Mode Checking in HAL

María García de la Banda[1], Peter J. Stuckey[2], Warwick Harvey[1], and Kim Marriott[1]

[1] Monash University, Clayton 3168, Australia,
mbanda@csse.monash.edu.au
[2] University of Melbourne, Parkville 3152, Australia

Abstract. Recent constraint logic programming (CLP) languages, such as HAL and Mercury, require type, mode and determinism declarations for predicates. This information allows the generation of efficient target code and the detection of many errors at compile-time. However, mode checking in such languages is difficult: The compiler is required to appropriately re-order literals in the predicate's definition for each predicate mode declaration. The task is further complicated by the need to handle complex instantiations which interact with type declarations, higher order functions and predicates, and automatic initialization of solver variables. Here we give the first formal treatment of mode checking in strongly typed CLP languages which require reordering of clause body literals during mode checking. We also sketch the mode checking algorithms used in the HAL compiler.

1 Introduction

While traditional logic and constraint logic programming (CLP) languages are untyped and unmoded, recent languages such as Mercury [14] and HAL [5] require type, mode and determinism declarations for (exported) predicates and functions. This information allows the generation of efficient target code (e.g. mode information provides a substantial speed improvement [4]), improves robustness and facilitates efficient integration with foreign language procedures. Here we describe our experience with mode checking in the HAL compiler.

HAL is a CLP language designed to facilitate "plug-and-play" experimentation with different solvers. To achieve this it provides support for user-defined constraint solvers, global variables and dynamic scheduling. Mode checking in HAL is one of the most complex stages in compilation. It requires the compiler to appropriately re-order literals in the body of each rule. Since predicates can be given multiple mode declarations, mode checking is performed for each of these modes and the compiler creates a specialized *procedure* for each mode (i.e. performs multi-variant specialization). Three issues make mode checking even more difficult. First, instantiations (which describe the possible states of program variables) may be very complex and interact with the type declarations. Second, accurate mode checking of higher order functions and predicates is difficult. Third, the compiler needs to handle automatic initialization of solver variables.

The contribution of this paper is two-fold. The first is to formalize mode checking in the context of CLP languages which are strongly typed and which may need reordering of clause body literals during mode checking. In order to do this we introduce "ti-trees", which are a kind of labelled deterministic regular tree. The second contribution is to describe the algorithms for mode checking currently employed in the HAL compiler. Since HAL and the logic programming language Mercury share similar type and mode systems,[1] much of our description and formalization also applies to mode checking in Mercury (which has not been previously described). However, there are significant differences between mode checking in the two languages. In HAL there is the need to handle automatic initialization of solver variables. Furthermore, determining the best reordering in HAL is more complex than in Mercury because the order in which constraints are solved can have a significant impact on efficiency [10]. Also, HAL handles a limited form of polymorphic mode checking. On the other hand, Mercury's mode system allows the specification of additional information about data structure liveness and usage.

Mode inference and checking of logic programs has been a fertile research field for many years. However, starting with [12, 3], almost all research has focused on mode checking/inference in traditional logic programming languages where the analysis assumes the given literal ordering is correct, only simple instantiations are used and higher-order predicates are largely ignored. Regular trees have been used before in logic programming to define types, e.g. [7], and instantiations, e.g. [9], usually in the context of inference of information. Here, although we use regular trees to formalize types, our type analysis [6] is based on a Hindley-Milner approach. A key difference with previous work (in particular [9]) is that we describe instantiations for polymorphic types, including higher-order objects. The only work on mode checking in strongly typed logic languages with reorderable clause bodies is that of [13]. Other related work has been on mode checking for concurrent logic programming languages [2]. There the emphasis has been on detecting communication patterns and possible deadlocks.

2 The HAL Language

In this section we provide an overview of the HAL language. The basic syntax follows the standard CLP syntax, with variables, rules and predicates defined as usual (see, for example, [11] for an introduction to CLP). HAL supports integer, float, string and char data types and terms over these types. However, the base language support is limited to assignment, testing for equality, and construction and deconstruction of ground data. More sophisticated constraint solving requires the programmer to import a constraint solver for the type. In the case of terms, the declaration `:- herbrand` f/n. indicates that the system should use a Herbrand solver for terms of type $f(T_1, \ldots, T_n)$. Types with an associated constraint solver are called *solver types*.

[1] In part, because HAL is compiled to Mercury

Programmers may annotate predicate definitions with type, mode and determinism declarations. Consider the following HAL program implementing a polymorphic stack using lists:

```
:- typedef list(T) ->
           ([] ; [T|list(T)]).
:- instdef elist -> [].
:- instdef list(I) ->
           ([]; [I|list(I)]).
:- instdef nelist ->
           [ground|list(ground)].
:- modedef out(I) -> (new -> I).
:- modedef in(I) -> (I -> I).
```

```
:- pred push(list(T),T,list(T)).
:- mode push(in,in,out(nelist) is det.
push(S0,E,S1) :- S1 = [E|S0].
:- pred pop(list(T),T,list(T)).
:- mode pop(in,out,out) is semidet.
:- mode pop(in(nelist),out,out) is det.
pop(S0,E,S1) :- S0 = [E|S1].
:- pred empty(list(T)).
:- mode empty(in) is semidet.
:- mode empty(out(elist)) is det.
empty(S) :- S = [].
```

Types are used to specify the representation format of program variables. For example, the type system in HAL distinguishes between constrained integers (`cint`) and standard numerical integers (`int`) since these have a different representation. Type definitions are (polymorphic) regular tree type statements. In the example above, the first line on the left defines a (parametric) list type. Equivalence types are also allowed. They are treated as macros for type expressions, and the compiler replaces equivalence types by their definition.

Instantiations are used to specify the possible values, within a type, that a program variable may have. The *base* instantiations for a variable are `new`, `old` and `ground`. A variable is `new` if it has not been seen by any constraint solver, `old` if it has, and `ground` if it is known to take a fixed value. For data structures such as trees or lists of solver variables, more complex instantiation states may be used. In our example, the next three lines on the left are instantiation definitions. The instantiation `elist` describes empty lists. The parametric instantiation `list(I)` describes lists whose elements have instantiation `I`. Note the deliberate reuse of the type name. The instantiation `nelist` describes non-empty ground lists.

A mode is of the form $Inst_1 \to Inst_2$ where $Inst_1$ describes the *call* (or input) instantiation and $Inst_2$ describes the *success* (or output) instantiation. The standard modes are mappings from one base instantiation to another: we use two letter codes (`oo`, `no`, `og`, `gg`=`in`, `ng`=`out`) based on the first letter of the instantiation, e.g. `ng` is `new`→`ground`. Every constraint solver is required to provide an initialization predicate, `init/1`, which has mode `no`. In our example, the fifth and sixth lines on the left are mode definitions, defining macros for modes. The `out(I)` mode requires a new object on call and returns an object with instantiation `I`. The `in(I)` mode requires instantiation `I` on call and has the same instantiation on success.

The first three lines on the right define the predicate `push/3`. The first line is a type declaration (polymorphic in element type `T`). The second is a mode declaration specifying that the first two arguments must be ground on call, the third returns a non-empty ground list, and there is always one solution (determinism `det`). Determinism declarations describe how many answers a predicate may have: `nondet` means any number of solutions; `multi` at least one solution; `semidet` at most one solution; `det` exactly one solution; `failure` no solutions; and `erroneous` a runtime error.

The next four lines similarly define `pop/3`. Note that it has two modes of usage: in the first mode `S0` might be an empty list and thus the `pop` might fail. In the second mode, `S0` is known to be a non-empty list and thus the `pop` always succeeds. The last four lines define `empty/1`, which also has two modes of usage: the first tests if a given list is empty, and the second returns a new empty list.

3 Type, Instantiation and Type-Instantiation Trees

In this section we formalise type and instantiation definitions in terms of deterministic regular trees. Then we introduce type-instantiation (ti-) trees which combine type and instantiation information and are the basis for mode checking in HAL.

Regular Trees Regular trees are a well understood formalism (see, for example, [8]) but algorithms for them are surprisingly hard to find in the literature: [9] gives algorithms for ordering ($\preceq$) and lower bound ($\sqcap$), while [1] give algorithms for (polymorphic) non-deterministic regular trees.

A signature Σ is a set of pairs f/n where f is a symbol and $n \geq 0$ is the integer arity of f. Let $\tau(\Sigma)$ denote the set of all ground terms (the Herbrand Universe) over Σ, We assume (for simplicity) that Σ contains at least one constant (i.e. arity 0) symbol.

A *(deterministic) regular tree* r over some signature Σ is a rooted directed graph with the following properties:

1. Each node a has a label denoted $label(a)$ and has $deg(a)$ outgoing edges labelled $1, \ldots, deg(a)$.
2. There are two classes of nodes: *functor nodes* and *set nodes*. Consider a node a with $label(a) = f$ and $deg(a) = n$. If a is a functor node then $f/n \in \Sigma$ and each outgoing edge ends at a set node. If a is a set node then $f \in Set$, all outgoing edges end at functor nodes, and these functor nodes refer to distinct function symbols, i.e. for each two children a_i and a_j, either $label(a_i) \neq label(a_j)$ or $deg(a_i) \neq deg(a_j)$. For standard regular trees $Set = \{OR\}$.
3. The root node is a set node.
4. Each node is reachable from the root node.

Note that a consequence of the definition is that regular trees are bipartite: set nodes alternate with functor nodes down the tree.

We use paths (sequences of integers) to refer to nodes in a regular tree: If r is a regular tree, $r.\epsilon$ refers to the root of r, while if $r.p$ refers to node a, then $r.p.i$ refers to the node reached from a by following the edge labelled i.

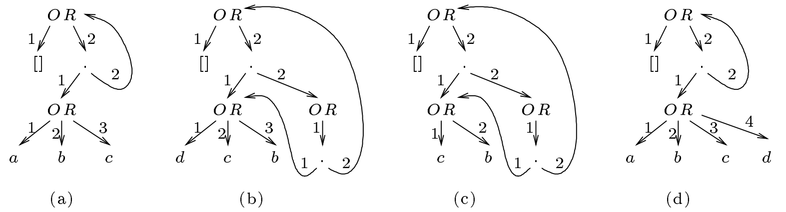

Fig. 1. Regular trees for lists of as, bs, cs and ds and their meet and join.

Each path p in a regular tree r defines a subset $[\![r.p]\!]$ of $\tau(\Sigma)$. This is the least set satisfying:

$$[\![r.p]\!] = \begin{cases} \bigcup\{[\![r.p.i]\!] \mid 1 \leq i \leq deg(r.p)\}, & \text{if } label(r.p) = OR \\ \{f(t_1, \ldots, t_n) \mid f = label(r.p),\ n = deg(r.p), & \\ \qquad\qquad \text{and } t_i \in [\![r.p.i]\!] \text{ for } 1 \leq i \leq n\}, & \text{otherwise.} \end{cases}$$

We extend this notation to a regular tree, r, by defining $[\![r]\!] = [\![r.\epsilon]\!]$. This is the *meaning* of the tree.

Example 1. Consider the signature $\{[]/0, `.'/2, a/0, b/0, c/0, d/0\}$ and the associated regular trees r_{1a} and r_{1b} shown in Figure 1(a) and (b), respectively. The tree r_{1a} defines lists of as, bs and cs. The notation $r_{1a}.2.2.2.1.2$ refers to the node labelled b. The tree whose root is $r_{1a}.2.1$ defines the set of terms $\{a, b, c\}$, while r_{1b} defines even length lists of bs, cs and ds.

The $[\![\cdot]\!]$ function induces a partial order on regular trees: $r_1 \preceq r_2$ iff $[\![r_1]\!] \subseteq [\![r_2]\!]$. With the addition of $\bot$, the least regular tree, and $\top$, the greatest regular tree, the partial order gives rise to a lattice over the regular trees. The tree $\bot$ will arise when modes can determine that execution fails, and $\top$ will arise when computing the upper bound of two incomparable regular trees.

We use $\sqcap$ to denote the meet (i.e. greatest lower bound) operator in this lattice, and $\sqcup$ to denote the join (i.e. least upper bound) operator. We have that $[\![r_1 \sqcap r_2]\!] = [\![r_1]\!] \cap [\![r_2]\!]$. Because we restrict ourselves to deterministic regular trees the join is inexact: $[\![r_1 \sqcup r_2]\!] \supseteq [\![r_1]\!] \cup [\![r_2]\!]$.

Example 2. Consider the regular trees r_{1a} and r_{1b} illustrated in Figure 1. Their meet $r_{1a} \sqcap r_{1b}$ is shown in Figure 1(c). Their join $r_{1a} \sqcup r_{1b}$ is shown in Figure 1(d).

Type Trees Types in HAL (as in most CLP languages) are naturally formalized using (possibly polymorphic) deterministic regular trees. We assume a fixed signature Σ_{term} of term constructors.

A *type constructor* f is a functor of some arity. A *type expression* (or simply *type*) is either a type variable v or a term of the form $f(t_1, \ldots, t_n)$ where f is an n-ary type constructor, and $t_1, \ldots, t_n$ are type expressions. A *type definition* for f is of the form

`:- typedef` $f(v_1, \ldots, v_n)$ `->` $(f_1(t^1_1, \ldots, t^1_{m_1}); \cdots; f_k(t^k_1, \ldots, t^k_{m_k}))$.

where $v_1, \ldots, v_n$ are distinct type variables, $\{f_1/m_1, \ldots, f_k/m_k\} \subseteq \Sigma_{term}$ are distinct term constructor/arity pairs, and $t_1^1, \ldots, t_{m_k}^k$ are type expressions involving at most variables $v_1, \ldots, v_n$.

Every type t has a corresponding regular tree r defined as follows: If t is a type variable v then r is a singleton set node with label v and with no children. We thus need to add to Set an infinite number of type variables. Conceptually, a type variable is simply a place holder which can be substituted with a type expression. Otherwise, t is of the form $f(e_1, \ldots, e_n)$, where f is defined by a type definition of the form above. Let θ be the substitution $\{v_1 \mapsto e_1, \ldots, v_n \mapsto e_n\}$. Then r has as root a OR labelled node a with k functor nodes as children. The j^{th} child of a is labelled f_j and has m_j children. The l^{th} child of $r.j$ is the tree corresponding to the type $\theta(t_l^j)$.

A type can be understood in a "naive" set-theoretic manner as the meaning of its associated regular tree. From now on we will not distinguish between types and their corresponding regular trees. For example, we will use the notation $[\![t]\!]$ on a variable-free type t to refer to the set of terms defined by its corresponding regular tree. For simplicity, in this paper we ignore the built-in types `float`, `int`, `char` and `string` whose treatment does not significantly complicate mode checking.

Example 3. Given the type definition: `:- typedef abc -> a ; b ; c.` the corresponding regular tree to type `list(abc)` is shown in Figure 1(a). The meaning, $[\![$`list(abc)`$]\!]$, is the set of lists of `a`s, `b`s and `c`s. The regular tree corresponding to the type expression `list(T)` is shown in Figure 2(a).

Instantiation Trees Instantiation definitions look very much like type definitions, the only difference being that the arguments themselves are instantiations rather than types. However, they should not be confused: a type describes the representation format for a variable and is therefore invariant over the life of the variable, while an instantiation describes the set of values a variable can have at a particular program point. More exactly, an instantiation describes at a particular point in execution how constrained a variable is and what values it may have been bound to.

An *instantiation constructor* g is a functor of some arity. An *instantiation expression* (or simply *instantiation*) is either a base instantiation (one of `ground`, `old` or `new`), an instantiation variable w, or a term of the form $g(i_1, \ldots, i_n)$ where g is an n-ary instantiation constructor, and $i_1, \ldots, i_n$ are instantiation expressions. A *instantiation definition* for g is of the form:

`:- instdef` $g(w_1, \ldots, w_n)$ `->` $(g_1(i_1^1, \ldots, i_{m_1}^1); \cdots ; g_k(i_1^k, \ldots, i_{m_k}^k))$.

where $w_1, \ldots, w_n$ are distinct instantiation variables, $\{g_1/m_1, \ldots, g_k/m_k\} \subseteq \Sigma_{term}$ are distinct term constructors, and $i_1^1, \ldots, i_{m_k}^k$ are instantiation expressions other than `new`,[2] involving at most the variables $w_1, \ldots, w_n$.

[2] HAL (and Mercury) do not allow uninitialized (`new`) data to be included in data structures.

Arbitrary polymorphic modes are not well-understood. For this reason, HAL requires instantiations appearing in a predicate mode declaration to be variable-free. As a result, mode checking only deals with variable-free instantiations. Thus, from now on we will assume all instantiations are variable free. We can associate a slightly extended form of regular tree with a variable-free instantiation, analogously to how we associate a regular tree to a type. The only differences are that there are no nodes labelled by instantiation variables and we require new set node labels $\{\texttt{new}, \texttt{old}, \texttt{ground}\} \in Set$ to express the base instantiations. Each of these set nodes has no outgoing arcs, and any `new` node must be the only node in its regular tree.

Type-Instantiation Trees Information about the type of a variable can be combined with an instantiation for that variable to give even more detailed information about the possible values the variable can take at a particular program point. To do this, we define a function $rt(t, i)$ from a type expression t and a variable-free instantiation expression i to a *type-instantiation regular tree* (or *ti-tree*).

The base instantiation `ground` represents all elements of the type but indicates that the program variable is bound to a unique value. Hence, if t is a variable-free type, $rt(t, \texttt{ground})$ is simply t. Otherwise, $rt(t, \texttt{ground})$ is obtained from t by replacing each node labelled by a type variable v by a node labelled $\texttt{ground}(v)$ with no children. Conceptually, this new node represents the tree $rt(t', \texttt{ground})$ obtained if v were replaced by the variable-free type t'.

The base instantiation `old` represents all elements of the type, including the possibility that the program variable may still not have a unique value for those parts of the type which are solver types (i.e. have an associated solver). The regular tree $rt(t, \texttt{old})$ is obtained from t by (a) adding a new child labelled $old_{t'}$ to any OR node corresponding to a solver type t', and (b) replacing the nodes labelled with a type variable v by a node labelled $\texttt{old}(v)$ with no children.

Example 4. Suppose that list types are solver types but the type `abc` is not. Then $rt(\texttt{list(abc)}, \texttt{old})$ $(= \mathit{olabc1})$ is the regular tree shown in Figure 1(a), but with an extra (third) child of the root labelled $old_{\texttt{list(abc)}}$. The set $[\![\mathit{olabc1}]\!]$ includes terms $\{[], [a|old_{\texttt{list(abc)}}], [b], [b, a, c, a|old_{\texttt{list(abc)}}]\}$. The symbol $old_{\texttt{list(abc)}}$ represents possible positions of `list(abc)` variables.

Suppose `abc` is a solver type but list types are not, then $rt(\texttt{list(abc)}, \texttt{old})$ $(= \mathit{olabc2})$ is again the tree shown in Figure 1 (a), but with a new (fourth) child of the non-root OR node labelled $old_{\texttt{abc}}$. It is shown in Figure 2(d). The set $[\![\mathit{olabc2}]\!]$ includes terms $\{[], [a], [old_{\texttt{abc}}, b, old_{\texttt{abc}}]\}$. Note that the two occurrences of the symbol $old_{\texttt{abc}}$ do not necessarily represent the same variable.

The base instantiation `new` cannot exist as part of another instantiation. Thus, the regular tree $rt(t, \texttt{new})$ is a singleton node with no incoming or outgoing edges, labelled new_t. Note that this is true regardless of whether t is a variable or not.

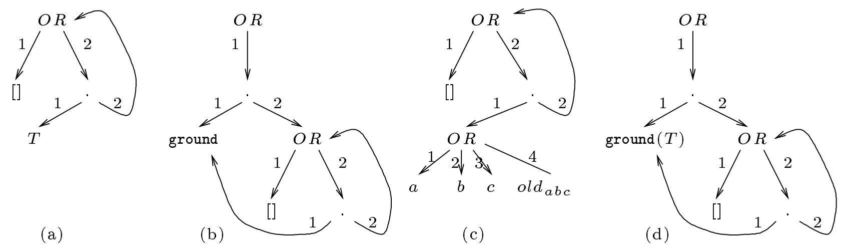

Fig. 2. Trees t=list(T), nelist, rt(list(abc), old) and rt(t, nelist).

Finally, the operation $rt(t, i)$ where i is a non-base instantiation is defined analogously to the $\sqcap$ operation. The only changes are those needed to handle the base instantiations which is described above.

A ti-tree is thus a regular tree where $\Sigma = \Sigma_{term} \cup \{old_t \mid t \text{ is a type expression}\}$ and $Set = \{OR\} \cup \{\texttt{ground}(v), \texttt{old}(v) \mid v \text{ is a type variable}\} \cup \{new_t \mid t \text{ is a type expression}\}$. We extend the partial ordering and hence meet and join on regular trees to ti-trees by taking into account the "is more constrained" partial order on the base instantiations. The extension is as follows. For the case of the singleton new_t: $new_t \preceq r$ iff $r = new_t$; $new_t \sqcup new_t = new_t$; $new_t \sqcup r = \top$ when $r \neq new_t$ (no representable upper bound); $new_t \sqcap r = r$ for any r[3] (usually representing when a variable is first instantiated to r). Let us now consider $\texttt{ground}(v)$ and $\texttt{old}(v)$. We assume mode checking occurs after type analysis and, thus, we know the code is type-correct and we can assume we have the most specific type description for each program variable. As a result, during mode checking we will only compare nodes containing the same type variable v. For this reason, we only need note that $\texttt{ground}(v) \preceq \texttt{old}(v)$, and the meet and join operations follow in the natural way.

Example 5. Assume abc is a solver type and list types are not. The regular trees in Figure 2(a), (b), (c) and (d) correspond to the type list(T), the instantiation nelist, and the the ti-trees rt(list(abc), old) and rt(list(T), nelist), respectively.

Finally, for the checking algorithm we introduce the concept of a *type-instantiation state* (or *ti-state*) $\{x_1 \mapsto r_1, \ldots, x_n \mapsto r_n\}$, which maps program variables to ti-trees. We can extend operations on ti-trees to ti-states over the same set of variables in the obvious manner. Given ti-states $TI = \{x_1 \mapsto r_1, \ldots, x_n \mapsto r_n\}$ and $TI' = \{x_1 \mapsto r'_1, \ldots, x_n \mapsto r'_n\}$ then $TI \preceq TI'$ iff $r_l \preceq r'_l$ for all $1 \leq l \leq n$, $TI \sqcap TI' = \{x_l \mapsto r_l \sqcap r'_l \mid 1 \leq l \leq n\}$ and $TI \sqcup TI' = \{x_l \mapsto r_l \sqcup r'_l \mid 1 \leq l \leq n\}$.

[3] This interpretation of $\sqcap$ does not agree with the ordering, instead it gives the correct answer when we use $\sqcap$ to create new instantiations during mode checking.

4 Basic Mode Checking

Mode checking is a complex process which aims to reorder body literals in order to satisfy the mode constraints provided by each mode declaration, generating different code for each mode declaration.

Before mode checking the HAL compiler has normalized the program (i.e. rewritten it to a form where each atom has distinct variables as arguments and each equality is either of the form $x = y$ or $x = f(x_1, \ldots, x_n)$, where $x, y, x_1, \ldots, x_n$ are distinct variables) and performed type checking (and inference) so each program variable has a type.

We shall now explain mode checking by showing how to check whether each program construct is schedulable for a given ti-state and, if so, what the resulting ti-state is. If the program construct is not schedulable for the given ti-state it may be reconsidered after other constructs are scheduled. We assume that before checking each construct for initial ti-state TI, we extend TI so that any variable of type t local to the construct is assigned the ti-tree new_t.

Equality Consider the equality $x_1 = x_2$ where x_1 and x_2 are variables of type t and the current ti-state is $TI = \{x_1 \mapsto r_1, x_2 \mapsto r_2\} \cup RTI$ (where RTI is the ti-state for the remaining variables). The two standard modes of usage for such an equality are **copy** (`:=`) and **unify** (`==`). If exactly one of r_1 and r_2 is new_t (say r_1), **copy** $x_1 := x_2$ can be performed and the resulting ti-state is $TI' = \{x_1 \mapsto r_2, x_2 \mapsto r_2\} \cup RTI$. If both are not new_t then **unify** $x_1 == x_2$ is performed and the resulting instantiation is $TI' = \{x_1 \mapsto r_1 \sqcap r_2, x_2 \mapsto r_1 \sqcap r_2\} \cup RTI$. If neither of the two modes of usage apply, the literal is not schedulable (yet).

Consider the equality $x = f(x_1, \ldots, x_n)$ where $x, x_1, \ldots, x_n$ are variables with types $\{x \mapsto t, x_1 \mapsto t_1, \ldots, t_n \mapsto t_n\}$ and current ti-state $TI = \{x \mapsto r, x_1 \mapsto r_1, \ldots, x_n \mapsto r_n\} \cup RTI$. The two standard modes of usage of such an equality are: **construct** (`:=`) and **deconstruct** (`=:`). The **construct** mode applies if r is new_t and none of the r_j are new_{t_j}. The resulting ti-state is $TI' = \{x \mapsto r', x_1 \mapsto r_1, \ldots, x_n \mapsto r_n\} \cup RTI$ where r' is the ti-tree defined by $OR \rightarrow_1 f(r_1, \ldots, r_n)$. The **deconstruct** mode applies if each r_j is new_{t_j} and r is not new_t and has no child old_t (which means it is definitely bound to some functor). If r has a child tree of the form $f(r'_1, \ldots, r'_n)$ the resulting ti-state is $TI' = \{x \mapsto r, x_1 \mapsto r'_1, \ldots x_n \mapsto r'_n\} \cup RTI$. If r has no child tree of this form, then the resulting ti-state is the same but with $r'_j = \bot, 1 \leq j \leq n$, indicating that the **deconstruct** must fail. If some of the variables x_j are `new` (i.e. $r_j = new_{t_j}$) and some are not (say $x_{k_1}, \ldots, x_{k_m}$) the mode checking process decomposes the equality constraint into a **deconstruct** followed by new equalities by introducing fresh variables, e.g. $x = f(x_1, \ldots, \mathit{fresh}_{k_j}, \ldots), \ldots, x_{k_j} = \mathit{fresh}_{k_j}, \ldots$. These new equalities are handled as above.

Example 6. Assume X and Y are ground lists and A is `new`. Scheduling the goal `Y = [A|X]` results in the code `Y =: [A|F], X == F`.

The above uses of **deconstruct** are guaranteed to be safe at runtime. One difference between HAL and Mercury is that HAL allows the use of the **deconstruct** mode when x is `old` (i.e. $r = rt(t, \texttt{old})$). In this case r has a child node of the form $f(r'_1, \ldots, r'_n)$ and we proceed as in the previous paragraph. Note that this is where the HAL mode system is weak (i.e. run-time mode errors can occur), since if the at run-time x is a variable the **deconstruct** will abort if it cannot initialize all of $x_1, \ldots, x_n$.

Note that if $r = new_t$ and some $r_i = new_{t_i}$ then the literal is not schedulable.

Predicates Consider the predicate call $p(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are variables with type $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. Assume p has the mode declaration $p(c_1 \to s_1, \ldots, c_n \to s_n)$ where c_j, s_j are the call and success instantiations, respectively, for argument j, and the current ti-state is $TI = \{x_1 \mapsto r_1, \ldots, x_n \mapsto r_n\} \cup RTI$.

The predicate call can be scheduled if for each $j \in \{1..n\}$ the current ti-state is stronger than (defines a subset of) the calling ti-state required for p, i.e. $r_j \preceq rt(t_j, c_j)$. If the predicate is schedulable for this mode the new ti-state is $TI' = \{x_1 \mapsto r_1 \sqcap rt(t_j, s_1), \ldots, x_n \mapsto r_n \sqcap rt(t_n, s_n)\} \cup RTI$. The predicate call can also be scheduled if for each j such that $r_j \not\preceq rt(t_j, c_j)$ then $rt(t_j, c_j) = new_{t_j}$. For each such j, the argument x_j in predicate call $p(x_1, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots, x_n)$ is replaced by $fresh_j$, where $fresh_j$ is a fresh new program variable, and the equation $fresh_j = x_j$ is added after the predicate call.

If multiple modes of the same predicate are schedulable, we choose mode with calling ti-state TI (defined as $\{x_1 \mapsto rt(t_1, c_1), \ldots, x_n \mapsto rt(t_n, c_n)\}$) such that, for each other schedulable calling ti-state TI', $TI' \not\preceq TI$.

Conjunctions, Disjunctions and If-Then-Elses To determine if a conjunction $G_1, \ldots, G_n$ is schedulable for initial ti-state TI we choose the left-most goal G_j which is schedulable for TI and compute the new ti-state TI_j. This default behaviour schedules goals as close to the user-defined left-to-right order as possible. If the state TI_j assigns $\perp$ to any variable, then the subgoal G_j must fail and hence the whole conjunction is schedulable. The resulting ti-state TI' maps all variables to $\perp$, and the final conjunction contains all previously scheduled goals followed by `fail`. If TI_j does not assign any variable to $\perp$ we continue by scheduling the remaining conjunction $G_1, \ldots, G_{j-1}, G_{j+1}, \ldots, G_n$ with initial ti-state TI_j. If all subgoals are eventually schedulable we have determined both an order of evaluation for the conjunction and a final ti-state.

To determine if a disjunction $G_1; \cdots; G_n$ is schedulable for initial ti-state TI we check whether each subgoal G_j is schedulable for TI and, if so, compute each resulting ti-state TI_j, obtaining the final ti-state $TI' = \bigsqcup_{j \in \{1..n\}} TI_j$. If this ti-state assigns $\top$ to any variable or one of the disjuncts G_j is not schedulable then the whole disjunction is not schedulable.

To determine whether an if-then-else $G_i \to G_t; G_e$ is schedulable for initial ti-state TI, we determine first whether G_i is schedulable for TI with resulting ti-state TI_i. If not the whole if-then-else is not schedulable. Otherwise, we try to schedule G_t in state TI_i (resulting in state TI_t say) and G_e in state TI (resulting in state TI_e say). The resulting ti-state is $TI' = TI_e \bigsqcup TI_t$. If one of G_t or G_e is not schedulable or TI' includes $\top$ the whole if-then-else is not schedulable.

Mode Declarations To check that a predicate with head $p(x_1, \ldots, x_n)$ and declared (or inferred) type $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ satisfies the mode declaration $p(c_1 \to s_1, \ldots, c_n \to s_n)$, we build an initial ti-state $TI = \{x_1 \mapsto rt(t_1, c_1), \ldots, x_n \mapsto rt(t_n, c_n)\}$. The body of the predicate (multiple rules are simply treated as a disjunction) is then analysed using TI. The mode declaration is correct if everything is schedulable with final ti-state $TI' = \{x_1 \mapsto s'_1, \ldots, x_n \mapsto s'_n\}$ and for each argument variable $1 \leq i \leq n$, $s'_i \preceq rt(t_i, s_i)$. If the body is not schedulable[4] or the resulting instantiations are not strong enough a mode error results.

Example 7. Consider mode checking of the following code:

```
:- pred dupl(list(T), list(T)). %% duplicate top of stack
:- mode dupl(in(nelist), out(nelist)) is det.
dupl(S0, S) :- S0 = [], S = [].
dupl(S0, S) :- pop(S0, A, S1), push(S0, A, S).
```

We start by constructing the initial ti-state $TI = \{\texttt{S0} \mapsto r_{2f}, \texttt{S} \mapsto new_{list(T)}\}$ where $r_{2d} = rt(\texttt{list}(T), \texttt{nelist})$ is the tree shown in Figure 2(d). Checking the first disjunct (rule) we have `S0 = []` schedulable as a deconstruct. The resulting ti-state assigns $\perp$ to `S0`, thus the whole conjunction is schedulable with $TI_1 = \{\texttt{S0} \mapsto \perp, \texttt{S} \mapsto \perp\}$. Checking the second disjunct, we first extend TI to map `A` to new_T and `S1` to $new_{list(T)}$. Examining the first literal `pop(S0, A, S1)` we find that both modes declared for `pop/3` are schedulable. Since the second mode has more specific calling instantiations, it is chosen and the ti-trees for `A` and `S1` become $\texttt{ground}(T)$ and $rt(\texttt{list}(T), \texttt{ground})$, respectively. Now the second literal is schedulable obtaining for `S` the ti-tree r_{2d}. Restricting to the original variables the final ti-state is $TI_2 = \{\texttt{S0} \mapsto r_{2d}, \texttt{S} \mapsto r_{2d}\}$. Taking the join $TI' = TI_1 \sqcup TI_2 = TI_2$. Checking this against the declared success instantiations we find the declared mode is correct. The code generated for the procedure is:

```
dupl_mode1(S0, S) :- fail.
dupl_mode1(S0, S) :- pop_mode2(S0, A, S1), push_mode1(S0, A, S).
```

where `pop_mode2/3` and `push_mode1/3` are the procedures associated to the second and first modes of the predicates, respectively. The first rule will be removed during code generation.

Note that the mode checking algorithm does not track variable dependencies and thus it might obtain a final type-instantiation state weaker than expected. This can be overcome by adding a definite sharing analysis and/or a dependency based groundness analysis to the mode checking phase. In practice, however, it seems these kinds of modes are rarely used.

[4] In this case the compiler reports an error about the least subpart of the goal which is not schedulable.

5 Automatic Initialization

Many constraint solvers require solver variables to be initialialized before they can be used in constraints. Thus, explicit initializations for local variables may need to be introduced. This is not only a tedious exercise for the user, it may even be impossible for multi-moded predicate definitions since each mode may require different initialization instructions. Therefore, the HAL mode checker automatically inserts variable initializations when required. Hence, whenever a literal cannot be scheduled because there is a requirement for an argument of type t to be $rt(t, \texttt{old})$ when it is new_t and t is a solver type, then the `init/1` predicate for type t can be inserted to make the literal schedulable.

Unfortunately, unnecessary initialization may slow execution and introduce unnecessary variables (when it interacts with implied modes). Hence, we would only like to add those initializations required so that mode checking will succeed. The HAL mode checker implements this by first trying to mode the procedure without allowing initialization, and then if this fails starting from the previous partial schedule it tries to find the leftmost unscheduled literal which can be scheduled with initialization. If such a literal is found the appropriate initialization calls are added, the literal is scheduled and then scheduling continues once more trying to schedule without initialization. If there is no literal schedulable with initialization the whole conjunct is not schedulable. This two phase approach is applied at every conjunct level individually.

Example 8. Consider the following program where `cint` is a solver type:

```
:- pred length(list(cint),int).
:- mode length(out(list(old)),in) is semidet.
length(L,N) :- N = 0, L = [].
length(L,N) :- N > 0, N = N1 + 1, L = [V|L1], length(L1,N1).
```

In the first phase the second rule is not schedulable since `L = [V|L1]` cannot be a **construct** (V is new) or a **deconstruct** (L is new). or In the second phase we try to schedule the remaining unscheduled literal, which can be managed by initializing V, obtaining:

```
length(L,N) :- NN := 0, NN == N, L := [].
length(L,N) :- N > 0, N1 := N - 1, length(L1,N1), init(V), L := [V|L1].
```

The initialization policy above is not always the best possible and we are investigating more informed policies which give the right tradeoff between adding constraints as early as possible, and delaying constraints until they can be tests or assignments.

6 Higher-Order Terms

Higher-order programming is particularly important in HAL because it is the mechanism used to implement dynamic scheduling, which is vital in CLP languages for extending and combining constraint solvers. Higher-order programming introduces two new kinds of literals: construction of higher-order objects and higher-order calls. A higher-order object is constructed using an equation of the form $h = p(x_1, \ldots, x_k)$ where $h, x_1, \ldots, x_k$ are variables and p is an n-ary predicate with $n \geq k$. The variable h is referred to as a higher-order object. Higher-order calls are literals of the form call$(h, x_{k+1}, \ldots, x_n)$ where $h, x_{k+1}, \ldots, x_n$ are variables. Essentially, the `call/n` literal supplies the $n-k$ arguments missing from the higher-order object h.

The *higher-order type* of a higher-order object h is of the form $pred(t_{k+1}, \ldots, t_n)$ where $pred/m$ is a special type construct and $t_{k+1}, \ldots, t_n$ are types. It provides the types of the $n-k$ arguments missing from h. The higher-order instantiation of h is of the form $pred(c_{k+1} \rightarrow s_{k+1}, \ldots, c_n \rightarrow s_n)$[5] where $pred/m$ is a special instantiation construct and $c_j \rightarrow s_j$ are call and success instantiations. It provides the modes of the $n-k$ arguments missing from h. Because of space limititations we omit the formal definitions of ti-trees for higher-order objects.

Example 9. Consider the following code:

```
:- pred map(pred(A,B), list(A), list(B)).
:- mode map(in(pred(in,out) is det), in, out) is det.
map(H, [], []).
map(H, [A|As], [B|Bs]) :- call(H,A,B), map(H,As,Bs).
:- pred code(char,int,char).
:- mode code(in,in,out) is det.
code(C1,I,C2) :- C2 = ord(chr(C1) + I).

?- H = code('a'), map(H,[7,0,11],S).
```

The `map/3` predicate takes a higher-order predicate with two missing arguments with types A and B and modes `in` and `out`, respectively. This predicate is applied to a list of As, returning a list of Bs. The literal `H = code('a')` builds a higher-order object which calls the `code` predicate with first argument `'a'`. The type-instantiation of $H2$ `pred(int::in,char::out)` is represented by the regular tree shown in Figure 3(b).

HAL treats the mode and determinism of higher order terms as if they were part of the "type". This means that, for example, in a list of higher-order objects, all elements must have the same mode and determinism. This makes sense since otherwise after removing an element from the list it cannot be called (since we have lost its mode and determinism information).[6] This treatment simplifies mode checking for higher-order ti-trees since the only comparable ti-trees with root labelled *pred* are identical.

Intuitively, a higher-order equation $h = p(x_1, \ldots, x_k)$ is schedulable if h is `new` and $x_1, \ldots, x_k$ are at least as instantiated as the call instantiations of one of the modes declared for p/n. If this is true for more than one mode, an ambiguity is reported. If it is not true for any mode, the equation is delayed until the arguments become more instantiated. Note that the instantiation of the x_js is unchanged, and in fact will not be updated when the call to h is made.

[5] In practice, the determinism also appears in the higher-order instantiation; for simplicity we ignore it here.

[6] Mercury treats this differently. Two higher-order objects with different mode and/or determinism information can be placed in the same list, but an error will occur when calling an element removed from this list.

Hence, higher-order objects lose precision of mode information. This would lead to erroneous mode information if one of the x_j is `new` and hence the call is not schedulable if this is the case. The HAL compiler warns if the declared mode of p used in the higher-order call may lost instantiation information.

A higher-order call $\texttt{call}(h, x_{k+1}, \ldots, x_n)$ is schedulable if $x_{k+1}, \ldots, x_n$ are at least as instantiated as the call instantiations of the arguments of the higher-order type-instantiation previously assigned to h. If this is not true, the call is delayed until the arguments become more instantiated. Just as for normal predicate calls, *implied modes* are also possible where if, h requires one of the x_l to be `new` and it isnt we can replace it in the `call` literal by a fresh variable $fresh_l$ and a following equation $fresh_l = x_l$.

7 Polymorphism and Modes

Consider the interface to the stack data type defined in Section 2 for storing polymorphic objects. This interface loses substantial information about the instantiations of retrieved items (they are only known to be ground). The problem is more severe for higher-order objects because the base instantiation `ground` applied to polymorphic code does not contain enough information for the higher-order object to be used (called).

Example 10. Consider the following goal:

```
?- empty(S0), I0 = code('a'), push(S0,I0,S1), pop(S1,I,S2), map(I,[7],S).
```

When item I is extracted from the list its type instantiation information is a `ground` object of type `pred(int,char)`. Since the mode and determinism information of I is lost, it cannot be used in `map` and a mode error results.

We could overcome the above problem by having a special version of each predicate for the case of higher-order predicates. But this defeats the idea of the abstract data type. Our approach is to use polymorphic type information to recover the lost mode information. This is an example of "Theorems for Free" [15]: since the polymorphic code can only copy[7] terms with polymorphic type it cannot create instantiations and, thus, the output instantiations of polymorphic arguments must result from the calling instantiations of non-output arguments. Hence, they have to be at least as instantiated as the join of the input instantiations.

To recover instantiation information we extend mode checking for procedures with polymorphic types to take into account the extra mode information that is implied by the polymorphic type. Due to space considerations we simply illustrate this by an example.

Example 11. Assume we are scheduling the `push/3` literal in the goal:

```
?- empty(S0), I0 = code('a'), push(S0,I0,S1), pop(S1,I,S2), map(I,[7],S).
```

[7] In HAL and Mercury it can also unify such terms as well, but this only creates more instantiated modes.

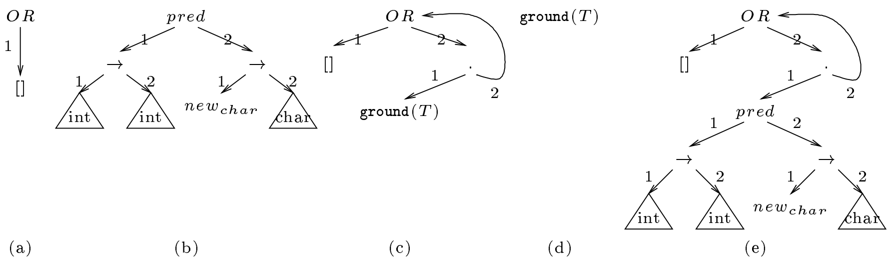

Fig. 3. Gaining information from polymorphic types.

for current ti-state $\{\texttt{S0} \mapsto \texttt{elist}, \texttt{I0} \mapsto \mathit{pred}(\texttt{int} \rightarrow \texttt{int}, new_{char} \rightarrow \texttt{char})$, the remaining variables being `new`. The corresponding ti-trees r_{3a} and r_{3b} are illustrated in Figure 3(a) and (b). The ti-trees defined by the type and mode declarations for the first two arguments of `push/3`, $rt(\texttt{list}(T), \texttt{ground})$ and $rt(T, \texttt{ground})$, are shown in Figure 3(c) and (d) respectively. The literal is schedulable and the matching $\texttt{ground}(T)$ nodes versus in Figure 3(c) and (d) versus the ti-trees of Figure 3(a) and (b), respectively gives $\{(\texttt{ground}(T), r_{3b})\}$. Hence in the output, the only $\{(\texttt{ground}(T)$ nodes must come from this input instantiation, thus the success instantiation for the third argument is $rt(\texttt{list}(T), \texttt{ground}) = r_{3c}$ with the $\texttt{ground}(T)$ node replaced by r_{3b}. Hence, the output instantiation of `S1` is as shown in Figure 3(e). Note that the mode information of the higher-order term is preserved.

There is a caveat which currently prevents this method from being more widely used in HAL. HAL currently assumes that a program variable with variable type may indeed be a solver type and hence can be initialized. This means polymorphic code can introduce the instantiation `old` for polymorphically typed code thus destroying the correctness of the "theorem for free". Thus, this enhanced polymorphic mode checking cannot be used when a matching type is a solver type. However, it is, of course, correct for higher-order types. When type classes are fully integrated in the compiler they will eliminate the assumption above thus removing the caveat.

8 Conclusion

We have formalized mode checking for CLP languages, such as HAL, with strong typing and re-orderable clause bodies, and described the algorithms currently used in the HAL compiler. The implementation of these algorithms in HAL is considerably more sophisticated than the simple presentation here. Partial schedules are computed and stored and accessed only when enough new instantiation information has been created to reassess them. Operations such as $\preceq$ are tabled and hence many operations are simply a lookup in a table. We have found mode checking is efficient enough for a practical compiler, taking 27% of overall compile time on average.

Acknowledgements

We would like to thank Bart Demoen for his contributions to the HAL mode system, and Fergus Henderson and Zoltan Somogyi for discussions of the Mercury mode system.

References

1. A. Aiken and B.R. Murphy. Implementing regular tree expressions. In *Proc. of the 5th FPCA*, LNCS, 427–447. Springer-Verlag, 1991.
2. C. Codognet, P. Codognet, and M. Corsini. Abstract Interpretation of Concurrent Logic Languages. In *Proc. of NACLP 90*, 215–232, October 1990.
3. S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989.
4. B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. Herbrand constraint solving in HAL. In *Proc. of ICLP*, 260–274. MIT Press, 1999.
5. B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. An overview of HAL. In *Proc. of CP99*, LNCS. Springer-Verlag, October 1999.
6. B. Demoen, M. García de la Banda, and P.J. Stuckey. Type constraint solving for parametric and ad-hoc polymorphism. In *Proc. of 22nd ACSC*, 217–228. Springer-Verlag, 1999.
7. T. Fruüwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Procs. IEEE Symp. on Logic in Computer Science*, 1991.
8. F. Gecseg and M. Steinby. *Tree Automata.* Academei Kaido, Budapest, 1984.
9. G. Janssens and M. Bruynooghe. Deriving descriptions of possible value of program variables by means of abstract interpretation. *JLP*, 13:205–258, 1993.
10. K. Marriott and P. Stuckey. The 3 R's of Optimizing Constraint Logic Programs: Refinement, Removal, and Reordering. In *Proc. of 19th POPL*, 334–344. 1992.
11. K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction.* MIT Press, 1998.
12. C.S. Mellish. Abstract Interpretation of Prolog Programs. *Abstract Interpretation of Declarative Languages*, 181–198, 1987.
13. Z. Somogyi. A system of precise modes for logic programs. In *Proc. of 4th ICLP*, 769–787, MIT Press, 1987.
14. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29:17–64, 1996.
15. P. Wadler. Theorems for free. In *FPCA '89 Conference on Functional Programming Languages and Computer Architecture*, 347–359. ACM, 1989.