preprocess, -min and -arith; it also splits clauses on negative equalities.
-model	Gives a counter-model for satisfiable queries. Needs Yices or CVC (implies Yices by default).
-smt	Produce SMT-LIB output without calling a prover.
-isLocal	Use this flag (true/false) to tell the program whether all the extensions are local or not. In the first case h-pilot is a full decision procedure in the latter it will only solve unsatisfiable problems. Default is false. This matters only if H-PILoT cannot derive a contradiction. In that case this means that there really is none only if the extensions are local.
-renameSubformulas	(true/false). Use subformula renaming when clausifying the input in order to avoid exponential growth. Default is true.
-sc	"Sanity check". If this flag is set the query clauses are left out. This is meant as a test in order to ensure that the axioms are not inconsistent already on their own.
-verbosity	This flags determines the verbosity level (0,1,2) in the output gotten from '-prClauses'. From taciturn to garrulous. Default is 0

6 Error handling

In case you get a parsing error you can use

`export OCAMLRUNPARAM='p'` (in *bash* syntax).

This will give you a walk-through of the parsing process. This is of great help in localizing syntax errors. To turn it back off use

`export OCAMLRUNPARAM=''`.

7 The input grammar

⟨start⟩ ::= ⟨base_functions⟩ ⟨extension_functions⟩ ⟨relations⟩ ⟨constants⟩ ⟨interval⟩
⟨baseTheory⟩ ⟨formulasOrClauses⟩ ⟨groundformulas⟩ ⟨query⟩

⟨base_functions⟩ ::= `Base_functions :=` { ⟨function_list⟩ }

⟨extension_functions⟩ ::= `Extension_functions :=` { ⟨function_list⟩ }

⟨relations⟩ ::= `Relations :=` { ⟨relation_list⟩ }

⟨constants⟩ ::= ϵ | `Constants :=` { constant_list }

⟨interval⟩ ::= ϵ
| `Interval :=` ⟨int⟩ ⟨sm⟩ ⟨identifier⟩`;`
| `Interval :=` ⟨identifier⟩ ⟨sm⟩ ⟨int⟩`;`
| `Interval :=` ⟨int⟩ ⟨sm⟩ ⟨identifier⟩ ⟨sm⟩ ⟨int⟩`;`

⟨base_theory⟩ ::= ϵ | `Base :=` ⟨clause_list⟩

⟨formulasOrClauses⟩ ::= ϵ | ⟨formulas⟩ | ⟨clauses⟩
| ⟨formulas⟩⟨clauses⟩ | ⟨clauses⟩⟨formulas⟩

⟨ground_formulas⟩ ::= ϵ | `Ground_Formulas :=` ⟨formula_list⟩

⟨query⟩ ::= `Query :=` ⟨ground_clauses⟩

⟨formulas⟩ ::= `Formulas :=` ⟨formula_list⟩

⟨clauses⟩ ::= `Clauses :=` ⟨clause_list⟩

⟨function_list⟩ ::= ϵ | ⟨function⟩ ⟨additional_functions⟩

⟨additional_functions⟩ ::= ϵ | `,` ⟨function⟩ ⟨additional_functions⟩

⟨relation_list⟩ ::= ϵ | ⟨relation⟩ ⟨additional_relations⟩

⟨additional_relations⟩ ::= ϵ | `,` ⟨relation⟩ ⟨additional_relations⟩

⟨relation⟩ ::= `(` ⟨uneqs⟩ `,` ⟨int⟩ `)` | `(` ⟨identifier⟩ `,` ⟨int⟩ `)`

⟨function⟩ ::= `(` ⟨identifier⟩ `,` ⟨int⟩ `)`
| `(` ⟨arithop⟩ `,` ⟨int⟩ `)`
| `(` ⟨arithop⟩ `,` ⟨int⟩ `,` ⟨int⟩ `, int )`
| `(` ⟨arithop⟩ `,` ⟨int⟩ `,` ⟨int⟩ `, real )`
| `(` ⟨identifier⟩ `,` ⟨int⟩ `,` ⟨int⟩ `)`
| `(` ⟨identifier⟩ `,` ⟨int⟩ `,` ⟨int⟩ `,` ⟨domain⟩ `)`
| `(` ⟨identifier⟩ `,` ⟨int⟩ `,` ⟨int⟩ `,` ⟨domain⟩ `,` ⟨domain⟩ `)`

⟨domain⟩ ::= `bool` | `int` | `real` | `pointer` | `pointer#` ⟨int⟩
| `scalar` | `free` | `free#` ⟨int⟩

```
⟨base_clause_list⟩ ::= ϵ | ⟨clause⟩ ; ⟨additional_base_clauses⟩

⟨additional_base_clauses⟩ ::= ϵ | ⟨base_clause⟩ ; ⟨additional_base_clauses⟩

⟨constant_list⟩ ::= ⟨constant⟩ ⟨additional_constants⟩

⟨additional_constants⟩ ::= ϵ | , ⟨constant⟩ ⟨additional_constant⟩

⟨constant⟩ ::= ( ⟨identifier⟩ , bool )
            | ( ⟨identifier⟩ , int )
            | ( ⟨identifier⟩ , real )
            | ( ⟨identifier⟩ , scalar )
            | ( ⟨identifier⟩ , pointer )
            | ( ⟨identifier⟩ , pointer# ⟨int⟩ )
            | ( ⟨identifier⟩ , free )
            | ( ⟨identifier⟩ , free# ⟨int⟩ )

⟨base_clause⟩ ::= ⟨clausematrix⟩ | ⟨universalQuantifier⟩ ⟨clausematrix⟩

⟨formula_list⟩ ::= ⟨formula⟩ | ⟨formula⟩ ; ⟨additional_formulas⟩

⟨additional_formulas⟩ ::= ϵ | ⟨formula⟩ ; ⟨additional_formulas⟩

⟨clause_list⟩ ::= ⟨clause⟩ | ⟨clause⟩ ; ⟨additional_clauses⟩

⟨additional_clauses⟩ ::= ϵ | ⟨clause⟩ ; ⟨additional_clauses⟩

⟨clause⟩ ::= ⟨clausematrix⟩
          | ⟨universalQuantifier⟩ ⟨clausematrix⟩
          | { ⟨formula⟩ } OR ⟨clausematrix⟩
          | ⟨universalQuantifier⟩ { ⟨formula⟩ } OR ⟨clausematrix⟩
          | { ⟨formula⟩ } --> ⟨clausematrix⟩
          | ⟨universalQuantifier⟩ { ⟨formula⟩ } --> ⟨clausematrix⟩

⟨universalQuantifier⟩ ::= ( FORALL ⟨variables⟩ ) .

⟨variables⟩ ::= ⟨name⟩ ⟨additional_variable⟩

⟨additional_variables⟩ ::= ϵ | , ⟨name⟩ ⟨additional_variables⟩

⟨ground_clauses⟩ ::= ϵ | ⟨clausematrix⟩ ; ⟨ground_clauses⟩

⟨clausematrix⟩ ::= ⟨literal⟩ | ⟨disjunctive_clause⟩ | ⟨sorted_clause⟩

⟨formula⟩ ::= ⟨atom⟩
           | NOT ( ⟨formula⟩ )
           | OR ( ⟨formula⟩ ⟨formula_plus⟩ )
           | AND ( ⟨formula⟩ ⟨formula_plus⟩ )
           | ( ⟨formula⟩ --> ⟨formula⟩ )
           | ( ⟨formula⟩ <--> ⟨formula⟩ )
           | ( FORALL ⟨variables⟩ ) . ⟨formula⟩
           | ( EXISTS ⟨variables⟩ ) . ⟨formula⟩
```

```
⟨formula_plus⟩ ::= , ⟨formula⟩ ⟨formula_star⟩

⟨formula_star⟩ ::= ϵ | , ⟨formula⟩ ⟨formula_star⟩

⟨disjunctive_clause⟩ ::= OR ( ⟨literal⟩ ⟨literal_plus⟩ )

⟨literal_plus⟩ ::= , ⟨literal⟩ ⟨literal_star⟩

⟨literal_star⟩ ::= ϵ | , ⟨literal⟩ ⟨literal_star⟩

⟨sorted_clause⟩ ::= ⟨atom_list⟩ --> ⟨atom_list⟩

⟨atom_list⟩ ::= ϵ | ⟨atom⟩ ⟨atom_star⟩

⟨atom_star⟩ ::= ϵ | , ⟨atom⟩ ⟨atom_star⟩

⟨literal⟩ ::= ⟨atom⟩ | NOT ( ⟨atom⟩ )

⟨atom⟩ ::= ⟨equality_atom⟩ | ⟨ineq_atom⟩ | ⟨predicate_atom⟩

⟨equality_atom⟩ ::= ⟨term⟩ = ⟨term⟩

⟨ineq_atom⟩ ::= ⟨term⟩ ⟨uneqs⟩ ⟨term⟩

⟨predicate_atom⟩ ::= ⟨identifier⟩ [ ⟨term⟩ ⟨additional_terms⟩ ]

⟨arguments⟩ ::= ⟨term⟩ ⟨additional_terms⟩

⟨additional_terms⟩ ::= ϵ | , ⟨term⟩ ⟨additional_terms⟩

⟨term⟩ ::= ⟨name⟩
        | ⟨operator⟩ ( ⟨arguments⟩ )
        | ⟨array⟩ ( ⟨arguments⟩ )
        | ⟨update⟩ ( ⟨arguments⟩ )
        | ⟨term_arith⟩ ⟨arithop⟩ ⟨term_arith⟩
        | - ⟨term_arith⟩

⟨term_arith⟩ ::= ⟨name⟩
              | ⟨operator⟩ ( ⟨arguments⟩ )
              | ( ⟨term_arith⟩ ⟨arithop⟩ ⟨term_arith⟩ )
              | ( - ⟨term_arith⟩ )

⟨sm⟩ ::= ≤ | <

⟨arithop⟩ ::= + | - | * | /

⟨uneqs⟩ ::= <= | >= | < | >

⟨operator⟩ ::= ⟨identifier⟩

⟨array:⟩ ::= write ( ⟨identifier⟩ , ⟨term⟩ , ⟨term⟩ )
          | write ( ⟨array⟩ , ⟨term⟩ , ⟨term⟩ )

⟨update⟩ ::= update ( ⟨identifier⟩ , ⟨term⟩ , ⟨term⟩ )
          | update ( ⟨update⟩ , ⟨term⟩ , ⟨term⟩ )
```

$\langle$*name*$\rangle$ ::= $\langle$*identifier*$\rangle$

$\langle$*identifier*$\rangle$ ::= *any string consisting of letters and numbers starting with a letter. It may end with "' ".*

$\langle$*int*$\rangle$ ::= *any non-negative number.*

References

[1] A. R. Bradley, Z. Manna, and H.B. Sipma, *What's decidable about arrays?*, Proceedings of 7th Int. Conf. on Verification, Model Checking and Abstract Interpretation, LNCS, vol. 3855, 2006.

[2] Harald Ganzinger, *Relating semantic and proof-theoretic concepts for polynomial time decidability of uniform word problems.*, Proceedings of the 16th IEEE Symposion on Logic in Computer Science (LICS'01), IEEE Comp. Soc. Press, 2001, pp. 81–92.

[3] S. McPeak and G.C. Necula., *Data structure specifications via local equality axioms.*, Computer Aided Verification, 17th International Conference (CAV 2005), LNCS, vol. 3576, 2005, pp. 476–490.

[4] V. Sofronie-Stokkermans, *Hierarchic reasoning in local theory extensions.*, 20th International Conference on Automated Deduction (CADE-20) (R. Nieuwenhuis, ed.), LNAI, no. 3632, Springer, 2005, pp. 219–234.

[5] ———, *Interpolation in local theory extensions.*, International Joint Conference on Automated Reasoning (IJCAR 2006) (U. Furbach and Natarajan Shankar, eds.), LNAI, no. 4130, Springer, 2006, pp. 235–250.

[6] ———, *Local reasoning in verification*, Proceedings of the Verification Workshop VERIFY'06, 2006.

[7] ———, *Hierarchical and modular reasoning in complex theories: The case of local theory extensions.*, Proceedings 6th International Symposion on Frontiers of Combining Systems (FroCos 2007), LNCS, no. 4720, Springer, 2007, invited paper, pp. 47–71.

[8] Viorica Sofronie-Stokkermans and Carsten Ihlemann, *Automated reasoning in some local extensions of ordered structures*, Proceedings of ISMVL 2007., IEEE Computer Society, 2007, `http://dx.doi.org/10.1109/ISMVL.2007.10`.

[9] Viorica Sofronie-Stokkermans, Swen Jacobs, and Carsten Ihlemann, *On local reasoning in verification*, Proceedings of TACAS'08., LNCS, vol. 4963, Springer, 2008, pp. 265–281.