TLP **5** (6): 623–667, 2005.
doi:10.1017/S1471068404002327 Printed in the United Kingdom

*Checking modes of HAL programs**

MARIA GARCÍA DE LA BANDA, WARWICK HARVEY, KIM MARRIOTT
School of Computer Science & Software Engineering, Monash University, Australia
(*e-mail:* {mbanda,marriott}@csse.monash.edu.au) (*e-mail:* wh@icparc.ic.ac.uk)

PETER J. STUCKEY
Department of Computer Science & Software Engineering, University of Melbourne, Australia
(*e-mail:* pjs@cs.mu.oz.au)

BART DEMOEN
Department of Computer Science, Catholic University Leuven, Belgium
(*e-mail:* bmd@cs.kuleuven.ac.be)

submitted 6 December 2002; revised 16 March 2004; accepted 15 September 2004

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. Unfortunately, mode checking in such languages is difficult. One of the main reasons is that, for each predicate mode declaration, the compiler is required to appropriately re-order literals in the predicate's definition. The task is further complicated by the need to handle complex instantiations (which interact with type declarations and higher-order predicates) and automatic initialization of solver variables. Here we define mode checking for strongly typed CLP languages which require reordering of clause body literals. In addition, we show how to handle a simple case of polymorphic modes by using the corresponding polymorphic types.

KEYWORDS: strong modes, mode checking, regular grammars

1 Introduction

While traditional logic and constraint logic programming (CLP) languages are untyped and unmoded, recent languages such as Mercury (Somogyi *et al.* 1996) and HAL (Demoen *et al.* 1999b; García de la Banda *et al.* 2002) require type, mode and determinism declarations for (exported) predicates. This information allows the generation of efficient target code (e.g. mode information can provide an order of magnitude speed improvement (Demoen *et al.* 1999a)), improves robustness and facilitates efficient integration with foreign language procedures. Here we describe our experience with mode checking in the HAL compiler.

* A preliminary version of this paper appeared under the title "Mode Checking in HAL," in the *Conference on Computational Logic (CL'2000)*, London, June 2000.

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 the compilation. 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. it performs multi-variant specialization). Mode checking involves traversing each predicate mode declaration to check that if the predicate is called with the input instantiation specified by the mode declaration then the following two properties are satisfied. First, the predicate mode declaration is *input-output correct*, that is, it is guaranteed that if the input instantiation satisfies this declaration then the result is an output instantiation that satisfies the declaration. And second, the predicate is *call correct*, that is, if the input instantiation satisfies this declaration then each literal occurring in the definition of the predicate is called with an input instantiation satisfying one of its declared modes.

Call correctness may require the compiler to re-order literals in the body of each rule, so that literals are indeed called with an appropriate input instantiation. Such reordering is essential in logic programming languages which wish to support multi-moded predicates while, at the same time, retaining a Prolog programming style in which a single predicate definition is provided for all modes of usage. And an important function of this reordering is to appropriately order the equalities inserted by the compiler during program normalisation for matching/constructing non-variable predicate arguments. The need to reorder rule bodies is one reason why mode checking is a rather complex task. However, it is not the only reason. Three other issues exacerbate the difficulty of mode checking. 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 predicates is difficult. Third, the compiler needs to handle automatic initialization of solver variables.

Although mode inference and checking of logic programs has been a fertile research field for many years, almost all research has focused on mode checking/inference in traditional (and thus untyped) logic programming languages where the analysis assumes the given literal ordering is fixed and cannot assume that a program is type correct. Thus, a main contribution of this paper is a complete definition of mode checking in the context of CLP languages which are strongly typed and which may require reordering of rule body literals during mode checking.

A second contribution of the paper 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[2])). However, there are significant differences between mode checking in the two languages. In HAL there is the need to handle automatic

[1] In part, because HAL is compiled to Mercury.
[2] Recently a thesis has been completed on Mercury mode checking (Overton 2003).

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

initialization of solver variables, and, in general, complex modes (other than `in` and `out`) are used more frequently since constraint solver variables are usually not ground. Furthermore, determining the best reordering in HAL is more complex than in Mercury because the order in which constraints are solved can have a more significant impact on efficiency (Marriott and Stuckey 1992). 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.

The rest of the paper is organized as follows. In the following section we review related work. Section 3 provides an informal view of the role of types, modes, and instantiations in the HAL language. Its aim is to give insight into the more rigorous formalization provided in section 4 which introduces type-instantiation grammars for combining type and instantiation information as the basis for mode checking in HAL. Section 5 describes the basic steps performed for mode checking HAL programs. Section 6 focuses on the automatic initialization needed by the modes of usage of some predicates. Section 7 discusses mode checking of higher order predicates and objects, while section 8 shows how to handle simple polymorphic modes. Finally, section 9 provides our conclusions and discusses some future work.

2 Related Work

Starting with Mellish (1987) and Debray (1989), there has been considerable research into mode checking and inference in traditional logic programming languages. However, as indicated above, there are two fundamental differences between that work and ours.

First, almost all research assumes that mode analysis is not required to reorder clause bodies. Second, while almost all research has focused on untyped logic programming languages, mode checking of HAL relies on predicates and program variables having a single (parametric polymorphic) Hindley–Milner type and the type correctness of the program with respect to this type. Access to type information allows us to handle more complex instantiations than are usually considered in mode analysis and also to handle mode checking of higher-order predicates in a more rigorous fashion: in most previous work higher-order predicates are largely ignored.

Another important difference is that we are dealing with constraint logic programming languages in which program variables need to be appropriately initialized before being sent to some constraint solver as part of a constraint. Requiring explicit initialization of solver variables puts additional burden on the programmer and makes it impossible to write multi-moded predicate definitions for which different modes require different variable initialisations. We have consequently chosen for the HAL compiler to *automatically* initialize solver variables, i.e. the compiler generates initialization code whenever necessary. To perform such automatic initialization mode checking in HAL must track which program variables are currently uninitialized (in our terminology are `new`). Tracking of uninitialized variables also supports

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

powerful optimizations which can greatly improve performance. For this reason the Mercury mode checker also tracks uninitialized variables.

This need to track uninitialized program variables is a significant difference between mode checking in the Mercury and HAL languages, and most logic programming work on modes. It is not the same as tracking so-called "free" variables in traditional logic programming: first free variables may be aliased to other variables, something that is not possible with uninitialized variables, second, uninitialized variables have to be tracked exactly: the compiler must not fail to initialize a variable, neither should it initialize a variable more than once. We will now review selected related work in detail.

The original work on mode checking in strongly typed logic languages with reorderable clause bodies is that of Somogyi (1987), which gives an informal presentation of a mode system based on types. This is perhaps the closest work in spirit since it was the basis of mode checking in Mercury. However, its mode system is much simpler than ours and it does not consider higher-order predicates or the problems of automatic initialization. The remaining work does not consider compile-time reordering.

Perhaps the most closely related work in traditional logic programming language analysis is the early work of Janssens and Bruynooghe (1993), which uses regular trees to define types and instantiations, and uses these trees to perform mode inference. The main differences are that Janssens and Bruynooghe (1993) do not consider reordering or tracking uninitialized variables. Other more technical differences are that, although we use deterministic tree grammars to formalize types, our type analysis (Demoen *et al.* 1999) is based on a Hindley–Milner approach. A key difference with this and other work such as that of Boye and Małuszyński (1997) is that we describe instantiations for polymorphic types, including higher-order objects. Also, in Janssens and Bruynooghe (1993), depth restrictions are imposed to make the generated regular trees finite. This is not needed in our approach. Finally, they use definite and possible sharing analysis to improve instantiation information. This is not done yet in HAL for complexity reasons (sharing analysis is quite expensive and thus a danger for practical compilation), however a simple sharing and aliasing analysis should indeed prove to be useful.

After the early work of Janssens and Bruynooghe (1993), there has been a significant amount of research aimed at improving the precision of the analysis by providing additional information about the structure of the terms. Initially, this was achieved by performing some simple pattern analysis and then providing this information to other analyses (see for example, Charlier and Hentenryck (1994) and Mulkers *et al.* (1995)). Later, with the gradual success of typed languages, pattern information was substituted by type information with which more accurate results could be obtained, i.e. type information was annotated with different kinds of information some of which were mode information (see, for example, Ridoux *et al.* (1999) and Smaus *et al.* (2000)). But most of this work was designed to either provide a general framework for combining type information with other kinds of information, or to infer some particular kind of information (such as mode information) from a program without reordering the literals in the body of

predicates. Furthermore, they were not interested in tracking uninitialized variables nor keeping enough instantiation information (i.e. which particular tree constructors can occur) for optimizations such as switch detection (Henderson *et al.* 1996). Again, further differences arise since we consider higher-order mode inference and polymorphic modes.

Recent work on *directional types* (e.g see Boye and Małuszyński (1997)) is much more analogous to HAL mode checking. There, they are interested in determining mode-correctness of a program given (user supplied) mode descriptions (called directional types). Apart from previously mentioned differences, the framework of Boye and Małuszyński (1997) uses directional types that are much simpler than the instantiations that we deal with here. Interestingly, the work of Boye and Małuszyński (1997) uses directional type correctness to show that a run-time reordering of a well-typed program will not deadlock, somewhat analogous to our compile-time reordering.

Type dependency analysis (Codish and Lagoon 2000) is also related to mode checking. Their analysis determines type dependencies from which we can read all the correct modes or directional types of a program. The framework is however restricted to use types (and modes) defined by unary function symbols and an ACI operator.

Other related work has been on mode checking for concurrent logic programming languages (Codognet *et al.* 1990): There the emphasis has been on detecting communication patterns and possible deadlocks.

The only other logic programming system we are aware of which does significant mode checking is Ciao (Bueno *et al.* 2002). The Ciao logic programming system (Bueno *et al.* 2002) does mode checking using its general assertion checking framework CiaoPP based on abstract interpretation (Hermenegildo *et al.* 2003). Modes are considered as simply one form of assertion, and indeed the notion of what is a mode is completely redefinable. The default modes are analyzed by the CiaoPP preprocessor using a combination of regular type inference and groundness, freeness and sharing analyses. Ciao modes are more akin to directional types, than the strong modes of HAL and Mercury, and the compiler will check them if possible, and optionally add run-time tests for modes that could not be checked at compile time. As with other earlier work the fundamental differences with the HAL mode system are in treatment of uninitialized variables, reordering, higher-order and polymorphic modes.

3 HAL by example

This section provides an informal view of the role of types, modes, and instantiations in the HAL language. The aim is to provide insight into the more rigorous formalization that will be provided in the following sections. We do this by explaining the example HAL program shown in Figure 1, which implements a polymorphic stack using lists. Note that HAL follows the basic CLP syntax, with variables, rules and predicates defined as usual (see, for example, Marriott and Stuckey (1998) for an introduction to CLP).

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

```
:- typedef list(T) -> ([] ; [T|list(T)]).

:- instdef elist -> [].
:- instdef list(I) -> ([]; [I|list(I)]).
:- instdef nelist(I) -> [I|list(I)].

:- modedef out(I) -> (new -> I).
:- modedef in(I) -> (I -> I).

:- pred push(list(T),T,list(T)).
:- mode push(in,in,out(nelist(ground))) 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(ground)),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 = [].
```

Fig. 1. Example HAL program implementing a polymorphic stack.

3.1 Types

Informally, a ground type describes a set of ground terms and is used as a reasonable approximation of the ground values a particular program variable can take. It is therefore an invariant over the life time of the variable. Types in HAL are prescriptive rather than descriptive, they restrict the possible values of a variable. Unlike much of the work performed on types for logic programming languages, our types only include the ground (also called fixed) values that a variable can take. Later we will describe how instantiations are used to express when a variable takes a value which is not completely fixed.

Types are specified using type definition statements. For instance, in the example shown in Figure 1, the line

```
:- typedef list(T) -> ([] ; [T|list(T)]).
```

defines the polymorphic type constructor `list/1` where `list(T)` is the type of lists with elements of parametric type[3] `T`. These lists are made up using the `[]/0` and `./2` (represented by `[·|·]`) tree constructors.

HAL includes the usual set of built-in basic types: `float` (floating point numbers), `int` (integers), `char` (characters) and `string` (strings). Like most typed languages, HAL provides the means to define type equivalences. For example, the statement

[3] In order to clearly distinguish between program variables and any other kinds of variables (type variables, instantiation variables, etc) we will refer to all other kinds of variables as parameters (i.e., type parameters, instantiation parameters, etc.).

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

```
:- typedef vector = list(int).
```

defines the type `vector` to be a list of integers. Equivalence types are simply macros for type expressions, and the compiler replaces equivalence types by their definition (circular type equivalences are not allowed). From now on we assume that equivalence types have been eliminated from the type expressions we consider. This can be achieved straightforwardly by applying substitution.

Finally, HAL allows a type to be declared as *hidden* so that its definition is not visible outside the module in which it is defined. We note that the treatment of hidden types is almost identical to that of type parameters and so omit them for simplicity.

It is important to note that a program variable's type is used by a compiler to determine the *representation format* for that variable, i.e. the particular way in which program variables are stored during execution. As a result, two program variables may have different types even though the representation of their values can be identical. For example, in a language providing both the ASCII character set and an extended international character set, variables representing each kind of character would need to have different types since their internal representation is different.

3.2 Solvers

In HAL a constraint solver is defined using a new type. Assume for example, that a programmer wishes to implement a constraint solver over floating point numbers. From the point of view of the user, the variables will take floating point values and thus one might expect them to have the built-in type `float`. But their internal representation cannot be a float as they need to keep track of internal information for the solver. As a result, the type of the variables cannot be the built-in type `float` but must be some other type defined by the solver, and whose implementation is hidden from the outside world. This is were we use abstract types, to hide this view from the outside world.

Example 1
For example a floating point solver type `cfloat` might be defined as

```
:- typedef cfloat -> var(int) ; val(float).
```

where the integer in the `var` tree constructor refers to a column number in a (global) simplex tableaux, and the `val` constructor is used to represent simple fixed value floating point numbers. □

Types defined by solvers are called *solver types* and variables with a solver type are called *solver variables*. Solvers must also provide an initialization procedure (`init/1`) and at least the equality (`=/2`) constraint for the type, although many other constraints will be usually provided. Note that solver variables must be initialized before they can be involved in any constraint. This is required so that the solver can keep track of its variables and initialize the appropriate internal data-structures for them.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

The case of Herbrand solver types (i.e. types for which there is a full unification solver) is somewhat special. Any user-defined type can be declared to be a Herbrand solver type by annotating its type definition with the words "deriving solver". For example:

```
:- typedef hlist(T) -> ([] ; [T|hlist(T)]) deriving solver.
```

defines the `hlist` Herbrand type. The compiler will then automatically create an initialization predicate for the type (which is actually identical for all Herbrand types) and an equality predicate for the type which handles not only simple construction, deconstruction and assignment of non-variable terms (which is the only equality support provided for non-solver types), but full unification. As a result, while variables with non-solver type `list` are always required to be bound at run-time to a list of fixed length (so that the limited support provided by construction, deconstruction and assignment is enough),[4] variables with type `hlist` may be bound to open ended lists, where the tail of the list is an unbound (list) variable.

3.3 Instantiations

Instantiations define the set of values, within a type, that a program variable may have at a particular program point in the execution, as well as the possibility that the variable (as yet) takes no value. Instantiation information is vital to the compiler to determine whether equations on terms are being used to construct terms, deconstruct terms or check the equality of two terms. Furthermore, instantiation information is needed to infer the determinism of predicates (i.e. how many answers a predicate has) and to perform many other low-level optimizations.

Although instantiations may seem very similar to types, they should not be confused: a type is invariant over the life of the variable, while instantiations change. Additionally, instantiations reflect the possibility of a variable having no value yet, or being "constrained" to some unknown set of values.

HAL provides three *base* instantiations for a variable: `ground`, `old` and `new`. A variable is `ground` if it is known to have a unique value; the compiler might not know exactly which value within the type (it might depend on the particular execution), but it knows it is fixed (for a solver variable this happens whenever the variable cannot be constrained further).

A variable is `new` if it has not been initialized and it has never appeared in a constraint (thus the name `new`). Thus, it is known to take no value yet. As we have indicated, the instantiation `new` leads to a crucial difference between mode checking in Mercury and HAL, and that investigated in most other research into mode checking of logic programs. Mercury and HAL demand that at each point in execution the compiler knows whether a variable has a value or not. This allows many compiler optimizations, and is a key to the difference in execution speed of Mercury and HAL to most other logic programming systems. The requirement to always have accurate instantiation information about which variables are `new` drives

[4] Note that the elements inside the list need not be ground!

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

many of the decisions made in the mode checking system. In particular, it means that a `new` variable is not allowed to appear inside a data structure, and can only be given a value by assignment or, if it is a solver variable, after initialization.

Finally, the instantiation `old` is used to describe a solver variable that has been initialized but for which nothing is known about its possible values. Note that the variable might be unconstrained, it might be ground, or anything in between (e.g. be greater than 5); the compiler simply does not know. In the case in which `old` is associated with a non-solver variable, it is deemed to be equivalent to `ground`. Note that in Mercury, where there are no solver types, each variable always has an instantiation which is either `new` or (a subset of) `ground`.

It is important to note that `new` is not analogous to free in the usual logic programming sense. A free variable in the HAL context is an `old` variable (thus, it has been initialized by the appropriate solver) which has never been bound to a non-variable term. Thus, free variables might have been aliased, while `new` variables cannot. This is exploited by the compiler by not giving a run-time representation to new variables. As a consequence, a `new` variable cannot occur syntactically more than once.

For data structures such as trees or lists of solver variables, more complex instantiation states may be used. These instantiations are specified using instantiation definition statements which look very much like type definitions, the only difference being that the arguments themselves are instantiations rather than types. For instance, in the example shown in Figure 1, the lines

```
:- instdef elist -> [].
:- instdef list(I) -> ([] ; [I|list(I)]).
:- instdef nelist(I) -> [I|list(I)].
```

define the instantiation constructors `elist/0`, `list/1` and `nelist/1`, which in the example are associated with variables of type `list/1`. In that context, the instantiation `elist` describes empty lists. The polymorphic instantiation `list(I)` describes lists with elements of parametric instantiation `I` (note the deliberate reuse of the type name). Finally, the instantiation `nelist(I)` describes non-empty lists with elements of parametric instantiation `I`.

When associated with a variable, an instantiation requires the variable to be bound to one of the outer-most functors in the right-hand-side of its definition, and the arguments of the functor to satisfy the instantiation of the corresponding arguments in the instantiation definition. In the case of `elist`, it would mean the variable is `ground`. In the remaining two cases, it would depend on the parametric instantiation `I`, but at the very least the variable would be known to be a nil-terminated list, i.e. its length is fixed.

Note that the separation of instantiation information from type information means we can associate the same instantiation for different types. For example, a program variable with solver type `hlist(int)` and instantiation `list(ground)` indicates that the program variable has a fixed length list as its value. A program variable with non-solver type `list(int)` and instantiation `list(ground)` indicates the same, but since the type is not a solver type, this would always be the case. The separation

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

of instantiation information from type information also makes the handling of polymorphic application much more straightforward, since we will simply associate a different type with the same instantiation.

As mentioned before, the instantiation `new` is not allowed to appear as an argument of any other instantiation. As a result, a variable can only be inserted in a data structure if it is either `ground` or initialized (and thus must be `old`). The main reason for this is the requirement for accurate mode information about `new` variables. It quickly becomes very difficult to *always* have correct instantiation information about which variables (and parts of data structures) are `new`. While sharing and aliasing analyses might allow us to keep track which variables are `new` in more situations, inevitably they lead to situations where we cannot determine whether the value of a variable is `new` or not, which is not acceptable to the compiler. We do however plan to use sharing and aliasing analysis to keep track of initialized (`old`) variables that have yet to be constrained (analogous to free variables in Prolog).

3.4 Modes

A mode is of the form $Inst_1 \rightarrow Inst_2$ where $Inst_1$ describes the *call* (or input) instantiation and $Inst_2$ describes the *success* (or output) instantiation. The *base* modes are mappings from one base instantiation to another: we use two letter codes (`oo`, `no`, `og`, `gg`, `ng`) based on the first letter of the instantiation, e.g. `ng` is `new→ground`. The usual modes `in` and `out` are also provided (as renamings of `gg` and `ng`, respectively).

Modes are specified using mode definition statements. For instance, in the example shown in Figure 1, the lines

```
:- modedef out(I) -> (new -> I).
:- modedef in(I) -> (I -> I).
```

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.

HAL allows the programmer to define mode equivalences and instantiation equivalences. As for type equivalences, from now on we assume that these equivalences have been eliminated from the program. For example

```
:- modedef in = in(ground).
:- modedef out = out(ground).
```

define `in` as equivalent to `ground -> ground`, and `out` as `new -> ground`.

3.5 Equality

The equality constraint is a special predicate in HAL. Equality will be normalized in HAL programs to take one of two forms $x_1 = x_2$, and $x = f(x_1, \ldots, x_n)$ where $x, x_1, \ldots, x_n$ are variables. Each form of equality supports a number of modes.

The equality $x_1 = x_2$ can be used in two modes. In the first mode, **copy** (`:=`), either x_1 or x_2 must be new and the other variable must not be new. Assuming x_1

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

is new the value of x_2 is copied into x_1. In the second mode **unify** (==) both x_1 and x_2 must not be new. This requires a full unification.

The equality $x = f(x_1, \ldots, x_n)$ can also be used in two modes. In the first mode, **construct** (:=), x must be new and each of $x_1, \ldots, x_n$ not new. A new f structure is built on the heap, and the values of $x_1, \ldots, x_n$ are copied into this structure. In the second mode, **deconstruct** (=:), each of $x_1, \ldots, x_n$ must be new and x must not be new. If x is of the form $f(a_1, \ldots, a_n)$ then the value of a_i is copied into x_i, otherwise the deconstruct fails.[5] As we shall see later, mode checking shall extend the use of these modes for other implicit modes.

3.6 Predicate declarations

HAL allows the programmer to declare the type and modes of usage of predicates. In our example of Figure 1, the lines

```
:- pred pop(list(T),T,list(T)).
:- mode pop(in,out,out) is semidet.
:- mode pop(in(nelist(ground)),out,out) is det.
```

give such declarations for predicate `pop/3`. The first line is a polymorphic type declaration (with parametric type `T`). It specifies the types of each of the three arguments of `pop/3`. The second and third lines are mode declarations specifying the two different modes in which the predicate can be executed. For example, in the first mode the first argument is `ground` on call and success, while the second and third arguments are `new` on call and `ground` on success.

Each mode declaration for a predicate defines a *procedure*, a different way of executing the predicate. The role of mode checking is not just to show these modes are correct, but also to reorder conjunctions occurring in the predicate definition in order to create these procedures.

The second and third lines also contain a determinism declaration. These describe how many answers a predicate may have for a particular mode of usage: `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 run-time error. Thus, in the second line, since `pop/3` for this mode of usage is guaranteed to have at most one solution but might fail (when the first argument is an empty list), the determinism is `semidet`. For the second mode, the first argument is not only known to be ground but also to be a non-empty list. As a result, the predicate can be ensured to have exactly one solution and so its determinism is `det`. Notice how by providing more complex instantiations we can improve the determinism information of the predicate. They also lead to more efficient code, since unnecessary checks (e.g. that the first argument of `pop/3` is bound to `./2`) are eliminated.

Currently, HAL requires predicate mode declarations for each predicate and checks they are correct. Predicate type declarations, on the other hand, can be omitted and, if so, will be inferred by the compiler (Demoen *et al.* 1999).

[5] This is a simplistic high level view, actually the system uses PARMA bindings and things are more complicated. See Demoen *et al.* (1999a) for details.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

4 Type, instantiation and type-instantiation grammars

In this section we formalize type and instantiation definitions in terms of (extended) regular tree grammars. Then we introduce type-instantiation (ti-) grammars which combine type and instantiation information and are the basis for mode checking in HAL. Throughout the section we will use `teletype` font when referring to (fixed) type and instantiation expressions, and sans serif font when referring to non-terminals of tree grammars.

4.1 HAL programs

We begin by defining basic terminology and HAL programs.

A *signature* Σ is a set of pairs f/n where f is a *function symbol* and $n \geqslant 0$ is the integer *arity* of f. A function symbol with 0 arity is called a *constant*. Given a signature Σ the set of all *trees* (the Herbrand Universe), denoted $\tau(\Sigma)$, is defined as the least set satisfying:

$$\tau(\Sigma) = \bigcup_{f/n \in \Sigma} \{f(t_1, \ldots, t_n) \mid \{t_1, \ldots, t_n\} \subseteq \tau(\Sigma)\}.$$

We assume (for simplicity) that Σ contains at least one constant symbol (i.e. a symbol with arity 0).

Let V be a set of symbols called *variables*. The set of all *terms* over Σ and V, denoted $\tau(\Sigma, V)$, is similarly defined as the least set satisfying:

$$\tau(\Sigma, V) = V \cup \bigcup_{f/n \in \Sigma} \{f(t_1, \ldots, t_n) \mid \{t_1, \ldots, t_n\} \subseteq \tau(\Sigma, V)\}$$

A *substitution* over signature Σ and variable set V is a mapping from variables to terms in $\tau(\Sigma, V)$, written $\{x_1/t_1, \ldots, x_n/t_n\}$. We extend substitutions to map terms in the usual way. A *unifier* for two terms t and t' is a substitution θ such that $\theta(t)$ and $\theta(t')$ are syntactically identical. A *most general unifier* of two terms t and t', denoted $mgu(t, t')$ is a unifier θ such for every other unifier θ' of t and t' there exists a substitution θ'' such that θ' is the composition of θ with θ''. Note that the only substitutions we shall deal with are over type and instantiation parameters.

As we will be dealing with programs, types and instantiations there will be a number of signatures of interest. Let V_{prog} be the set of program variable symbols, and Σ_{tree} be the tree constructors appearing in the program, and Σ_{pred} be the predicate symbols appearing in the program. Let V_{type} and Σ_{type} be the type variables and type constructors, and similarly let V_{inst} and Σ_{inst} be the instantiation variables and instantiation constructors. Note that these alphabets may overlap.

An *atom* is of the form $p(s_1, \ldots, s_n)$ where $\{s_1, \ldots, s_n\} \subseteq \tau(\Sigma_{tree}, V_{prog})$ and $p/n \in \Sigma_{pred}$. A *literal* is either an atom, a variable-variable equation $x_1 = x_2$ where $\{x_1, x_2\} \subseteq V_{prog}$, or a variable-functor equation $x = f(x_1, \ldots, x_n)$ where $f/n \in \Sigma_{tree}$ and $x, x_1, \ldots, x_n$ are distinct elements of V_{prog}. A *goal* G is a literal, a conjunction of goals $G_1, \ldots, G_n$, a disjunction of goals $G_1; \cdots; G_n$ or an if-then-else G_i `->` $G_t; G_e$ (where G_i, G_e, G_t are goals). A *predicate definition* is of the form A `:-` G where A is an atom and G is a goal.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

Note that we are assuming the programs have been normalized, so that each literal has distinct variables as arguments, each equality is either of the form $x_1 = x_2$ or $x = f(x_1, \ldots, x_n)$, where $x, x_1, \ldots, x_n$ are distinct variables, and multiple bodies for a single predicate have been replaced by one disjunctive body.

A *predicate type declaration* is of the form

$$\texttt{:- pred } p(t_1, \ldots, t_n)$$

where $\{t_1, \ldots, t_n\} \subseteq \tau(\Sigma_{type}, V_{type})$ are type expressions. A *predicate mode declaration* is of the form

$$\texttt{:- mode } p(c_1 \rightarrow s_1, \ldots, c_n \rightarrow s_n)$$

where $\{c_1, \ldots, c_n, s_1, \ldots, s_n\} \subseteq \tau(\Sigma_{inst})$ are ground instantiation expressions. A *complete predicate definition* for predicate symbol $p/n \in \Sigma_{pred}$ consists of a predicate definition, a predicate type declaration, and a non-empty set of predicate mode declarations for p/n. A *program* is a collection of complete predicate definitions for distinct predicate symbols.

4.2 Tree grammars

Tree grammars are a well understood formalism (see, for example, Gecseg and Steinby (1984) and Comon *et al.* (1997)) for defining regular tree languages. We first review the standard definitions for tree grammars since we shall have to extend these to handle the complexities of mode checking.

A *tree grammar* r over signature Σ and *non-terminal set* NT is a finite set of *production rules* of the form $x \rightarrow t$ where $x \in NT$ and t is of the form $f(x_1, \ldots, x_n)$ where $f/n \in \Sigma$ and $\{x_1, \ldots, x_n\} \subseteq NT$. For each $x \in NT$ and $f/n \in \Sigma$ we require that there is at most one rule of the form $x \rightarrow f(x_1, \ldots, x_n)$; hence the grammars are *deterministic*.

We have chosen to restrict ourselves to deterministic tree grammars: these grammars are expressive enough for Hindley–Milner types and they give rise to simpler, more efficient algorithms – an important consideration for a compiler designed for large real-world programs.

We assume that from a grammar r we can determine its *root non-terminal*, denoted $root(r)$. In reality this is an additional piece of information attached to each grammar. We shall write grammars so that the root non-terminal appears on the left hand side of the first production rule in r.

It will often be useful to extract a sub-grammar r' from a grammar r defining some non-terminal x appearing in r. If x is a non-terminal occurring in grammar r, then $subg(x, r)$ is the set of rules in r for x and all other non-terminals reachable from x. Or more precisely, $subg(x, r)$ is the smallest set of rules satisfying

$$\begin{aligned} subg(x,r) &\supseteq \{x \rightarrow t \in r\} \\ subg(x,r) &\supseteq \{x' \rightarrow t \in r \mid x' \in NT, \exists x'' \rightarrow g(x''_1, \ldots, x', \ldots, x''_m) \in subg(x,r)\} \end{aligned}$$

The root of the grammar $subg(x, r)$ is x, i.e. $root(subg(x, r)) = x$.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

Example 2
Consider the signature $\{[]/0, '.'/2, a/0, b/0, c/0, d/0\}$ and the non-terminal set {abc, list(abc), bcd, evenlist(bcd), oddlist(bcd)}, then two example regular tree grammars over this signature and non-terminal set are r_1:

$$\begin{array}{rcl} \mathsf{list(abc)} & \rightarrow & [] \\ \mathsf{list(abc)} & \rightarrow & [\mathsf{abc}|\mathsf{list(abc)}] \\ \mathsf{abc} & \rightarrow & a \\ \mathsf{abc} & \rightarrow & b \\ \mathsf{abc} & \rightarrow & c \end{array}$$

and r_2:

$$\begin{array}{rcl} \mathsf{evenlist(bcd)} & \rightarrow & [] \\ \mathsf{evenlist(bcd)} & \rightarrow & [\mathsf{bcd}|\mathsf{oddlist(bcd)}] \\ \mathsf{oddlist(bcd)} & \rightarrow & [\mathsf{bcd}|\mathsf{evenlist(bcd)}] \\ \mathsf{bcd} & \rightarrow & b \\ \mathsf{bcd} & \rightarrow & c \\ \mathsf{bcd} & \rightarrow & d \end{array}$$

The root non-terminal of r_1 is list(abc), while the root non-terminal of r_2 is evenlist(bcd). The grammar $subg(\mathsf{abc}, r_1)$ consists of the last three rules of r_1 while the grammar $subg(\mathsf{oddlist(bcd)}, r_2)$ includes all of the rules of r_2 but we would write the third rule in the first position, to indicate the root non-terminal was oddlist(bcd). □

A production of form $x \rightarrow s$ in some grammar r can be used to rewrite a term $t \in \tau(\Sigma, NT)$ containing an occurrence of x to the term $t' \in \tau(\Sigma, NT)$ where t' is obtained by replacing the occurrence of x in t by s. This is called a *derivation step* and is denoted by $t \Rightarrow t'$. We let $\Rightarrow^*$ be the transitive, reflexive closure of $\Rightarrow$. The *language generated* by r, denoted by $[\![r]\!]$, is the set

$$\{t \in \tau(\Sigma) \mid root(r) \Rightarrow^* t\}$$

Example 3
For example, consider the grammars of Example 2. The set $[\![r_1]\!]$ is all lists of a's, b's and c's, while $[\![r_2]\!]$ is all even length lists of b's, c's and d's. □

For brevity we shall often write tree grammars in a more compressed form. We use

$$x \rightarrow t_1; t_2; \cdots; t_n$$

as shorthand for the set of production rules: $x \rightarrow t_1$, $x \rightarrow t_2$, ..., $x \rightarrow t_n$.

The $[\![\cdot]\!]$ function induces a pre-order on tree grammars: $r_1 \preceq r_2$ iff $[\![r_1]\!] \subseteq [\![r_2]\!]$. If we regard grammars with the same language as equivalent, $\preceq$ gives rise to a natural partial order over these equivalence classes of tree grammars. In fact they form a lattice. However, we shall largely ignore these equivalence classes since all of our operations work on concrete grammars.

We shall also make use of two special grammars. The first is the *least tree grammar*, which we denote by $\bot$. We define that $[\![\bot]\!] = \emptyset$, and so, as its name suggests we have that $\bot \preceq r$ for all grammars r. During mode checking the $\bot$ grammar indicates where execution is known to fail. The second special grammar is the *error grammar*,

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

denoted by $\top$. It is used to indicate that a mode error has occurred and we define that $r \preceq \top$ for all tree grammars r.

We use $\sqcap$ to denote the meet (i.e. greatest lower bound) operator on grammars, and $\sqcup$ to denote the join (i.e. least upper bound) operator. We assume that the non-terminals appearing in the two grammars to be operated on are renamed apart.

We have that $[\![r_1 \sqcap r_2]\!] = [\![r_1]\!] \cap [\![r_2]\!]$. Because we restrict ourselves to deterministic tree grammars the join is inexact: That is to say, $[\![r_1 \sqcup r_2]\!] \supseteq [\![r_1]\!] \cup [\![r_2]\!]$, and for some r_1 and r_2, $[\![r_1 \sqcup r_2]\!] \neq [\![r_1]\!] \cup [\![r_2]\!]$. Of course, since it is the join, it is as precise as possible: for any grammar r such that $[\![r]\!] \supseteq [\![r_1]\!] \cup [\![r_2]\!]$, we have that $[\![r]\!] \supseteq [\![r_1 \sqcup r_2]\!]$.

Algorithms for determining if $r_1 \preceq r_2$, and constructing $r_1 \sqcap r_2$ and $r_1 \sqcup r_2$ are straightforward and omitted.[6]

Example 4

Consider the grammars r_1 and r_2 of Example 2. Their meet $r_1 \sqcap r_2$ is:

```
meet(list(abc),evenlist(bcd))  →  [] ; [meet(abc,bcd) | meet(list(abc),oddlist(bcd))]
                meet(abc,bcd)  →  b ; c
meet(list(abc),oddlist(bcd))   →  [meet(abc,bcd) | meet(list(abc),evenlist(bcd))]
```

while their join $r_1 \sqcup r_2$ is:

```
join(list(abc),evenlist(bcd))  →  [] ; [join(abc,bcd) | join(list(abc),oddlist(bcd))]
                join(abc,bcd)  →  a ; b ; c ; d
join(list(abc),oddlist(bcd))   →  [] ; [join(abc,bcd) | join(list(abc),evenlist(bcd))]
```

Note that the language generated by the grammar $r_1 \sqcup r_2$ could be represented with fewer rules. In the compiler there is no effort to build minimal representations of grammars since non-minimal grammars do not seem to occur that often in practice. □

4.3 Types

Types in HAL are polymorphic Hindley-Milner types. *Type expressions* (or *types*) are terms in the language $\tau(\Sigma_{type}, V_{type})$ where Σ_{type} are *type constructors* and variables V_{type} are *type parameters*. Each type constructor $f/n \in \Sigma_{type}$ must have a definition.

Definition 5

A *type definition* for $f/n \in \Sigma_{type}$ is of the form

$$\texttt{:- typedef} \quad f(v_1, \ldots, v_n) \quad \texttt{->} \quad (f_1(t_1^1, \ldots, t_{m_1}^1); \cdots; f_k(t_1^k, \ldots, t_{m_k}^k)).$$

where $v_1, \ldots, v_n$ are distinct type parameters, $\{f_1/m_1, \ldots, f_k/m_k\} \subseteq \Sigma_{tree}$ are distinct tree constructor/arity pairs, and $t_1^1, \ldots, t_{m_k}^k$ are type expressions involving at most parameters $v_1, \ldots, v_n$. The type definition for f/n may optionally have `deriving solver` appended. If so then types of the form $f(t_1, \ldots, t_n)$ are *solver types*, otherwise they are *non-solver types*. □

[6] The final operations of interest are given in the appendix.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

Clearly, the type definition for f can be viewed as simply a set of production rules over signature Σ_{tree} and non-terminal set $\tau(\Sigma_{type}, V_{type})$.

We can associate with each (non-parameter) type expression the production rules that define the topmost symbol of the type. Let t be a type expression of the form $f(t_1, \ldots, t_n)$ and let f/n have type definition

$$\texttt{:- typedef}\quad f(v_1, \ldots, v_n)\quad \texttt{->}\quad (f_1(t_1^1, \ldots, t_{m_1}^1); \cdots; f_k(t_1^k, \ldots, t_{m_k}^k)).$$

We define $rules(t)$ to be the production rules:

$$\theta(f(v_1, \ldots, v_n)) \rightarrow (f_1(\theta(t_1^1), \ldots, \theta(t_{m_1}^1)); \cdots; f_k(\theta(t_1^k), \ldots, \theta(t_{m_k}^k)))$$

where $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$. If $t \in V_{type}$ we define $rules(t)$ to be the empty set.

We can extend this notation to associate a tree grammar with a type expression. Let $grammar(t)$ be the least set of production rules such that:

$$\begin{array}{lcl} grammar(t) & \supseteq & rules(t) \\ grammar(t) & \supseteq & \bigcup\{rules(t') \mid \exists x' \rightarrow g(t'_1, \ldots, t', \ldots, t'_m) \in grammar(t))\} \end{array}$$

We assume that $root(grammar(t)) = t$. Note at this point we make no distinction between solver types and non-solver types; this will only occur once we consider instantiations.

To avoid type expressions that depend on an infinite number of types we restrict the type definitions to be *regular* (Mycroft 1984). A type t is *regular* if $grammar(t)$ is finite.[7]

Consider for example the non-regular type definition:

```
:- typedef erk(T) -> node(erk(list(T)), T).
```

The meaning of the type `erk(int)` depends on the meaning of the type `erk(list(int))`, which depends on the meaning of the type `erk(list(list(int)))`, etc. By restricting to regular types we are guaranteed that each type expression only involves a finite number of types.

A *ground type expression* t is an element of $\tau(\Sigma_{type})$. The grammar corresponding to ground type expression t defines the meaning of the type expression as a set of trees ($[\![grammar(t)]\!]$). Note that during run-time every variable (for each invocation of a predicate) has a unique ground type in $\tau(\Sigma_{type})$.

Example 6
Given the type definitions:

```
:- typedef abc -> a ; b ; c.
:- typedef list(T) -> [] ; [T | list(T)].
```

then the grammar r_1 shown in Example 2 is $grammar($`list(abc)`$)$. The set $[\![r_1]\!]$ is the set of lists of `a`'s, `b`'s and `c`'s. The grammar $grammar($`list(T)`$)$ is

$$\mathsf{list}(T)\quad \rightarrow\quad [\,]\ ;\ [T|\mathsf{list}(T)]$$

[7] Note that non-regular types are rarely used (although see Okasaki (1998)). The compiler could be extended to support mode checking for non-regular types as long as we keep the restriction to regular instantiations.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

The set $[\![grammar(\texttt{list(T)})]\!] = \{[]\}$. □

Note that the grammars corresponding to non-ground type expressions are not very interesting, as illustrated in the above example. We can think of a non-ground type expression as a mapping from grounding substitutions to (ground) types whose meaning is then given by their corresponding grammar.

The built-in types `float`, `int`, `char` and `string` are conceptually expressible as (possibly infinite) tree grammars. For example, `int` can be thought of as having the (infinite) definition:

```
:- typedef int -> 0 ; 1 ; -1 ; 2 ; -2 ; 3 ; ...
```

Though the infinite number of children will render some of the algorithms on tree grammars ineffective this is easily avoided in the compiler by treating the type expressions specially (we omit details in our algorithms since it is straightforward).

Note that in HAL, type inference and checking is performed using a constraint-based Hindley–Milner approach on the type expressions (Demoen *et al.* 1999). In this paper we assume that type analysis has been performed previously, and there are no type errors. For the purposes of mode checking the type correctness of a program has four main consequences. First, each program variable is known to have a unique polymorphic type. Second, all values taken by a variable during the execution are known to be members of this type. Third, calls to a polymorphic predicate are guaranteed to have an equal or more specific type than that of the predicate. Fourth, all type parameters appearing in the type of a variable in the body of a predicate are known to also appear in the type of some variable in the head of the predicate. Together, these guarantee that whenever we compare grammars during mode checking, they correspond to exactly the same type.[8] This is used to substantially simplify the algorithms for mode checking (see, for example, the re-definition of function $\preceq$ in section 4.6, and the assumption on the existence of type environment θ at the beginning of section 7.1).

4.4 Values

Types only express sets of fixed values (subsets of $\tau(\Sigma_{tree})$). However, during execution variables do not always have a fixed value and it is the role of mode checking to track these changes in variable instantiation. Thus, to perform mode checking we need to introduce special constants, `#fresh#` and `#var#`, to represent the two kinds of non-fixed values that a program variable can have during execution.

The `#fresh#` constant is used to represent that a program variable takes no value (i.e. it has not been initialized), and corresponds to the `new` instantiation. Note that in HAL there is no run-time representation for `#fresh#` variables. As a result, the compiler needs to know at all times whether a variable is `new` or not. Thus, any tree language including `#fresh#` and some other term is not a valid description of the values of a program variable.

[8] Even mode checking a call to a polymorphic predicate will use the calling type, which may be more specific than the predicate's declared type.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

The #var# constant is used to represent a program variable (or part of a value) that has been initialized but not further constrained. It corresponds to a "free" variable in the usual logic programming sense. The #var# constructor will arise in descriptions of `old` instantiations, where we can define values which are not fixed. Of course it will only make sense for variables of solver types to take on this value.

The values that a variable can take are thus represented by trees in $\tau(\Sigma_{tree} \cup \{\#\texttt{var}\#\}) \cup \{\#\texttt{fresh}\#\}$.

4.5 Instantiations

A type expression by itself represents a set of fixed values. An instantiation by itself has little meaning, it is just a term in the language of expressions. Its meaning is only defined when it is considered in the context of a type expression. For instance, the meaning of `ground` depends upon the type of the variable it is referring to.

In the following section we define a function $\text{RT}(t, i)$ which takes a type expression t and an instantiation i and returns a tree-grammar defining the set of (possibly non-fixed) values that a program variable with the given type and instantiation can take. In this section we define the function $\text{BASE}(t, i)$ which is the function $\text{RT}(t, i)$ for the particular case in which i is a base instantiation. In order to avoid name clashes, the function creates a unique non-terminal grammar symbol $ti(t, base)$ for the type t and base instantiation *base* with which it is called and returns this together with the grammar for t and *base*. The symbol $ti(t, i)$ represents the root of the tree-grammar which defines the possible values of a variable of type t and instantiation i.

When a program variable is `new` it can only have one possible value, #fresh#. Hence the grammar returned by $\text{BASE}(t, \texttt{new})$ for any type t is simply

$$ti(t, \texttt{new}) \quad \rightarrow \quad \#\texttt{fresh}\#$$

In a slight abuse of notation we will use `new` to refer to this grammar.

When a program variable is `ground` it can take any fixed value. If the type t of the variable is ground, then $\text{BASE}(t, \texttt{ground})$ is identical to the grammar defining its type ($grammar(t)$). Type parameters complicate this somewhat. Since we are going to reason about the values of variables with non-ground types we need a way of representing the possible ground values of a type parameter. We introduce new constants of the form $\$\texttt{ground}(v)\$$ where $v \in V_{type}$ to represent these languages. So for $t \in V_{type}$ the grammar $\text{BASE}(t, \texttt{ground})$ is defined as

$$ti(t, \texttt{ground}) \quad \rightarrow \quad \$\texttt{ground}(t)\$$$

For arbitrary types t, $\text{BASE}(t, \texttt{ground})$ is defined as the union of the rules

$$ti(t', \texttt{ground}) \quad \rightarrow \quad f(ti(t_1, \texttt{ground}), \ldots, ti(t_n, \texttt{ground}))$$

for each $t' \rightarrow f(t_1, \ldots, t_n)$ occurring in $grammar(t)$, with

$$ti(t', \texttt{ground}) \quad \rightarrow \quad \$\texttt{ground}(t')\$$$

for each $t' \in V_{type}$ occurring in $grammar(t)$.

Conceptually, the new constant $\$\texttt{ground}(v)\$$ is a place holder for the grammar $\text{BASE}(t', \texttt{ground})$ obtained if v were replaced by the ground type t'.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

When a program variable is `old` it can take any initialized value. This will have a different effect on the parts of the type which are solver types themselves and on those which are not. Non-solver types do not allow the possibility of taking an initialized but unbound value (represented by the value `#var#`). Thus, for solver types t we shall add a production rule $t \rightarrow$ `#var#` to the usual rules defining the type, while non-solver types remain unchanged. In order to handle type parameters we introduce another set of constants `$old(`v`)$` where $v \in V_{type}$. Each constant is simply a place holder for BASE(t', `old`) obtained if v were replaced by the ground type t'. Thus, BASE(t, `old`) for $t \in V_{type}$ is defined as

$$ti(\mathtt{t}, \mathtt{old}) \quad \rightarrow \quad \mathtt{\$ground(}t\mathtt{)\$} \ ; \ \mathtt{\$old(}t\mathtt{)\$}$$

and otherwise BASE(t, `old`) is defined as the rules

$$ti(t', \mathtt{old}) \quad \rightarrow \quad f(ti(t_1, \mathtt{old}), \ldots, ti(t_n, \mathtt{old}))$$

for each rule $t' \rightarrow f(t_1, \ldots, t_n)$ in *grammar*(t), together with

$$ti(t', \mathtt{old}) \quad \rightarrow \quad \mathtt{\#var\#}$$

for each solver type t' occurring in *grammar*(t), and

$$ti(t', \mathtt{old}) \quad \rightarrow \quad \mathtt{\$ground(}t'\mathtt{)\$} \ ; \ \mathtt{\$old(}t'\mathtt{)\$}$$

for each type variable $t' \in V_{type}$ occurring in *grammar*(t).

The reason we represent an `old` variable of type t using both the `$ground(`t'`)$` and `$old(`t'`)$`, is that then a `ground` variable of type t defines a sublanguage. This will simplify many algorithms.

Example 7
Given the type definitions:

```
:- typedef abc -> a ; b ; c.
:- typedef hlist(T) -> [] ; [T | hlist(T)] deriving solver.
```

Then *olabc1* = BASE(`hlist(abc)`, `old`) is the grammar:

$$\begin{aligned} ti(\mathsf{hlist(abc)}, \mathtt{old}) \quad &\rightarrow \quad [] \ ; \ [ti(\mathsf{abc}, \mathtt{old}) \mid ti(\mathsf{hlist(abc)}, \mathtt{old})] \ ; \ \mathtt{\#var\#} \\ ti(\mathsf{abc}, \mathtt{old}) \quad &\rightarrow \quad a \ ; \ b \ ; \ c \end{aligned}$$

The set $[\![olabc1]\!]$ includes the values $[], [a|\mathtt{\#var\#}], [b], [b, a, c, a|\mathtt{\#var\#}]$. The symbol `#var#` represents an uninstantiated variable, and so the second and fourth values are open-ended lists.

As another example, imagine we swap which type is a solver type. That is, suppose we have definitions

```
:- typedef habc -> a ; b ; c deriving solver.
:- typedef list(T) -> [] ; [T | list(T)].
```

Then *olabc2* = BASE(`list(habc)`, `old`) is the grammar:

$$\begin{aligned} ti(\mathsf{list(habc)}, \mathtt{old}) \quad &\rightarrow \quad [] \ ; \ [ti(\mathsf{habc}, \mathtt{old}) \mid ti(\mathsf{list(habc)}, \mathtt{old})] \\ ti(\mathsf{habc}, \mathtt{old}) \quad &\rightarrow \quad a \ ; \ b \ ; \ c \ ; \ \mathtt{\#var\#} \end{aligned}$$

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

The set $[\![olabc2]\!]$ includes the values $[], [a], [\#\texttt{var}\#, b, \#\texttt{var}\#]$ which are all fixed-length lists whose elements may be variables. Note that the two occurrences of the symbol `#var#` in the last tree do not necessarily represent the same solver variable.

Finally BASE(`hlist(T),old`) is (using the first definition)

$$\begin{array}{rcl} ti(\texttt{hlist}(T),\texttt{old}) & \rightarrow & [] \ ; \ [ti(T,\texttt{old}) \mid ti(\texttt{hlist}(T),\texttt{old})] \ ; \ \#\texttt{var}\# \\ ti(T,\texttt{old}) & \rightarrow & \$\texttt{ground}(T)\$ \ ; \ \$\texttt{old}(T)\$ \end{array}$$

□

Let us now consider instantiations in general, rather than only base instantiations. *Instantiation expressions* (or *instantiations*) are terms in the language $\tau(\Sigma_{inst}, V_{inst})$ where Σ_{inst} are *instantiation constructors* and variables V_{inst} are *instantiation parameters*. Each instantiation constructor $g/n \in \Sigma_{inst}$ must have a definition. Often, we will overload functors as both type and instantiation constructors (so Σ_{type} and Σ_{inst} are not disjoint). The base instantiations (`ground`, `old` and `new`) are simply special 0-ary elements of Σ_{inst}.

Definition 8
An *instantiation definition* for g is of the form:

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

where $w_1,\ldots,w_n$ are distinct instantiation parameters, $\{g_1/m_1,\ldots,g_k/m_k\} \subseteq \Sigma_{tree}$ are distinct tree constructors, and $i_1^1,\ldots,i_{m_k}^k$ are instantiation expressions other than `new`[9] involving at most the parameters $w_1,\ldots,w_n$. Just as for type definitions, we demand that instantiation definitions are *regular*.[10] □

We can associate a set of production rules $rules(i)$ with an instantiation expression i just as we do for type expressions. For the base instantiations we define $rules(\texttt{new}) = rules(\texttt{old}) = rules(\texttt{ground}) = \emptyset$.

A *ground instantiation* is an element of $\tau(\Sigma_{inst})$. The existence of instantiation parameters during mode analysis would significantly complicate the task of the analyzer. This is mainly because functions to compare type-instantiations or to compute their join and meet would need to return a set of constraints involving instantiation parameters. Furthermore, predicate mode declarations containing instantiation parameters might need to express some constraints involving those instantiations. Therefore, for simplicity, HAL (like Mercury[11]) requires instantiations appearing in a predicate mode declaration to be ground. As a result, mode checking only deals with ground instantiations and, from now on, we will assume all instantiations are ground.

The reason this problem does not arise with type parameters is that, as mentioned before, type correctness guarantees that whenever we compare type-instantiations, the two types being compared are syntactically identical. Thus, if two type parameters are being compared, they are guaranteed to be the same type parameter.

[9] As mentioned before, disallowing nesting of the `new` instantiation simplifies mode analysis. It also ensures that all subparts of a data structure have a proper representation at run-time.
[10] It is hard to see how to lift this restriction.
[11] Recently Mercury has added a (as yet unreleased) feature allowing limited non-ground instantiations in predicate modes.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

```
RT(t,i)
    (r, _) := RT(t,i,∅)
    return r

RT(t,i,P)
    if (ti(t,i) ∈ P) return (∅, ti(t,i)))
    if (i is a base instantiation) return (BASE(t,i), ti(t,i))
    if (t ∈ V_type) return (⊤, _)
    r := ∅
    foreach rule x_i → f(x_i1, ..., x_in) in rules(i)
        if exists rule x_t → f(x_t1, ..., x_tn) in rules(t)
            for j = 1..n
                (r_j, x_j) := RT(x_tj,x_ij, P ∪ {ti(t,i)})
                if (r_j = ⊤) return (⊤, _)
            endfor
            r := {ti(t,i) → f(x_1, ..., x_n)} ∪ r ∪ r_1 ∪ ··· ∪ r_n
        endif
    endfor
    return (r, ti(t,i))
```

Fig. 2. Algorithm for computing the type instantiation grammar RT(t,i).

4.6 Type-instantiation grammars

In this section we define the function RT(t,i) which takes a type expression t and a ground instantiation expression i and returns a *type-instantiation tree grammar* (or *ti-grammar*). Mode checking will manipulate *ti-grammar*s, built from the types and instantiations occurring in the program.

The function RT defines the meaning of combining a type with an instantiation by extending BASE to non-base instantiations. A non-base instantiation combines with a type in a manner analogous to the $\sqcap$ operation over the rules defining each other. Intuitively the function RT intersects the grammars of t and i. This is not really the case because of special treatment of type parameters and base instantiations.

Figure 2 gives the algorithm for computing RT(t,i). The function RT(t,i,P) does all of the work. It creates a unique grammar symbol $ti(t,i)$ for the type t and instantiation i with which it is called and returns this with the type instantiation grammar for t and i. Its last argument P is the set of grammar symbols constructed in the parent calls: this is used to check that the symbol $ti(t,i)$ has not already been encountered and so avoids infinite recursion. The root of the grammar r returned is the symbol $ti(t,i)$.

Note that it is a mode error to associate a non-base instantiation with a parameter type $t \in V_{type}$, since we cannot know what function symbols make up the type t. In this case the algorithm returns the special $\top$ grammar indicating a mode error.

Example 9

Consider the types `list/1` and `habc` of Example 7 and instantiation `nelist/1` from the program in Figure 1. Then ti-grammar RT(`list(habc)`,`nelist(old)`) is

$$\begin{aligned} ti(\mathsf{list(habc)}, \mathsf{nelist(old)}) &\rightarrow [ti(\mathsf{habc}, \mathsf{old}) \mid ti(\mathsf{list(habc)}, \mathsf{list(old)})] \\ ti(\mathsf{list(habc)}, \mathsf{list(old)}) &\rightarrow [] \ ; \ [ti(\mathsf{habc}, \mathsf{old}) \mid ti(\mathsf{list(habc)}, \mathsf{list(old)})] \\ ti(\mathsf{habc}, \mathsf{old}) &\rightarrow a \ ; \ b \ ; \ c \ ; \ \#\mathtt{var}\# \end{aligned}$$

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

while RT(list(T),nelist(ground)) is

$$\begin{array}{rcl} ti(\mathsf{list}(T),\mathsf{nelist}(\mathsf{ground})) & \rightarrow & [ti(T,\mathsf{ground})|ti(\mathsf{list}(T),\mathsf{list}(\mathsf{ground}))] \\ ti(\mathsf{list}(T),\mathsf{list}(\mathsf{ground})) & \rightarrow & [] \ ; \ [ti(T,\mathsf{ground}) \mid ti(\mathsf{list}(T),\mathsf{list}(\mathsf{ground}))] \\ ti(T,\mathsf{ground}) & \rightarrow & \$\mathtt{ground}(T)\$ \end{array}$$

□

A ti-grammar is thus a regular tree grammar defined over the signature

$$\Sigma_{tree} \cup \{\$\mathtt{old}(v)\$, \$\mathtt{ground}(v)\$ \mid v \in V_{type}\} \cup \{\#\mathtt{var}\#, \#\mathtt{fresh}\#\}$$

and non-terminal set

$$\tau(\Sigma_{type} \cup \Sigma_{inst} \cup \{ti/2\}, V_{type}) \cup \{\mathsf{new}\}$$

Note that by construction the partial ordering and meet and join on tree grammars extend to ti-grammars including type parameters. As mentioned before, type correctness guarantees that during mode checking we will only compare ti-grammars for the same type parameter $v \in V_{type}$. For this reason, we only need note that $\text{RT}(v, \mathtt{ground}) \preceq \text{RT}(v, \mathtt{old})$ for a parameter $v \in V_{type}$, which follows from the construction, as $[\![\text{RT}(v, \mathtt{ground})]\!] = \{\$\mathtt{ground}(v)\$\}$ and $[\![\text{RT}(v, \mathtt{old})]\!] = \{\$\mathtt{ground}(v)\$, \$\mathtt{old}(v)\$\}$ and the meet and join operations follow in the natural way.

The operations that we perform on ti-grammars during mode checking will be $\preceq$, abstract conjunction and abstract disjunction. Abstract conjunction differs slightly from $\sqcap$ since we will be changing variables with a new ti-grammar to ti-grammars for bound values (whenever the variable becomes instantiated). The abstract conjunction operation $\wedge$ is defined as:

$$r_1 \wedge r_2 \quad = \quad \begin{cases} r_1, \text{ where } r_2 = \mathsf{new} \\ r_2, \text{ where } r_1 = \mathsf{new} \\ r_1 \sqcap r_2, \text{ otherwise} \end{cases}$$

Abstract disjunction is again slightly different from the $\sqcup$ operation. Since the compiler needs to know whether the value of a variable is new or not, we need to ensure the abstract disjunction operation does not create ti-grammars (other than $\top$) in which this information is lost, i.e. grammars that include #fresh# as well as other terms. The abstract disjunction operation $\vee$ is defined as:

$$r_1 \vee r_2 \quad = \quad \begin{cases} r_1 \sqcup r_2, \text{ where } r_1 \neq \mathsf{new} \text{ and } r_2 \neq \mathsf{new} \\ \mathsf{new}, \text{ where } r_1 = \mathsf{new} \text{ and } r_2 = \mathsf{new} \\ \top, \text{ otherwise} \end{cases}$$

Finally, 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-grammars. Ti-grammars are used during mode checking to store the possible values of the program variables at each program point. We can extend operations on ti-grammars to ti-states over the same set of variables in the obvious pointwise 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 $1 \leqslant l \leqslant n$,
- $TI \wedge TI' = \{x_l \mapsto r_l \wedge r'_l \mid 1 \leqslant l \leqslant n\}$ and
- $TI \vee TI' = \{x_l \mapsto r_l \vee r'_l \mid 1 \leqslant l \leqslant n\}$.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

5 Basic mode checking

Mode checking is a complex process which aims to reorder body literals to satisfy the mode constraints provided by each mode declaration. The aim of this is to be able to generate specialized code for each mode declaration. The code corresponding to each mode declaration is referred to as a *procedure*, and calls to the original predicate are replaced by calls to the appropriate procedure. Recall that before mode checking is applied the HAL compiler performs type checking (and inference) so that each program variable has a type, and the program is guaranteed to be type correct.

5.1 Well-moded programs

We now define what it means for a HAL program to be well-moded.

The execution of a HAL program is performed on *procedures* which are predicates re-ordered for a particular mode. At run-time each type parameter has an associated ground type. For our purposes we assume a given *type environment* θ (a ground type substitution) describes the run-time types associated with each type parameter.

A *call* to a procedure p/n in mode $p(c_1 \rightarrow s_1, \ldots, c_n \rightarrow s_n)$ is a type environment θ and a *value* d_i for each argument $1 \leqslant i \leqslant n$. It follows from type correctness of the program that $d_i \in [\![\text{RT}(\theta(t_i), \texttt{old})]\!] \cup \{\texttt{\#fresh\#}\}$ for each argument $1 \leqslant i \leqslant n$.

A program is *input-output mode-correct* if any call to a predicate which is correct with respect to the input instantiation for some mode declared for that predicate will only have answers that are correct with respect to the output instantiation of that mode. More formally, a program is *input-output mode-correct* if for each procedure p/n with declared type $p(t_1, \ldots, t_n)$ in mode $p(c_1 \rightarrow s_1, \ldots, c_n \rightarrow s_n)$, and for any call of the form $p(d_1, \ldots, d_n)$ with type environment θ such that $d_i \in [\![\text{RT}(\theta(t_i), c_i)]\!], 1 \leqslant i \leqslant n$, it is the case that the resulting values $d'_1, \ldots, d'_n$ on success of the procedure are such that $d'_i \in [\![\text{RT}(\theta(t_i), s_i)]\!], 1 \leqslant i \leqslant n$. In other words, the declared mode is satisfied by the code generated for the procedure.

Example 10

For example the first mode for predicate `pop/3`, defined in Example 1,

```
:- mode pop(in,out,out) is semidet.
```

will be shown to be input-output mode-correct by showing that if the first argument to `pop/3` is ground at call time, and the last two arguments `new`, then all three arguments will be ground on success of the predicate. □

A program is *call mode-correct* if any call to a predicate which is correct with respect to the input instantiation for some mode declared for that predicate will only lead to calls to literals within the definition of the predicate which are mode-correct. More formally, a program is *call mode-correct* if for each procedure p/n with declared type $p(t_1, \ldots, t_n)$ in mode $p(c_1 \rightarrow s_1, \ldots, c_n \rightarrow s_n)$, and for any call of the form $p(d_1, \ldots, d_n)$ with type environment θ such that $d_i \in [\![\text{RT}(\theta(t_i), c_i)]\!], 1 \leqslant i \leqslant n$, it is the case that each call to a procedure p'/n' with type (given by the occurrence in the definition of p/n) $p'(t'_1, \ldots, t'_{n'})$ in mode $p(c'_1 \rightarrow s'_1, \ldots, c'_{n'} \rightarrow s'_{n'})$ of the form $p'(d'_1, \ldots, d'_n)$ is such that $d'_i \in [\![\text{RT}(\theta(t'_i), c'_i)]\!], 1 \leqslant i \leqslant n'$, and any call to an equality

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

of the form $x_1 = x_2$ is either a copy or unify, and any call to an equality of the form $x = f(x_1, \ldots, x_n)$ is either a construct or deconstruct. In other words each mode-correct call leads to only mode-correct calls.

Example 11
Consider the following code which duplicates the top element of the stack:

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

Showing call mode-correctness for the procedure for `dupl/2` involves showing that any correct call to `dupl/2` (that is with the first argument a non-empty ground list, and its second argument `new`) will call `push/3` and `pop/3` with correct input instantiations for one of their given modes, and each equation must be either a construct or deconstruct. □

A program is *well-moded* if it is input-output mode-correct and call mode-correct.

We shall now explain mode checking by showing how to check whether each program construct is schedulable for a given ti-state TI and, if so, what the resulting ti-state TI' is. The scheduling also returns a goal illustrating the order of execution of conjunctions, and the mode for each equation or predicate call. If the program construct is not schedulable for the given ti-state it may be reconsidered after other constructs have been scheduled. We assume that before checking each construct for an initial ti-state TI, we extend TI so that any variable of type t local to the construct is assigned the ti-grammar new.

5.2 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 (say r_1), the **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 then **unify** $x_1 == x_2$ is performed and the resulting instantiation is $TI' = \{x_1 \mapsto r_1 \wedge r_2, x_2 \mapsto r_1 \wedge r_2\} \cup RTI$. If neither of the two modes of usage apply (i.e. both variables are new), the literal is not schedulable (although it might become schedulable after automatic initialization, see Section 6).

Consider the equality $x = f(x_1, \ldots, x_n)$ where $x, x_1, \ldots, x_n$ are variables with types $\{x \mapsto t, x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ 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 and none of the r_j are new. 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-grammar defined by $\{a \rightarrow f(root(r_1), \ldots, root(r_n))\} \cup r_1 \cup \cdots \cup r_n$, where a is a new non-terminal, (i.e. the grammar defining the terms constructible from an f with arguments from $r_1, ..., r_n$ respectively). The **deconstruct** mode applies if each r_j is new and r is not new and has no production rule $root(r) \rightarrow$ #var#

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

(which means it is definitely bound to some functor). The resulting ti-state is $TI' = \{x \mapsto r, x_1 \mapsto r'_1, \ldots, x_n \mapsto r'_n\} \cup RTI$ where $r'_1, \ldots, r'_n$ are defined below. If r has a production rule of the form $root(r) \rightarrow f(y_1, \ldots, y_n)$, then the $r'_j = subg(y_j, r), 1 \leqslant i \leqslant n$. If r has no rule of this form, then the resulting ti-state is the same but with $r'_j = \bot, 1 \leqslant j \leqslant n$, indicating that the **deconstruct** must fail. If some of the variables x_j are `new` 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. $x = f(x_1, \ldots, fresh_{k_j}, \ldots), \ldots, x_{k_j} = fresh_{k_j}, \ldots$. These new equalities are handled as above.

Note that if r = `new` and some r_i = `new` then the literal is not schedulable (although it might become schedulable after automatic initialization, again see section 6).

Example 12
Assume `X` and `Y` are ground lists, while `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 run-time and correspond to the modes of usage allowed by Mercury. HAL, in addition to the above, allows the use of the **deconstruct** mode when x is `old` (i.e. r contains a production rule $root(r) \rightarrow$ `#var#`). In this case we check whether r has a production rule of the form $root(r) \rightarrow f(y_1, \ldots, y_n)$ and we proceed as in the previous paragraph. Note that this is (the only place) where the HAL mode system is not completely strong (i.e. run-time mode errors can occur). The following example illustrates the need for this behavior.

Example 13
Consider the types `abc/0` and `hlist/1` from Example 7, the following use of `append/3` may not detect a mode error until run-time:

```
:- pred append(hlist(abc), hlist(abc), hlist(abc)).
:- mode append(oo, oo, no) is nondet.
append(X, Y, Z) :- X = [], Y = Z.
append(X, Y, Z) :- X = [A|X1], append(X1, Y, Z1), Z = [A|Z1].
```

The equation `X = [A|X1]` is schedulable as a deconstruct since `X` is `old`. However, if at run-time `X` is not bound when `append/3` is called, the deconstruct will generate a run-time error since `A` is not a solver variable and, thus, it cannot be initialized. Note that if we did not allow deconstruction on `old` variables then the above predicate would not pass mode checking thus preventing mode-correct goals like

```
?- X = [a,b,c], init(Y), append(X,Y,Z).
```

from being compiled. □

If we never allow Herbrand solver types to contain non-solver types (as in the example above), the problem cannot occur. This gap in mode checking seems unavoidable if we are to allow Herbrand solver types to contain non-solver types. However, it seems that in practice this gap is not problematic: in most programs, the possibility of a run-time mode error does not exist. Whenever it does, the compiler emits a warning message. In fact, we have never detected a run-time mode error.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

5.3 Predicates

In this subsection we describe the scheduling of predicate calls so that the resulting program after scheduling is call mode-correct.

Consider the predicate call $p(x_1,\ldots,x_n)$ where each x_i is a variable with type t_i. 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 $TI = \{x_1 \mapsto r_1,\ldots,x_n \mapsto r_n\} \cup RTI$.

Note that the handling of polymorphic application is hidden here, since the type t_i of the variable x_i is type in the calling literal $p(x_1,\ldots,x_n)$, which may be more specific than the declared/inferred type of argument i of p. Because instantiations are separate from types this is straightforwardly expressed by constructing the ti-grammar for the mode specific calling type t_i and the appropriate instantiations.

The predicate call can be scheduled if for each $1 \leqslant j \leqslant n$ the current ti-state restricts the j-th argument more than (defines a subset of) the calling ti-state required for p, i.e. $r_j \preceq \mathrm{RT}(t_j, c_j)$. If the predicate call is schedulable for this mode the new ti-state is $TI' = \{x_1 \mapsto r_1 \wedge \mathrm{RT}(t_1, s_1),\ldots,x_n \mapsto r_n \wedge \mathrm{RT}(t_n, s_n)\} \cup RTI$. The predicate call can also be scheduled if for each j such that $r_j \not\preceq \mathrm{RT}(t_j, c_j)$ then $\mathrm{RT}(t_j, c_j) = \mathsf{new}$. For each 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. Such "extra" modes are usually referred to as *implied modes*.

Example 14

Consider the goal `empty(S0)` for the program of Figure 1 where the type of `S0` is given by $\{\texttt{S0} \mapsto \texttt{list(abc)}\}$ (which is more specific than the declared type `list(T)`) and the current ti-state is $TI = \{\texttt{S0} \mapsto \mathsf{new}\}$. The two modes for `empty` (in expanded form) are

```
:- mode empty(ground -> ground) is semidet.
:- mode empty(new -> ground) is det.
```

The first mode of `empty` cannot be scheduled since $\mathsf{new} \not\preceq \mathrm{RT}(\texttt{list(abc)}, \texttt{ground})$, but the second mode can be scheduled, since $\mathsf{new} \preceq \mathrm{RT}(\texttt{list(abc)}, \texttt{new}) = \mathsf{new}$. □

If more than one mode of the same predicate is schedulable, in theory the compiler should try each possibility. Unfortunately, this search may be too expensive. For this reason, HAL (like Mercury) chooses one schedulable mode and commits to it. This behavior might lead to the compiler failing to check a mode-correct procedure (see Example 27). To minimize this risk, we choose a schedulable mode whose success ti-state TI defined as $\{x_1 \mapsto \mathrm{RT}(t_1, s_1),\ldots,x_n \mapsto \mathrm{RT}(t_n, s_n)\}$ is minimal; that is, for each other schedulable mode with success ti-state TI' it is the case that $TI' \not\preceq TI$. Note that there may be more than one mode with a minimal success ti-state. In the case that we have more than one mode with the same minimal success state then we use a mode with a minimal call ti-state.

Example 15

Consider the scheduling of the goal `pop(A,B,C)` where current ti-state is $TI = \{A \mapsto r_1, B \mapsto \mathsf{new}, C \mapsto r_3\}$ where r_1 and r_3 are defined by the grammars

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

$$\begin{array}{lcl} r_1 & \rightarrow & [r_2 \mid r_3] \\ r_2 & \rightarrow & b \\ r_3 & \rightarrow & [] \end{array}$$

That is $A = [b]$ and $C = []$. Neither of the declared modes for `pop`, shown below, are immediately applicable.

```
:- mode pop(in,out,out) is semidet.
:- mode pop(in(nelist(ground)),out,out) is det.
```

But both modes fit the conditions for an implied mode. Since the second mode has a more specific success ti-state (the first argument is known to be non-empty) it is chosen. The resulting code is `pop_mode2(A,B,Fresh), Fresh = C`, where mode checking will then schedule the new equation appropriately. □

The idea is to maintain as much instantiation information as possible, thus restricting as little as possible the number of schedulable modes for the remaining literals. In our experience with compiling real programs this policy seems adequate to avoid any problems. It is straightforward, but in practice too expensive, to implement a complete search for all possible schedules.

5.4 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 behavior schedules goals as close to the programmer given 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 $\perp$ to any variable 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.

Example 16
Consider scheduling the goal

```
Y = [U1|U2], U2 = [], X = [U1|U3].
```

where `X` is initially `RT(list(T),ground)`, and the remaining variables are `new`. The first literal is not schedulable and will remain so until both `U1` and `U2` are no longer new. We consider then the second literal, which is schedulable as a construct, thus changing the type-instantiation of `U2` to `RT(list(T),elist)`. Since the first literal remains unschedulable, we consider the third literal which is schedulable as a deconstruct, thus changing the type-instantiation of `X`, `U1` and `U3` to `RT(list(T),nelist(ground))`, `RT(T,ground)`, and to `RT(list(T),ground)`, respectively. Since both `U1` and `U2` are no longer new, the first literal is now schedulable as a construct. The resulting code is

```
U2 := [], X =: [U1|U3], Y := [U1|U2].
```

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

In the final ti-state the instantiation of Y is given by the tree-grammar

$$\begin{array}{rcl} \texttt{Y} & \rightarrow & [ti(T, \mathsf{ground})|ti(\mathsf{list}(T), \mathsf{elist})] \\ ti(\mathsf{list}(T), \mathsf{elist}) & \rightarrow & [] \\ ti(T, \mathsf{ground}) & \rightarrow & \$\texttt{ground}(T)\$ \end{array}$$

in other words it is a list of length exactly one. □

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 $TI' = \bigvee_{j \in \{1..n\}} TI_j$. 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 $G_i \rightarrow G_t; G_e$ 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 $TI' = TI_t \vee TI_e$. If one of G_t or G_e is not schedulable or TI' includes $\top$ the whole if-then-else is not schedulable. Note that the analysis of $G_i \rightarrow G_t; G_e$ is identical to that of $(G_i, G_t); G_e$ except that all goals of G_i must be scheduled before those of G_t.

5.5 Mode declarations

In this subsection we discuss how mode-correctness is checked for each mode declaration.

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 $p(c_1 \rightarrow s_1, \ldots, c_n \rightarrow s_n)$, we build the initial ti-state $TI = \{x_1 \mapsto \textsc{rt}(t_1, c_1), \ldots, x_n \mapsto \textsc{rt}(t_n, c_n)\}$. The body of the predicate is then analyzed starting from the state TI. The mode declaration is correct if (a) everything is schedulable and (b) if the final ti-state is $TI' = \{x_1 \mapsto r'_1, \ldots, x_n \mapsto r'_n\}$, then for each argument variable $1 \leqslant i \leqslant n$, $r'_i \preceq \textsc{rt}(t_i, s_i)$. If the body is not schedulable or the resulting instantiations are not strong enough, a mode error results. Note that (a) ensures that the predicate is call mode-correct for that mode while (b) ensures that it is input-output mode-correct.

Example 17

Consider mode checking of the following code from Example 11 which makes use of the code in Figure 1:

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

We start by constructing the initial ti-state $TI = \{\texttt{S0} \mapsto gnelT, \texttt{S} \mapsto \mathsf{new}\}$ where $gnelT = \textsc{rt}(\texttt{list(T)}, \texttt{nelist(ground)})$ is the ti-grammar shown in Example 9. Checking the first disjunct (rule) we have `S0 = []` schedulable as a deconstruct. The resulting ti-state assigns $\bot$ to `S0`, and thus the whole conjunction is schedulable

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

with $TI_1 = \{\texttt{S0} \mapsto \bot, \texttt{S} \mapsto \bot\}$. Checking the second disjunct, we first extend TI to map `A` and `S1` to `new`. Examining the first literal `push(S0, A, S)` we find that it is not schedulable since `A` has instantiation `new` and is required to be `ground`. Examining the second literal `pop(S0, A, S1)` we find that both modes declared for `pop/3` are schedulable. Since the second mode has more specific success instantiations, it is chosen and the ti-grammars for `A` and `S1` become RT(T, `ground`) and RT(`list(T)`, `ground`), respectively. Now the first literal is schedulable obtaining for `S` the ti-grammar $gnel\,T$. Restricting to the original variables the final ti-state is $TI_2 = \{\texttt{S0} \mapsto gnel\,T, \texttt{S} \mapsto gnel\,T\}$. Taking the join $TI' = TI_1 \vee TI_2 = TI_2$. 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 with the second and first modes of the predicates, respectively.

Note that the HAL compiler's current mode analysis does not track variable dependencies and thus it may obtain a final type-instantiation state weaker than expected.

Example 18
Consider the solver type `habc/0` of Example 7. The following program does not pass mode checking:

```
:- pred p(list(habc), habc).
:- mode p(list(old) -> ground, in) is semidet.
p(L, E) :- L = [].
p(L, E) :- L = [E1|L1], E = E1, p(L1, E).
```

The first literal of the second rule is a deconstruct. After that deconstruct variable `L` is never touched and hence its instantiation is never updated; in particular it is not updated when the instantiation of `E1` and `L1` change. The inferred type-instantiation for `L` at the end of the second rule is thus RT(`list(habc)`, `nelist(old)`) rather than RT(`list(habc)`, `nelist(ground)`). Hence, mode checking fails. □

This could be overcome by adding a definite sharing analysis and/or a dependency based groundness analysis to the mode checking phase. Whenever a variable which definitely shares with another (through an equation e) is touched, we modify the resulting ti-state as if the equation e has been rescheduled to update sharing variables. This is (partially) implemented, for example, in the alias branch of the Mercury compiler.

6 Automatic initialization

As mentioned before, constraint solvers must provide an initialization procedure (`init/1`) for their solver type. This procedure takes a solver variable with instantiation `new` and returns it with instantiation `old`, after initializing whatever data-structures (if any) the solver needs.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

Many of the predicates exported by constraint solvers (including most constraints) require the solver variables appearing as arguments to be already initialized. Thus, explicit initializations for local variables may need to be introduced. Not only is this a tedious exercise for the programmer, 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. In particular, whenever a literal cannot be scheduled because there is a requirement for an argument of type t to be $\text{RT}(t, \texttt{old})$ when it is new and t is a solver type, then the `init/1` predicate for type t can be inserted to make the literal schedulable.

Example 19
Assume we have an integer solver with solver type `cint/0`.

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

where the predicate `+(X,Y,Z)` models $X+Y=Z$ and requires at least two arguments to be ground on call and all arguments are ground on return.

For the first mode `L = [X|L1]` cannot be scheduled as a construct until `X` has a ti-grammar different from new. Hence, `X` needs to be initialized. In the second mode `L = [X|L1]` can be scheduled as a deconstruct and thus no initialization is needed. The two resulting procedures are:

```
length_mode1(L, N) :- (L := [], N == 0
      ; +outinin(N1, 1, N), N > 0, length_mode1(L1, N1), init(X), L := [X|L1]).
length_mode2(L, N) :- (L == [], N := 0
      ; L =: [X|L1], length_mode2(L1, N1), +ininout(N1, 1, N), N > 0).
```

where we have rewritten the call to `+/3` to show the mode more clearly ($+_{outinin}$ indicates that the first argument is `out` and the rest are `in`, $+_{ininout}$ indicates that the third argument is `out` and the other arguments are `in`). □

Unfortunately, unnecessary initialization may slow down execution and introduce unnecessary variables (when it interacts with implied modes). Hence, we would like to only 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. If this fails it will start from the previous partial schedule looking for the leftmost unscheduled literal l which can be scheduled by initializing variables which (a) have either a solver type or a parameter type (e.g. $v \in V_{type}$) and (b) do not appear in an unscheduled literal to the left which equates them to a term (if so, chances are the equation will become a construct and no initialization is needed). If such an l is found the appropriate initialization calls are inserted before l, and then scheduling continues once more trying to schedule without initialization. If no l is found the whole conjunct is not schedulable. This two phase approach is applied at each conjunct level individually.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

Example 20
Consider the following program where `cint/0` is a solver type:

```
:- instdef evenlist(T) -> ([] ; [T|oddlist(T)]).
:- instdef oddlist(T) -> [T|evenlist(T)].

:- pred pairlist(list(cint),int).
:- mode pairlist(out(evenlist(old)),in) is nondet.
pairlist(L,N) :- N = 0, L = [].
pairlist(L,N) :- N > 0, +(N1,1,N), L = [V|L1], L1 = [V|L2], pairlist(L2,N1).
```

In the first phase all literals in the second rule are schedulable except `L = [V|L1]` and `L1 = [V|L2]` which can be neither a construct (`V`, `L1` and `L2` are new) nor a deconstruct (both `L` and `L1` are new). In the second phase we examine the two remaining unscheduled literals: the second literal can be scheduled by initializing `V`. Once this is done the first literal can be scheduled obtaining:

```
pairlist(L,N) :- N == 0, L := [].
pairlist(L,N) :- N > 0, +outinin(N1,1,N), pairlist(L2,N1),
                 init(V), L1 := [V|L2], L := [V|L1].
```

□

Many other different initialization heuristics could be applied. We are currently investigating more informed policies which give the right tradeoff between adding constraints as early as possible, and delaying constraints until they can become tests or assignments.

7 Higher-order objects

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 $n \geqslant k$. 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` literal supplies the $n - k$ arguments missing from the higher-order object h.

In order to represent types and instantiations for higher-order objects we need to extend the languages of type and instantiation expressions. The *higher-order type* of a higher-order object h constructed in the previous paragraph is of the form $pred(t_{k+1}, \ldots, t_n)$ where $pred/(n-k)$ is a new special type constructor 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 $pred(c_{k+1} \to s_{k+1}, \ldots, c_n \to s_n)$[12] where $pred/(n-k)$ is a new special instantiation construct and $c_j \to s_j$ give the

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

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

call and success instantiations of argument j respectively. It provides the modes of the $n-k$ arguments missing from h. Note that for the first time we allow `new` instantiations appearing inside instantiation expressions (since they will often be call instantiations). But their appearance is restricted to the outermost arguments of higher-order instantiations.

Now we must extend the $\text{RT}(t,i)$ operation to handle higher-order types and instantiations. Let us first consider the case in which i is the higher-order instantiation $pred(c_{k+1} \to s_{k+1}, \ldots, c_n \to s_n)$. If t is the higher-order type $pred(t_{k+1}, \ldots, t_n)$ then $\text{RT}(t,i)$ returns the grammar

$$ti(t,i) \quad \to \quad \$\texttt{ipred}\$(root(tc_{k+1}), root(ts_{k+1}), \ldots, root(tc_n), root(ts_n))$$

together with the grammars $tc_{k+1}, \ldots, tc_n, ts_{k+1}, \ldots, ts_n$ where $tc_j = \text{RT}(t_j, c_j)$ and $ts_j = \text{RT}(t_j, s_j)$. If t is not a higher-order type or has the wrong arity then $\text{RT}(t,i) = \top$, indicating an error. The new constant \$ipred\$ simply collects the call and success ti-grammars for the higher-order object's missing arguments.

The extension of $\text{RT}(t,i)$ for the case of base instantiations i is similar to the treatment of type parameters. A higher-order object can be `new` or `ground`, but if it is `old` this is identical to `ground` since higher-order objects never have an attached solver. $\text{RT}(pred(t_1, \ldots, t_n), \texttt{new})$ is treated as before (i.e. it creates a new ti-grammar). Similarly $\text{RT}(pred(t_1, \ldots, t_n), \texttt{ground})$ generates a production rule using a new constant \$gpred\$ of the form

$$ti(pred(t_1, \ldots, t_n), \texttt{ground}) \to \$\texttt{gpred}\$$$

$\text{RT}(pred(t_1, \ldots, t_n), \texttt{old})$ generates the same grammar (since it is equivalent). Since we will only compare the higher-order ti-grammar against other ti-grammars for the same type we can safely omit the information about the argument types $(t_1, \ldots, t_n)$.

The new constant \$gpred\$ acts like $\$\texttt{ground}(v)\$$ but it can also be compared with more complicated ti-grammars (with production rules for function symbol \$ipred\$) of the same type. The full code for $\text{RT}(t,i)$ is given in the appendix.

Example 21

Consider the following code:

```
:- pred map(pred(T1,T2), list(T1), list(T2)).
:- 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).

:- typedef sign -> (neg ; zero ; pos).

:- pred mult(sign, sign, sign).
:- mode mult(in, in, out) is det.

?- H1 = mult(pos), map(H1, [neg,zero,pos], L1).
```

The `map/3` predicate takes a higher-order predicate with two missing arguments of parametric types `T1` and `T2` and modes `in` and `out`, respectively. The ti-grammar describing the input instantiation of the first argument of `map/3` is the grammar with root $a_1 = ti(\texttt{pred(T1,T2)}, \texttt{pred(in,out)})$, defined by

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

$$
\begin{array}{rcl}
a_1 & \rightarrow & \texttt{\$ipred\$}(ti(\mathsf{T1},\mathsf{ground}),ti(\mathsf{T1},\mathsf{ground}),\mathsf{new},ti(\mathsf{T2},\mathsf{ground})) \\
ti(\mathsf{T1},\mathsf{ground}) & \rightarrow & \texttt{\$ground}(T1)\texttt{\$} \\
ti(\mathsf{T2},\mathsf{ground}) & \rightarrow & \texttt{\$ground}(T2)\texttt{\$} \\
\mathsf{new} & \rightarrow & \texttt{\#fresh\#}
\end{array}
$$

This predicate is applied to a list of `T1`s, returning a list of `T2`s. The literal `H1 = mult(pos)` constructs a higher-order object which multiplies the sign of its first argument by `pos`, returning the result in its second argument. The type-instantiation of `H1`, `RT(pred(sign,sign),pred(in,out))`, is the grammar with root $a_2 = ti(\mathsf{pred(sign,sign),pred(in,out)})$ and rules:

$$
\begin{array}{rcl}
a_2 & \rightarrow & \texttt{\$ipred\$}(ti(\mathsf{sign},\mathsf{ground}),ti(\mathsf{sign},\mathsf{ground}),\mathsf{new},ti(\mathsf{sign},\mathsf{ground})) \\
ti(\mathsf{sign},\mathsf{ground}) & \rightarrow & neg\ ;\ zero\ ;\ pos \\
\mathsf{new} & \rightarrow & \texttt{\#fresh\#}
\end{array}
$$

□

We need to extend the ordering $\preceq$ to higher-order type-instantiations as well as the operations $\wedge$ and $\vee$. Two higher-order ti-grammars r and r' defined with rules

$$root(r) \quad \rightarrow \quad \texttt{\$ipred\$}(xc_1, xs_1, \ldots, xc_n, xs_n)$$

and

$$root(r') \quad \rightarrow \quad \texttt{\$ipred\$}(xc'_1, xs'_1, \ldots, xc'_n, xs'_n)$$

satisfy $r \preceq r'$ iff for $i = 1,\ldots,n$, $subg(xc'_i, r') \preceq subg(xc_i, r)$ and $subg(xs_i, r) \preceq subg(xs'_i, r')$. Intuitively, if $r \preceq r'$, then any higher-order `call(`r'`,...)` should be replaceable by `call(`r`,...)`. For this to work, two conditions must be fulfilled. First, r must be able to deal with any values that r' can deal with (and perhaps more). Thus, $subg(xc'_i, r') \preceq subg(xc_i, r)$. And second, r must return the same values as r' or less. Thus, $subg(xs_i, r) \preceq subg(xs'_i, r')$. For more details see the example below.

We define $r \preceq$ `RT(pred(`$t_1,\ldots,t_n$`),ground)` for any ti-grammar r of the appropriate type except `new`. The full definition of $\preceq$ is given in the appendix. The $\wedge$ and $\vee$ operations follow naturally from the ordering, and are given in the appendix.

Example 22

Consider the following code and goal:

```
:- typedef abc -> a ; b ; c.
:- instdef ab -> a ; b.

:- pred ho1(abc,abc).
:- mode ho1(in(ab),out(ab)) is det.
ho1(A,B) :- A = B.

:- pred ho2(abc,abc).
:- mode ho2(in,out) is det.
ho2(A,B) :- A = a, B = b.
ho2(A,B) :- A = b, B = c.
ho2(A,B) :- A = c, B = a.

?- HO1 = ho1, HO2 = ho2, (HO = HO1 ; HO = HO2).
```

During scheduling of the disjunction, the ti-grammar for `HO1` is `RT(pred(abc,abc), pred(in(ab),out(ab)))`, i.e.:

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

```
   ho1  →  $ipred$(gndab, gndab, new, gndab)
gndab   →  a ; b
  new   →  #fresh#
```

and the ti-grammar for HO2 is

```
   ho2  →  $ipred$(gndabc, gndabc, new, gndabc)
gndab   →  a ; b ; c
  new   →  #fresh#
```

The abstract disjunction of these two grammars to build the ti-grammar for HO gives

```
   ho  →  $ipred$(gndab, gndabc, new, gndabc)
```

Notice the call ti-grammars have been abstractly conjoined. This illustrates the contravariant nature of calling instantiations of higher-order predicates. The higher-order object in HO can only be safely applied to an input a or b since it may be predicate ho1. It can only be guaranteed to give output a,b or c since it may be predicate ho2. □

7.1 Scheduling higher-order

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, we again choose one schedulable mode (using the same criteria used for calls to first order predicates) and commit to it. If it is not true for any mode, the equation is delayed until the arguments become more instantiated. Formally, let the current ti-state be $TI = \{h \mapsto r, x_1 \mapsto r_1, \ldots, x_k \mapsto r_k\} \cup RTI$ and the types $\{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$. Let the (declared or inferred) predicate type of p/n be $p(dt_1, \ldots, dt_n)$, then (because of type correctness) we have that there exists θ such that $\theta(dt_j) = t_j$.

Consider the declared mode $p(c_1 \to s_1, \ldots, c_n \to s_n)$. The higher-order equation is schedulable if $r = \mathsf{new}$ and for each $1 \leqslant j \leqslant k$, $r_j \preceq \mathrm{RT}(t_j, c_j) \ \wedge \ r_j \neq \mathsf{new}$. The resulting ti-state is

$$\{h \mapsto r', x_1 \mapsto r_1, \ldots, x_k \mapsto r_k\} \cup RTI.$$

where $tc_j = \mathrm{RT}(\theta(dt_j), c_j)$ and $ts_j = \mathrm{RT}(\theta(dt_j), s_j)$ for $k+1 \leqslant j \leqslant n$ and $r' = \{a \to \$\mathtt{ipred}\$(root(tc_{k+1}), root(ts_{k+1}), \ldots, root(tc_n), root(ts_n))\}$, where a is a new non-terminal, together with the grammars for $tc_{k+1}, ts_{k+1}, \ldots, tc_n, ts_n$.

Note that the instantiation of each x_j is unchanged and, in fact, will not be updated even when h is called. This is because in general we cannot ensure when or if the call has actually been made. As a result, mode checking with higher-order objects can be imprecise. In particular, if one of the r_j is new we may not know if it becomes initialised or not since we do not know if the call to h which will initialise it has been made. Since we must be able to precisely track when a variable has become initialised, we do not allow a call to be scheduled if this is the case (hence the $r_j \neq \mathsf{new}$ condition above).

A higher-order call $\mathtt{call}(h, x_{k+1}, \ldots, x_n)$ 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

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

type-instantiation previously assigned to h. If this is not true, the call is delayed until the arguments become more instantiated. Formally, let the current ti-state be $TI = \{h \mapsto r, x_{k+1} \mapsto r_{k+1}, \ldots, x_n \mapsto r_n\} \cup RTI$. The call is schedulable if r has a production rule of the form

$$root(r) \rightarrow \$\texttt{ipred}\$(xc_{k+1}, xs_{k+1}, \ldots, xc_n, xs_n)$$

and for each $j \in k+1, ..., n$, $r_j \preceq subg(xc_j, r)$. The resulting instantiation is $TI' = \{h \mapsto r, x_{k+1} \mapsto r_{k+1} \wedge subg(xs_{k+1}, r), \ldots, x_n \mapsto r_n \wedge subg(xs_n, r)\} \cup RTI$. Just as for normal predicate calls, implied modes are also possible where if, for example, xc_l is new, we can replace x_l with a fresh variable $fresh_l$ and a following equation $fresh_l = x_l$. And, if necessary, the mode checker will add calls to initialise solver variables.

Example 23
Consider the following code and assume all goals are schedulable in the order written:

```
:- instdef only_a -> a.
:- modedef abc2a -> (ground -> only_a).

:- pred p(abc, abc, abc)
:- mode p(abc2a, in, out(only_a)) is semidet.

?- G1, p(A,B,C), G2.
?- G1, H = p(A), call(H,B,C), G2.
```

The two queries would appear to have identical effects. However, mode checking for the second goal will not determine that the instantiation for `A` becomes `only_a` by the time it reaches goal G_2. Assuming `A` was `ground` before `H = p(A)`, then the type-instantiation of `H` is the grammar with root $x = ti(pred(\mathsf{abc}, \mathsf{abc}), pred(\mathsf{in}, \mathsf{out}(\mathsf{only\_a})))$ and rules:

$$\begin{aligned} x &\rightarrow \$\texttt{ipred}\$(ti(\mathsf{abc}, \mathsf{ground}), ti(\mathsf{abc}, \mathsf{ground}), \mathsf{new}, ti(\mathsf{abc}, \mathsf{only\_a})) \\ ti(\mathsf{abc}, \mathsf{ground}) &\rightarrow a \ ; \ b \ ; \ c \\ ti(\mathsf{abc}, \mathsf{only\_a}) &\rightarrow a \\ \mathsf{new} &\rightarrow \#\texttt{fresh}\# \end{aligned}$$

Of course in this case it is obvious that the predicate is being called before G_2, and so it could be inferred that the instantiation of `A` was only_a at that point. However, in the usual case such analysis is harder, since the construction of a higher-order term and its eventual execution are usually performed in different predicates. Indeed, in general it is impossible to know at compile time whether at a given program point the higher-order predicate has been executed or not. □

8 Polymorphism and modes

Polymorphic predicates are very useful because they can be used for different types. Unfortunately, mode information can be lost since only the base instantiations `ground`, `new`, and `old` can be associated with type parameters.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

Example 24
Consider the interface to the stack data type defined in Figure 1 and the following program:

```
:- pred q(abc).
:- mode q(in) is semidet.
:- mode q(in(only_a)) is det.

?- empty(S0), I0 = a, push(S0, I0, S1), pop(S1, I, S2), q(I).
```

Although list `S1` is indeed a list only containing items `a` this information is lost after executing `push` since the output instantiation declared for this predicate is simply ground. Because of this, the first mode of predicate `q/1` will be selected for literal `q(I)`, thus losing the information that `q(I)` could not fail. □

This loss of instantiation information for arguments to polymorphic predicates may have severe consequences 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 25
Consider the following goal using code from Figure 1 and Example 21:

```
?- empty(S0), I0 = mult(pos), push(S0,I0,S1), pop(S1,I,S2), map(I,[neg],S).
```

When item `I` is extracted from the list its ti-grammar is $\text{RT}(t, \mathsf{ground})$ where t is type $pred(\mathsf{sign}, \mathsf{sign})$. As a result, it cannot be used in `map` since its mode and determinism information has been lost, i.e. the check $\text{RT}(t, \mathsf{ground}) \preceq \text{RT}(t, \mathsf{pred(in,out)})$ fails. □

We could overcome the above problem by having a special version of each stack predicate to handle the higher-order predicate case. But this requires modifying the `stack` module, defeating the idea of an abstract data type. Also, this modification is required for each mode of the higher-order object that the programmer wishes to make use of. Clearly, this is not an attractive proposition.

Our approach is to use polymorphic type information to recover the lost mode information. This is an example of "Theorems for Free" (Wadler 1989)): since the polymorphic code can only equate 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.

8.1 Polymorphic mode checking

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. Consider the predicate call $p(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are variables with type $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ and current ti-state $TI = \{x_1 \mapsto r_1, \ldots, x_n \mapsto r_n\} \cup RTI$. Suppose the predicate type declared (or inferred) for p is $p(dt_1, \ldots, dt_n)$. Note that because of type correctness there exists the type substitution θ where $\theta(dt_j) = t_j$.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

```
COLLECT_SET(r1,r2,P)
  x1 := root(r1); x2 := root(r2)
  if ((x1, x2) ∈ P) return ∅
  if (x1 = new) return ∅
  if (x1 → $old(v)$ ∈ r1) return {(old, v, r2)}
  if (x1 → $ground(v)$ ∈ r1) return {(ground, v, r2)}
  M := ∅
  foreach rule x1 → f(x11, ..., x1n) in r1
    if exists rule x2 → f(x21, ..., x2n) in r2
      for i := 1..n
        M := M ∪ COLLECT_SET(subg(x1i, r1), subg(x2i, r2), P ∪ {(x1, x2)})
  return M
```

Fig. 3. Algorithm for collecting the type-instantiations that match type parameters.

Assume the literal is schedulable for mode declaration $p(c_1 \rightarrow s_1, \ldots, c_n \rightarrow s_n)$. We proceed by matching the ti-trees $\text{RT}(dt_j, c_j)$ against the current instantiations r_j in a process analogous to the matching that occurs in the meet function. Note that $\text{RT}(dt_j, c_j)$ is the ti-grammar which contains information on the positions of type parameters in the declared type of p.

Consider the function $\text{COLLECT\_SET}(r_1, r_2, \emptyset)$, defined in Figure 3, which returns the set of triples $(\texttt{old}, v, r')$ and $(\texttt{ground}, v, r')$ obtained by collecting each ti-grammar, r', in r_2 matching occurrences of $\$\texttt{old}(v)\$$ and $\$\texttt{ground}(v)\$$ in r_1. Let $M = \cup_{j=1}^{n}$ $\text{COLLECT\_SET}(\text{RT}(dt_j, c_j), r_j, \emptyset)$. We will use this information to compute the success instantiations as follows: since the only success type-instantiation information for elements of parametric type v must come from its call type-instantiations, we can safely assume that any success type-instantiation is at least as instantiated as the join (upper bounds) of the calls.

Note that when determining ground success information, we need only consider ground calling instantiations, since ground success instantiations cannot result from old call instantiations. On the other hand, for old success information, we need to consider both old and ground calling instantiations, since old success instantiations can result from either. Hence the following definitions for $ground(v, M)$ and $old(v, M)$, which compute upper bounds on success instantiations for v based on the call instantiation information collected in M:

$$\begin{aligned} ground(v, M) &= \bigvee\{r \mid (\texttt{ground}, v, r) \in M\} \\ old(v, M) &= \bigvee\{r \mid (\texttt{ground}, v, r) \in M \text{ or } (\texttt{old}, v, r) \in M\} \end{aligned}$$

Because the literal is schedulable for the given mode we know that no r_i contains new for any t. Thus, the abstract disjunctions in $ground(v, M)$ and $old(v, M)$ never lead to $\top$.

Let ps_j be the result of replacing in $\text{RT}(dt_j, s_j)$ each non-terminal x with productions of the form

$$x \rightarrow \$\texttt{ground}(v)\$ \ ; \ \$\texttt{old}(v)\$$$

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

by $root(old(v, M))$ and removing the rules for x, and replacing each non-terminal x with productions of the form

$$x \rightarrow \$\texttt{ground}(v)\$$$

by $root(ground(v, M))$ and removing the rules for x, and finally adding the rules in $old(v, M)$ and $ground(v, M)$. The new ti-state resulting after scheduling the polymorphic literal is $TI' = \{x_1 \mapsto r_1 \wedge ps_1, \ldots, x_n \mapsto r_n \wedge ps_n\} \cup RTI$.

Example 26
Assume we are scheduling the `push/3` literal in the goal using code from Figure 1 and Example 21:

```
?- empty(S0), I0 = mult(pos), push(S0,I0,S1), pop(S1,I,S2), map(I,[neg],S).
```

for current ti-state $\{\texttt{S0} \mapsto r_3, \texttt{I0} \mapsto r_4\}$, the remaining variables being new, where r_3 is the grammar

$$ti(\mathsf{list(sign)}, \mathsf{elist}) \quad \rightarrow \quad []$$

and r_4 is the grammar with root $a = ti(pred(\mathsf{sign}, \mathsf{sign}), pred(\mathsf{in}, \mathsf{out}))$ defined by

$$\begin{array}{rcl} a & \rightarrow & \$\texttt{ipred}\$(ti(\mathsf{sign}, \mathsf{ground}), ti(\mathsf{sign}, \mathsf{ground}), \mathsf{new}, ti(\mathsf{sign}, \mathsf{ground})) \\ ti(\mathsf{sign}, \mathsf{ground}) & \rightarrow & neg \ ; \ zero \ ; \ pos \end{array}$$

The ti-grammars defined by the declared type and mode declarations for the first two arguments of `push/3` are: $r_5 = \text{RT}(\texttt{list(T)}, \texttt{ground})$ or the grammar

$$\begin{array}{rcl} ti(\mathsf{list(T)}, \mathsf{ground}) & \rightarrow & [] \ ; \ [ti(\mathsf{T}, \mathsf{ground}) \mid ti(\mathsf{list(T)}, \mathsf{ground})] \\ ti(\mathsf{T}, \mathsf{ground}) & \rightarrow & \$\texttt{ground}(T)\$ \end{array}$$

and, $r_6 = \text{RT}(\mathsf{T}, \mathsf{ground})$, the grammar

$$ti(\mathsf{T}, \mathsf{ground}) \quad \rightarrow \quad \$\texttt{ground}(T)\$$$

The literal is schedulable and the matching process determines that $\text{COLLECT\_SET}(r_5, r_3) = \emptyset$, $\text{COLLECT\_SET}(r_6, r_4) = \{(\mathsf{ground}, \mathsf{T}, r_4)\}$ and $M = \{(\mathsf{ground}, \mathsf{T}, r_4)\}$. The improved analysis determines that extra success instantiation (ps_3) for the third argument (`S1`) by improving $\text{RT}(\texttt{list(T)}, \texttt{nelist(ground)})$ which is

$$\begin{array}{rcl} ti(\mathsf{list(T)}, \mathsf{nelist(ground)} & \rightarrow & [ti(\mathsf{T}, \mathsf{ground}) \mid ti(\mathsf{list(T)}, \mathsf{list(ground)}] \\ ti(\mathsf{list(T)}, \mathsf{list(ground)} & \rightarrow & [] \ ; \ [ti(\mathsf{T}, \mathsf{ground}) \mid ti(\mathsf{list(T)}, \mathsf{list(ground)}] \\ ti(\mathsf{T}, \mathsf{ground}) & \rightarrow & \$\texttt{ground}(T)\$ \end{array}$$

replacing the last rule by r_4 and occurrences of $ti(\mathsf{list(T)}, \mathsf{list(ground)}$ by $root(r_4) = a$ obtaining

$$\begin{array}{rcl} ti(\mathsf{list(T)}, \mathsf{nelist(ground)} & \rightarrow & [a \mid ti(\mathsf{list(T)}, \mathsf{list(ground)}] \\ ti(\mathsf{list(T)}, \mathsf{list(ground)} & \rightarrow & [] \ ; \ [a \mid ti(\mathsf{list(T)}, \mathsf{list(ground)}] \\ a & \rightarrow & \$\texttt{ipred}\$(ti(\mathsf{sign}, \mathsf{ground}), ti(\mathsf{sign}, \mathsf{ground}), \mathsf{new}, ti(\mathsf{sign}, \mathsf{ground})) \\ ti(\mathsf{sign}, \mathsf{ground}) & \rightarrow & neg \ ; \ zero \ ; \ pos \end{array}$$

Note that the mode information of the higher-order term has been preserved. The mode checking for the call to `pop/3` will similarly preserve the higher-order mode information, and the original goal will be schedulable. □

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

The interaction between polymorphic mode analysis and higher-order constructs and calls is in fact slightly more complicated than discussed previously. This is because higher-order objects allow us to give arguments to a predicate in a piecewise manner. This affects the execution of COLLECT_SET which was collecting the set M over all predicate arguments simultaneously. In order to handle these accurately we need to store the information from M found during the higher-order object construction, to be used in the higher-order call. That is, we need to store $ground(v, M)$ and $old(v, M)$ for each type parameter v appearing in the remaining arguments as part of the ti-grammar for the higher-order object.

An alternative approach used by the HAL compiler is to update the success instantiations stored in the ti-grammar of the higher-order object based on the extra information from polymorphism. When the call to the higher-order polymorphic predicate is analyzed, the matching process also matches the success instantiations of the higher-order object to recover the previous matching information.

9 Conclusions and future work

The ultimate aim of mode checking is to ensure that the compiler has correct instantiation information at every program point to allow program optimization. It is reasonably straightforward (but laborious) to show that the mode checking defined in this paper ensures that the resulting program has input-output and call correctness. Some subtle points that arise are as follows. First, it is an invariant that any ti-grammar (or sub-grammar) r occurring in the mode checking process that contains rule $root(r) \rightarrow \#\texttt{var}\#$ must be equivalent to $\text{RT}(t, \texttt{old})$ for some t, which means that when variables are bound indirectly (through shared variables) the correctness of the ti-state is maintained. Second, if a procedure is input-output correct for its declared type, then it is also input-output correct for any instance of the type. This follows from the limited possibilities for manipulating objects of variable type (essentially copying and testing equality).

This means that compiler optimizations can safely be applied. The only mode error that may be detected at run-time arises from situations explained in section 5.2 and Example 13.[13] The compiler emits warnings when such a possibility exists.

We have described for the first time mode checking for CLP languages, such as HAL, which have strong typing and re-orderable clause bodies, and described the algorithms currently used in the HAL compiler. The actual implementation of these algorithms in the HAL compiler 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. For the compiler compiling itself (29,000 lines of HAL code in 27 highly interdependent modules compiled in 15 mins 20 secs) mode checking requires 16.4% of overall

[13] Note this does not invalidate the input-output or call correctness for the remainder of the program.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

compile time. While compiling the libraries (4600 lines of HAL code in 12 almost independent modules compiled in 47 secs) it takes 13.1% of overall compile time. And compiling a suite of small to medium size benchmarks (6200 lines of HAL code in 67 modules compiled in 183 secs) it takes 13.0% of overall compile time,

There is considerable scope for future work. One aim is to strengthen mode checking. We plan to add tracking of aliasing and groundness dependencies. Another problem is that currently HAL (like Mercury) never undoes a feasible choice of ordering the literals. This can lead to correctly moded programs not being checkable as in Example 27. In practice this behavior is rare, but we would like to explore more complete strategies.

Example 27
Consider the following declarations and goal:

```
:- pred p(list(int),list(int)).
:- mode p(out,out) is det.
:- mode p(in(evenlist(ground)),out(evenlist(ground))) is det.

:- pred q(list(int)).
:- mode q(out(evenlist(ground))) is det.

:- pred r(list(int)).
:- mode r(in(evenlist(ground))) is det.

?- p(L0, L1), q(L0), r(L1).
```

The first two literals of the goal are schedulable in the order given, as `p_mode1(L0, L1)`, `q_mode1(L2)`, `L2 = L0` but then `r(L1)` is not schedulable (the list `L1` may not be of even length). There is a feasible schedule: `q_mode1(L0)`, `p_mode2(L0, L1)`, `r_mode1(L1)` which is missed by both HAL and Mercury, since they don't undo the feasible schedule for the first two literals. In order to avoid this problem HAL allows the user to name modes of a predicate and hence specify exactly which mode is required. □

A second aim is to improve the efficiency of the reordered code, by, for instance, reducing the number of initializations. The final aim is to provide mode inference as well as mode checking – the ability to reorder body literals makes this a potentially very expensive process.

Acknowledgements

We would like to thank Fergus Henderson and Zoltan Somogyi for discussions of the Mercury mode system, and their help in modifying Mercury to support HAL features. We would also like to thank the anonymous referees whose suggestions have enormously improved the paper.

Appendix A Algorithms

In this appendix we give full versions of the tree operations mentioned in the paper. The basic tree operations are relatively straightforward, but new kinds of nodes for solver variables,

polymorphic types and higher-order terms complicate this somewhat. Recall that we assume we are dealing with type correct programs, hence the operations make use of this to avoid many redundant comparisons. For example when comparing the order of two ti-grammars, then if one is a predicate type, the other must be an identical predicate type.

The ordering relation $r_1 \preceq r_2$ on two ti-grammars is defined as the result of $\text{LT}(r_1, r_2, \emptyset)$.

```
LT(p1, p2, P)
  if (p2 = ⊤) return true
  if (p1 = ⊤) return false
  if ((root(p1), root(p2)) ∈ P) return true
  if (p2 = new and p1 ≠ new) return false
  case:
    p1 = new: return (p2 = new)
    root(p1) → $old(v)$ ∈ p1:          %% p1 = BASE(v, old)
      return root(p2) → $old(v)$ ∈ p2
    root(p1) → $ground(v)$ ∈ p1:       %% p1 = BASE(v, ground)
      return root(p2) → $ground(v)$ ∈ p2
    root(p1) → $gpred$ ∈ p1:           %% p1 = BASE(pred(t1,...,tn), ground)
      return root(p2) → $gpred$ ∈ p2
    root(p1) → $ipred$(tc1, ts1, ..., tcn, tsn) ∈ p1:   %% non-base higher-order ti
      if (root(p2) → $gpred$ ∈ p2) return true
      let root(p2) → $ipred$(tc'1, ts'1, ..., tc'n, ts'n) ∈ p2
      for i := 1..n
        if (¬LT(subg(tc'i, p2), subg(tci, p1), P ∪ {(root(p1), root(p2))})) return false
        if (¬LT(subg(tsi, p1), subg(ts'i, p2), P ∪ {(root(p1), root(p2))})) return false
      endfor
      return true
    default:
      foreach root(p1) → f(x1, ..., xn) ∈ p1
        if (∃root(p2) → f(x'1, ..., x'n) ∈ p2)
          for i := 1..n
            if (¬LT(subg(xi, p1), subg(x'i, p2), P ∪ {(root(p1), root(p2))})) return false
          endfor
        else return false
      endfor
      return true
```

The abstract conjunction operation $r_1 \wedge r_2$ on two ti-grammars is defined as the first element of the pair returned by $\text{CONJ}(r_1, r_2, \emptyset)$.

```
CONJ(p1,p2,P)
  if (p1 = ⊤) return (⊤, _)
  if (p2 = ⊤) return (⊤, _)
  if (p2 = new) return (p1, root(p1))
  if (meet(root(p1), root(p2)) ∈ P) return (∅, meet(root(p1), root(p2)))
  case:
    p1 = new: return (p2, root(p2))
    root(p1) → $old(v)$ ∈ p1:          %% p1 = BASE(v, old)
      return (p2,root(p2))
    root(p1) → $ground(v)$ ∈ p1:       %% p1 = BASE(v, ground)
      return (p1,root(p1))
    root(p1) → $gpred$ ∈ p1:           %% p1 = BASE(pred(t1,...,tn), ground)
      return (p2,root(p2))
    root(p1) → $ipred$(tc1, ts1, ..., tcn, tsn) ∈ p1:   %% non-base higher-order ti
```

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

if $(root(p_2) \rightarrow \$\texttt{gpred}\$ \in p_2)$ **return** $(p_1, root(p_1))$
let $root(p_2) \rightarrow \$\texttt{ipred}\$(xc'_1, xs'_1, \ldots, xc'_n, xs'_n) \in p_2$
for $i := 1..n$
$(tc_i, xc''_i) := \text{DISJ}(subg(xc_i, p_1), subg(xc'_i, p_2), P)$
$(ts_i, xs''_i) := \text{CONJ}(subg(xs_i, p_1), subg(xs'_i, p_2), P)$
if $(tc_i = \top$ or $ts_i = \top)$ **return** $(\top, \_)$
endfor
$p := \{meet(root(p_1), root(p_2)) \rightarrow \$\texttt{ipred}\$(xc''_1, xs''_1, \ldots, xc''_n, xs''_n)\} \cup$
$tc_1 \cup \cdots \cup tc_n \cup ts_1 \cup \cdots \cup ts_n$
return $(p, meet(root(p_1), root(p_2)))$
default:
$p := \emptyset$
foreach $root(p_1) \rightarrow f(x_1, \ldots, x_n) \in p_1$
if $(\exists root(p_2) \rightarrow f(x'_1, \ldots, x'_n) \in p_2)$
for $i := 1..n$
$(p''_i, x''_i) := \text{CONJ}(subg(x_i, p_1), subg(x'_i, p_2), P \cup \{meet(root(p_1), root(p_2))\}))$
if $(p''_i = \top)$ **return** $(\top, \_)$
endfor
$p := meet(root(p_1), root(p_2)) \rightarrow f(x''_1, \ldots, x''_n) \cup p \cup p''_1 \cup \cdots \cup p''_n$
endfor
return $(p, meet(root(p_1), root(p_2)))$

The abstract disjunction operation $r_1 \vee r_2$ on two ti-grammars, is defined as the first element of the pair returned by $\text{DISJ}(r_1, r_2, \emptyset)$.

DISJ(p_1,p_2,P)
if $(p_1 = \top)$ **return** $(\top, \_)$
if $(p_2 = \top)$ **return** $(\top, \_)$
if $(p_1 = \texttt{new}$ and $p_2 = \texttt{new})$ **return** $(\{\texttt{new} \rightarrow \#\texttt{fresh}\#\}, \texttt{new})$
if $(p_2 = \texttt{new})$ **return** $(\top, \_)$
if $(\exists join(root(p_1), root(p_2)) \in P)$ **return** $(\emptyset, join(root(p_1), root(p_2)))$
case:
$p_1 = \texttt{new}$: **return** $(\top, \_)$
$root(p_1) \rightarrow \$\texttt{old}(v)\$ \in p_1$: %% $p_1 = \text{BASE}(v, \texttt{old})$
return $(p_1, root(p_1))$
$root(p_1) \rightarrow \$\texttt{ground}(v)\$ \in p_1$: %% $p_1 = \text{BASE}(v, \texttt{ground})$
return $(p_2, root(p_2))$
$root(p_1) \rightarrow \$\texttt{gpred}\$ \in p_1$: %% $p_1 = \text{BASE}(pred(t_1, \ldots, t_n), \texttt{ground})$
return $(p_1, root(p_1))$
$root(p_1) \rightarrow \$\texttt{ipred}\$(xc_1, xs_1, \ldots, xc_n, xs_n) \in p_1$: %% non-base higher-order ti
if $(root(p_2) \rightarrow \$\texttt{gpred}\$ \in p_2)$ **return** $(p_2, root(p_2))$
let $root(p_2) \rightarrow \$\texttt{ipred}\$(xc'_1, xs'_1, \ldots, xc'_n, xs'_n) \in p_2$
for $i := 1..n$
$(tc_i, xc''_i) := \text{CONJ}(subg(xc_i, p_1), subg(xc'_i, p_2), P)$
$(ts_i, xs''_i) := \text{DISJ}(subg(xs_i, p_1), subg(xs'_i, p_2), P)$
if $(tc_i = \top$ or $ts_i = \top)$ **return** $(\top, \_)$
endfor
$p := \{join(root(p_1), root(p_2)) \rightarrow \$\texttt{ipred}\$(xc''_1, xs''_1, \ldots, xc''_n, xs''_n)\} \cup$
$tc_1 \cup \cdots \cup tc_n \cup ts_1 \cup \cdots \cup ts_n$
return $(p, join(root(p_1), root(p_2)))$
default:
$p := \emptyset$
foreach $root(p_1) \rightarrow f(x_1, \ldots, x_n) \in p_1$
if $(\exists root(p_2) \rightarrow f(x'_1, \ldots, x'_n) \in p_2)$

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

for $i := 1..n$
$(p_i'', x_i'') := \text{DISJ}(subg(x_i, p_1), subg(x_i', p_2), P \cup \{join(root(p_1), root(p_2))\})$
if $(p_i'' = \top)$ **return** $(\top, \_)$
endfor
$p := \{join(root(p_1), root(p_2)) \rightarrow f(x_1'', \ldots, x_n'')\} \cup p \cup p_1'' \cup \cdots \cup p_n''$
else
$p := \{join(root(p_1), root(p_2) \rightarrow f(x_1, \ldots, x_n)\} \cup p \cup$
$subg(x_1, p_1) \cup \cdots \cup subg(x_n, p_1)$
endif
endfor
foreach $root(p_2) \rightarrow f(x_1', \ldots, x_n') \in p_2$
if $(\neg\exists root(p_1) \rightarrow f(x_1, \ldots, x_n) \in p_1)$
$p := \{join(root(p_1), root(p_2) \rightarrow f(x_1', \ldots, x_n')\} \cup p \cup$
$subg(x_1', p_2) \cup \cdots \cup subg(x_n', p_2)$
endfor
return $(p, join(root(p_1), root(p_2)))$

The RT operation constructs a ti-grammar from a type t and instantiation i and is defined as the first element in the pair resulting from $\text{RT}(t, i, \emptyset)$.

RT(t,i,P)
if $(\exists ti(t, i) \in P)$ **return** $(\emptyset, ti(t, i))$
case:
i is a base instantiation: **return** BASE(t,i,P)
$i = pred(c_1 \rightarrow s_1, \ldots, c_n \rightarrow s_n)$:
if $(t \neq pred(t_1, \ldots, t_n))$ **return** $(\top, \_)$
let t be of the form $pred(t_1, \ldots, t_n)$
for $j = 1..n$
$(tc_j, xc_j) := \text{RT}(t_j, c_j, P)$
$(ts_j, xs_j) := \text{RT}(t_j, s_j, P)$
if $(tc_j = \top$ or $ts_j = \top)$ **return** $(\top, \_)$
endfor
$r := \{ti(t, i) \rightarrow \$\texttt{ipred}\$(xc_1, xs_1, \ldots, xc_n, xs_n)\} \cup$
$tc_1 \cup \cdots \cup tc_n \cup ts_1 \cup \cdots \cup ts_n$
return $(r, ti(t, i))$
default:
if $(t \in V_{type})$ **return** $(\top, \_)$
$r := \emptyset$
foreach $x \rightarrow f(x_{i1}, \ldots, x_{in}) \in rules(i)$
if $(\exists x' \rightarrow f(x_{t1}, \ldots, x_{tn}) \in rules(t))$
for $j = 1..n$
$(r_j, x_j) := \text{RT}(x_{tj}, x_{ij}, P \cup \{ti(x_{tj}, x_{ij})\})$
if $(r_j = \top)$ **return** $(\top, \_)$
endfor
$r := \{ti(t, i) \rightarrow f(x_1, \ldots, x_n)\} \cup r \cup r_1 \cup \cdots \cup r_n$
endif
endfor
return $(r, ti(t, i))$

BASE(t,$base$,P)
if $(base = \texttt{new})$ **return** $(\{\texttt{new} \rightarrow \#\texttt{fresh}\#\}, \texttt{new})$
if $(ti(t, base) \in P)$ **return** $(\emptyset, ti(t, base))$
if $(t \in V_{type})$:
if $(base = \texttt{ground})$ **return** $(\{ti(t, \texttt{ground}) \rightarrow \$\texttt{ground}(t)\$\}, ti(t, \texttt{ground}))$

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

else return $(\{ti(v, \texttt{old}) \rightarrow \$\texttt{ground}(v)\$ \ ; \ \$\texttt{old}(v)\$\}, ti(v, \texttt{old}))$
else if (t is of the form $pred(t_1, \ldots, t_n)$)
return $(\{ti(pred(t_1, \ldots, t_n), \texttt{ground}) \rightarrow \$\texttt{gpred}\$\}, \ ti(pred(t_1, \ldots, t_n), \texttt{ground}) \ )$
else
$r := \emptyset$
foreach $x \rightarrow f(t_1, \ldots, t_n)$ in $rules(t)$
for $j \in 1..n$
$(r_j, x_j) :=$ BASE$(t_j, base, P \cup \{ti(t, base)\})$
endfor
$r := \{ti(t, base) \rightarrow f(x_1, \ldots, x_n)\} \cup r \cup r_1 \cup \cdots \cup r_n$
endfor
if ($base = \texttt{old}$ and t is a solver type) **then** $r := \{ti(t, base) \rightarrow \#\texttt{var}\#\} \cup r$
return $(r, ti(t, base))$

References

Boye, J. and Małuszyński, J. 1997. Directional types and the annotation method. *J. Logic Programming 33,* 3, 179–220.

Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P. and Puebla, G. 2002. The Ciao Prolog system reference manual. Technical report, Technical University of Madrid (UPM). (http://clip.dia.fi.upm.es/Software/Ciao.)

Charlier, B. L. and Hentenryck, P. V. 1994. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems 16,* 1, 35–101.

Codish, M. and Lagoon, V. 2000. Type dependencies for logic programs using ACI-unification. *Theoretical Computer Science 238,* 1–2, 131–159.

Codognet, C., Codognet, P., and Corsini, M. 1990. Abstract interpretation of concurrent logic languages. *North American Conference on Logic Programming*. pp. 215–232.

Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S. and Tommasi, M. 1997. Tree automata techniques and applications. Available on: (`http://www.grappa.univ-lille3.fr/tata`.)

Debray, S. K. 1989. Static inference of modes and data dependencies in logic programs. *ACM Transactions on Programming Languages and Systems 11,* 3, 418–450.

Demoen, B. García de la Banda, M., Harvey, W., Marriott, K., and Stuckey, P. 1999a. Herbrand constraint solving in HAL. In: *Logic Programming: Proceedings of the 16th International Conference*, D. D. Schreye, Ed. MIT Press, pp. 260–274.

Demoen, B., García de la Banda, M., Harvey, W., Marriott, K. and Stuckey, P. 1999b. An overview of HAL. In: *Proceedings of the Fourth International Conference on Principles and Practices of Constraint Programming*, J. Jaffar, Ed. LNCS. Springer-Verlag, pp. 174–188.

Demoen, B., García de la Banda, M., and Stuckey, P. 1999. Type constraint solving for parametric and ad-hoc polymorphism. In: *Proceedings of the 22nd Australian Computer Science Conference*, J. Edwards, Ed. Springer-Verlag, pp. 217–228.

García de la Banda, M., Demoen, B. Marriott, K., and Stuckey, P. 2002. To the gates of HAL: a HAL tutorial. *Proceedings of the Sixth International Symposium on Functional and Logic Programming*. LNCS 2441, Springer-Verlag, pp. 47–66.

Gecseg, F. and Steinby, M. 1984. *Tree Automata*. Academei Kaido, Budapest.

Henderson, F., Somogyi, Z. and Conway, T. 1996. Determinism analysis in the Mercury compiler. *Proceedings of the Australian Computer Science Conference*. pp. 337–346.

https://doi.org/10.1017/S1471068404002327 Published online by Cambridge University Press

Hermenegildo, M., Puebla, G., Bueno, F. and López-García, P. 2003. Program development using abstract interpretation (and the Ciao system preprocessor). In: *Proceedings of the 10th International Symposium on Static Analysis*, R. Cousot, Ed. LNCS 2694, Springer-Verlag, pp. 127–152.

Janssens, G. and Bruynooghe, M. 1993. Deriving descriptions of possible values of program variables by means of abstract interpretation. *J. Logic Programming 13*, 205–258.

Marriott, K. and Stuckey, P. 1992. The 3 R's of optimizing constraint logic programs: Refinement, removal, and reordering. *19th. Annual ACM Conference on Principles of Programming Languages*. ACM, pp. 334–344.

Marriott, K. and Stuckey, P. 1998. *Programming with Constraints: an Introduction*. MIT Press.

Mellish, C. 1987. Abstract Interpretation of Prolog Programs. *Abstract Interpretation of Declarative Languages*, 181–198.

Mulkers, A., Simoens, W., Janssens, G. and Bruynooghe, M. 1995. On the Practicality of Abstract Equation Systems. *International Conference on Logic Programming*. MIT Press, pp. 781–795.

Mycroft, A. 1984. Polymorphic type schemes and recursive definitions. *International Symposium on Programming*. LNCS 167, Springer-Verlag, pp. 217–228.

Okasaki, C. 1998. *Purely Functional Data Structures*. Cambridge University Press.

Overton, D. 2003. Precise and expressive mode systems for typed logic programming languages. PhD thesis, Department of Computer Science and Software Engineering, University of Melbourne. (http://www.cs.mu.oz.au/research/mercury/information/papers/dmo-thesis.ps.gz.)

Ridoux, O., Boizumault, P. and Malesieux, F. 1999. Typed static analysis: Application to groundness analysis of prolog and lambda-prolog. In *Proceedings of the International Symposium on Functional and Logic Programming*. LNCS, Springer-Verlag, pp. 267–283.

Smaus, J.-G., Hill, P. and King, A. 2000. Mode analysis domains for typed logic programs. In: *Proceedings of the 9th International Workshop on Logic-based Program Synthesis and Transformation*, A. Bossi, Ed. LNCS, Springer-Verlag, pp. 82–101.

Somogyi, Z. 1987. A system of precise modes for logic programs. *Logic Programming: Proceedings of the 4th International Conference*, pp. 769–787.

Somogyi, Z., Henderson, F. and Conway, T. 1996. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *J. Logic Programming 29*, 17–64.

Wadler, P. 1989. Theorems for free. *Conference on Functional Programming Languages and Computer Architecture*. ACM Press, pp. 347–359.