Adding constraint solving to Mercury

Ralph Becket[1], Maria Garcia de la Banda[2], Kim Marriott[2], Zoltan Somogyi[1,3], Peter J. Stuckey[1,3], and Mark Wallace[2]

[1] Department of Computer Science and Software Engineering, The University of Melbourne, Australia
[2] School of Computer Science and Software Engineering Monash University, Australia
[3] NICTA Victoria Laboratory, Australia

Abstract. The logic programming language Mercury is designed to support programming in the large. Programmer declarations in conjunction with powerful compile-time analysis and optimization allow Mercury programs to be very efficient. The original design of Mercury did not support constraint logic programming (CLP). This paper describes the extensions we added to Mercury to support CLP. Unlike similarly motivated extensions to Prolog systems, our objectives included preserving the purity of Mercury programs as much as possible, as well as avoiding any impact on the efficiency of non-CLP predicates and functions.

1 Introduction

Constraint logic programming (CLP) [9] is considered the archetypal form of constraint programming thanks to three properties inherited from logic programming: its declarativeness, which allows users to state problems simply and correctly, its relational nature, which suits the definition and usage of constraints, and its built-in backtracking, which simplifies the specification of search. However, declarativeness also complicates the implementation of efficient constraint solvers, since this often requires the use of programming techniques (such as destructive update or control of the goal execution order) that lack a straightforward declarative reading.

As a result, constraint solvers are often implemented in other (non-declarative) languages, thus achieving efficiency of constraint solving but incurring an interface overhead between the modelling language and the constraint solver. Not only must the solver interface allow constraints to be passed to the solver, but it must also support memory management and backtracking. Typically such functionality is not directly provided by external solvers, making the interface complex and unnecessarily inefficient. Furthermore, the common need to access internal solver information means that the representation of the constraint store and its efficient but non-declarative manipulation spreads throughout the CLP interface. Since virtually no CLP languages make a distinction between declarative and non-declarative code (effectively making all code non-declarative)

CLP is, in practice, less efficient and arguably no cleaner than constraint programming embedded within a procedural paradigm.

In contrast to CLP, Mercury [12] has explored a different direction for the logic programming paradigm: that of a purely declarative general purpose language designed to support well-engineered, efficient, large programs. This is achieved by including an effective module system, strong, expressive, and statically checked type, mode, and determinism systems, clear separation between declarative and non-declarative code, and extensive compiler optimizations.

The above characteristics make Mercury an excellent candidate for achieving our objective: to design and implement a CLP platform that retains the advantages of CLP without compromising on programming style, efficiency, or scalability. This objective is similar to that driving the design of the HAL language [3], which compiled to Mercury, adopted its module, type, mode, and determinism systems, and extended them to support solvers. We chose instead to extend Mercury to become itself a CLP language. Our reasons for doing this were basically software engineering reasons: the HAL compiler duplicates much of the work done by the Mercury compiler and generates Mercury code. It is significantly simpler and more efficient to have all this work done under one roof.

Our extension of Mercury builds on some of the authors' positive and negative experiences building and using HAL. Indeed, while the main change required to support constraint solving in Mercury (an extension to the mode system to allow constrained variables, as described in section 3.1) was done to support HAL's generation of Mercury code, the design of several key issues in this paper, such as the handling of solver types and solver interfaces, is improved significantly compared with HAL. We believe these changes extend Mercury's support for constraint programming a significant step further, making it a clean declarative language for implementing efficient constraint solvers.

Our main contribution in this paper is a new design for solver types that provides a clean separation between the viewpoint of the solver user (the *external* view) which sees them as traditional solver variables, and the viewpoint of the solver implementor (the *internal* view) which sees them as data structures used to access the information required by the external view. Our extension supports this duality through the use of two different types with different instantiation states, linked together by a new `solver type` declaration. Several "solver interface cast" functions are automatically generated from this solver type declaration to allow easy conversion from one type and instantiation state to the others.

The two different types and instantiation states make it easier for solver implementors to provide a purely declarative interface to users, while still using imperative techniques. Thus, only solver implementors need to peek below the "purity" hood. This solution is not only semantically cleaner than that used in HAL, but also more powerful.

The rest of the paper is organized as follows. Section 2 introduces the necessary background. Section 3 describes the changes required to allow Mercury programs to use solvers written in other languages, while Section 4 describes the features we added to allow solvers to be written in Mercury itself, possibly as hybrid solvers written on top of other solvers. The last section provides comparisons to related work.

2 Background

While the syntax of Mercury is based on the syntax of Prolog, semantically the two languages are very different due to Mercury's purity; its type, mode, determinism and module systems; and its support for evaluable functions. Mercury has a strong Hindley-Milner type system very similar to Haskell's. Mercury programs are statically typed; the compiler knows the type of every argument of every predicate (from declarations or inference) and every local variable (from inference).

The initial version of the mode system classified each predicate argument as either input or output. If input, the argument passed by the caller must be a ground term; if output, the argument passed by the caller must be a distinct free variable, which the predicate or function will instantiate to a ground term. The extensions we describe later in this paper introduce other instantiation states and modes. It is possible for a predicate or function to have more than one mode; the usual example is append, which has two principal modes: append(in,in,out) and append(out,out,in). We call each mode of a predicate or function a *procedure*. Each procedure has a determinism, which puts limits on the number of its possible solutions. Procedures with determinism *det* succeed exactly once; *semidet* procedures succeed at most once; *multi* procedures succeed at least once; while *nondet* procedures may succeed any number of times.

2.1 Foreign code and purity

Mercury supports access to code written in other languages by allowing the implementation of a Mercury predicate or function to be given in one of the languages supported by the Mercury compiler. On Unix systems, that means C; when generating code for .NET, it means C# or managed C++. For example, the syntax for defining a function `sin/1` in C is:

```
:- func sin(float) = float.
:- pragma foreign_proc("C", sin(X::in) = (Result::out), [promise_pure],
                        "Result = sin(X);").
```

The arguments of the foreign_proc pragma give the language of the code, the name of the predicate or function together with its arguments and their modes, a list of attributes, and the foreign code itself. The promise_pure attribute is part of the Mercury purity system, which classifies every predicate into one of the following three categories:

pure — the predicate is referentially transparent: the set of values it computes for its output arguments is completely determined by the values of the input arguments. Calls to such predicates may be optimized away or reordered freely.

impure — the predicate is not referentially transparent: the set of values it computes for its output arguments may depend on the current state of the computation in an arbitrary way. The execution of a call to an impure predicate may affect the behavior of future calls to impure and semipure predicates in an arbitrary manner. Calls to such predicates cannot be optimized away or reordered.

semipure — again, the predicate is not referentially transparent. However, the execution of a call to a semipure predicate cannot affect the behavior of future calls to impure and semipure predicates. Calls to such predicates can be optimized away or reordered *only* within the fenceposts formed by the surrounding impure calls.

Thus, predicates that write (and possibly read) state beyond their arguments are impure, while those that read but do not write are semipure:

```
:- semipure pred get_global(globaltype::out) is det.
:- pragma foreign_proc("C", get_global(X::out), [promise_semipure],
                        "X = some_global_variable;").
:- impure pred set_global(globaltype::in) is det.
:- pragma foreign_proc("C", set_global(X::in),    [],
                        "some_global_variable = X;").
```

Mercury allows programs to use impure code to implement a pure interface. Non-pure calls must be marked, and the user must promise that the predicate at the interface behaves as a pure predicate (i.e., its outputs depend only on its inputs). One example is the solutions predicate, which returns all solutions of a given goal as a sorted list. While this interface is purely declarative, the implementation of solutions uses a failure-driven loop patterned after Prolog's findall. After every success of the given goal,

it saves the solution in a global variable and backtracks; when the goal has no more solutions, it picks up the list of recorded solutions from the global variable, sorts them and returns the result. This code is clearly impure, but the effect of this impurity is not visible outside solutions, because no other part of the system looks at that global variable. While predicates implemented in foreign languages are impure by default, Mercury predicates are pure unless they call semipure or impure predicates.

2.2 Type-specific representation and equality

Since all types are known at compile time, term representation is specialized for each type. A given bit pattern may thus mean one term if it represents a value of type t1, and another for a value of type t2. The generic unify and compare predicates take an extra input describing the type of the arguments, which is used to invoke the specific unify or compare predicate for that type. These type-specific unify and compare predicates are usually created automatically by the compiler from the type definition, but programmers are allowed to define their own unify and/or compare predicates. For Mercury types defined in foreign languages, this is the only way unification and comparison *can* be defined. Additionally, this can also be useful for Mercury types on which semantic equality differs from structural equality, as it is the case when using unordered lists that may contain duplicates to represent sets. The type declaration:

```
:- type set_unordlist(T) ⟶ unord(list(T))
        where equality   is sort_elimdups_and_unify,
              comparison is sort_elimdups_and_compare.
```

specifies that for values of type set_unordlist(T), where T can be any type, unification should be defined by the sort_elimdups_and_unify predicate. User-defined equality imposes a proof obligation on the programmer, who must ensure that the equality predicate satisfies the usual properties of reflexivity, transitivity, commutativity and, most importantly, equivalence under replacement of equal objects. In other words, given A = B, it should be possible to substitute B for A in any call in the program without changing the call's results, with the exception of calls to semipure and impure predicates, which are known not to be referentially transparent [6].

3 Interfacing to external solvers

The easiest way to add constraint solving capability to a Mercury program is to provide an interface to an existing solver such as CPLEX [2] written in a foreign language. The natural way to do this is to create a module that

exports the operations of the external solver together with an abstract type representing the variables that participate in the constraints. This requires an extension of the mode system, an extension that is also needed for writing solvers entirely in Mercury, as we will discuss in section 4.

3.1 New versus old variables

The standard version of the Mercury mode system requires the compiler to know exactly which goal binds each variable. This is inherently impossible to achieve in constraint programs since, given a sequence of constraints, the particular constraint that fixes the value of a variable is frequently data dependent. We therefore added an instantiation state called old which indicates the variable is known to a solver and thus may be constrained, but it is not known whether it is ground. For example, a finite domain variable might be known to be greater than 3 and smaller than 7 (thus, it is not free), but its exact value might not be known yet (thus, it is not semantically ground either).

All variables start life in instantiation state new. For ordinary variables life is simple: at some point known to the compiler they become ground. For variables that occur in constraints, things are a bit more complex. When they first become known to their constraint solver (at a point known to the compiler) they change from new to old. Their instantiation state stays old as more constraints are added to them, unless they participate in an operation that is *known* to fix their value, in which case they become ground. However, most constrained variables die old.

The predicate or function that first makes a variable known to a solver takes that variable from instantiation state new to old (referred to as argument mode no, after their initials). In most cases, this is a specialized initialization predicate that does nothing else. Usually, solver operations work only on variables that have already been initialized, so they take solver variables from instantiation state old to old, which we abbreviate as argument mode oo. In some solvers, adding a constraint can be more efficient if some of its variables are known not to have previous constraints on them. In such cases, the solver may export operations that both initialize a variable and put a constraint on it. For example, a function like addition (+) on a constrained float type cfloat may be declared as

```
:- pred   cfloat + cfloat = cfloat.
:- mode oo + oo = no is det.
```

The set of things user programs can do with old variables is restricted to passing them around, putting them into data structures, and calling

the operations of their solver module. (This is enforced by the definition of their type being visible only in the solver module; they are not distinguished syntactically from other variables.) Since the concrete representation of old variables is usually just an index into the constraint store, unifying them with a term or with another variable using Mercury's usual structural equality is not meaningful. Solver programmers must avoid this unsoundness by defining type-specific equality predicates for the types of constrained variables.

3.2 *Tell* versus *ask* goals

Solver operations can usually be divided into two classes, *tells* and *asks* (some operations are both). While tell operations add new constraints to the store, ask operations inspect the store, usually to decide whether a constraint is entailed by it or not. Thus, tell operations are usually semidet, since they might find the resulting store to be inconsistent. It is also possible for a tell operation to be det if it works with fresh (new) variables, as in the addition example above.

Tell operations can be (and usually are) pure. Even though their implementation includes side-effects (updates to the global constraint store), these are not visible to the solver user as long as the solver ensures the operations are order-independent, i.e., the consistency or inconsistency of the store does not depend on the order in which constraints are added. Consider the Herbrand constraint solver built into every Prolog system. A unification is a tell operation which adds a new Herbrand constraint to the store, and it is implemented via side-effects such as making one variable point to another. Nevertheless, from the user's point of view, the solver maintains referential transparency and is thus pure.

Unfortunately, tell constraints can cause problems when appearing in a negated context, i.e., in the body of a negation or in the condition of an if-then-else (if C then T else E is semantically equivalent to $(C \wedge T) \vee (\neg\exists C \wedge E)$). This is because tell constraints often have arguments of mode oo, and Mercury cannot decide whether these arguments become further constrained by the tell. If they are, this is unsound since the negated goal binds variables visible from the outside. This can, for example, destroy the commutativity of conjunction: if X and Y are initially unconstrained, then executing $X \geq Y$, not$(X < Y)$ should succeed, whereas not$(X < Y)$, $X \geq Y$ would fail ($X < Y$ would succeed after constraining X and Y, so its negation would fail). Tell constraints occuring in negated contexts should be translated to ask constraints (see e.g. [4]). Our current solution is to implicitly make impure any goal occurring in a negated context that contains a nonlocal variable with inst old. In these cases, it is up to the

programmer to decide whether such goals really are pure and add a purity promise if they are.

The result of executing an ask constraint depends upon the state of the constraint store, and thus upon when it is executed. Whereas tell goals can be reordered arbitrarily with respect to each other, ask goals should not be reordered with respect to tell goals. Furthermore, since some ask goals also change the constraint store, they should not be reordered with respect to other ask goals either. The simplest way to achieve this is to make them impure. This is also semantically desirable, since the undisciplined use of ask goals can break referential transparency. Consider the ask constraint fixed(X) which succeeds if solver variable X is fixed to a unique value. The goals fixed(X), X = 3 and X = 3, fixed(X) would have different behaviour.

4 Writing constraint solvers in Mercury

We want to allow programmers to write solvers directly in Mercury, either from scratch, or using other solvers. This requires significant additional changes to the language.

4.1 Solver types

Solver users see constrained variables as black boxes whose implementation is hidden, and which spend most of their life in instantiation state old. Solver writers, on the other hand, must know the constraint variable's structure and must be able to manipulate it. For efficiency, this usually requires constraint variables to have a more concrete instantiation state such as ground. While the notion of abstract data types can be used to provide the user's view, we need a new mechanism, which we call *solver types*, to support the solver writer's view.

Solver types are for variables whose values may have constraints placed on them. Solver types are exported abstractly, i.e., their definition stays hidden and the only operations users can invoke on their values are those exported by the solver module. The following provides the module interface of the solver we will use as our running example:

```
:- solver type po_vertex.
:- pred init(po_vertex::no) is det.
:- pred eq(po_vertex::oo, po_vertex::oo) is semidet.
:- pred '<'(po_vertex::oo, po_vertex::oo) is semidet.
:- pred '≤'(po_vertex::oo, po_vertex::oo) is semidet.
:- impure pred order(list(po_vertex)::in, list(po_vertex)::out) is semidet.
```

The first line declares po_vertex to be an abstract solver type, while the other lines declare the operations available on it. The init predicate

creates a fresh variable of the solver type; eq, $<$, and $\leq$ each tell the solver to impose the constraint they stand for; and order asks for a total order consistent with the partial order required by the constraints imposed so far, using the supplied order as a preference to break any ties.

The implementation section of the solver module defines some auxiliary types (vertex is a type synonym for integers, and constraint is a type with data constructors, lt for less-than constraints and le for less-than-or-equal-to constraints), and then gives the actual definition of the solver type as follows:

```
:- type vertex     == int.
:- type constraint ⟶ lt(vertex, vertex) ; le(vertex, vertex).

:- solver type po_vertex where
    representation   is vertex,
    equality         is eq,
    initialisation   is init,
    constraint_store is
        [ mutable(counter, int, 0),
          mutable(constraints, set(constraint), empty_set) ].
```

The following subsections explain the various parts of this declaration.

4.2 The external and internal views of solver types

A solver type presents the "external" view of a constrained variable, which is the only view available to its users. Every solver type also has an underlying *representation* type, which presents the "internal" view visible only to the solver implementation. This representation type is specified by the representation is vertex part of the declaration.

In our example, a constrained variable of solver type po_vertex is represented by a variable of type vertex, which is just a synonym for integer. These two types are semantically quite different. Equating two values of type vertex simply requires testing whether two integers are the same, while equating two values of type po_vertex requires adding a new eq constraint to the store and testing its consistency. For instance, let V_1 and V_2 be two currently unconstrained variables of type po_vertex with internal representations 42 and 69, respectively. While $V_1 = V_2$ should succeed, constraining the two solver variables to behave identically with respect to all other solver operations, $42 = 69$ should fail.

We can distinguish between these two kinds of equality thanks to Mercury's support for user-defined equality. The "equality is eq" part of the declaration indicates to Mercury that equality for values of solver

type po_vertex is defined by the eq predicate, rather than by the default structural equality relation. (The Mercury compiler and runtime system together implement the unification $V_1 = V_2$ in the previous paragraph by calling eq(V_1, V_2).) Values of the internal type vertex, on the other hand, will use the standard equality definition for integers. This separation into two types allows solver writers to ensure the referential transparency of exported predicates and functions, something that must be done by any declarative language with true programmer defined equality. [1]

Separating the external, solver type from the internal, representation type also allows their treatment to differ in other respects, such as making just one of them an instance of a type class, or providing different implementations for the methods of a type class. For example, consider an overloaded predicate show for pretty-printing. Applying show to a vertex should simply print an integer (since a vertex *is* an integer), but showing a po_vertex could, for instance, list the constraints on the vertex.

4.3 Converting between internal and external views

We need to provide ways of moving from the external type with its external instantiation state (old) to the internal type with its internal instantiation state (usually ground, but see section 4.6), and vice versa. The solver type declaration allows the Mercury compiler to automatically create the two casting functions required by the solver writer to do this. For our running example, these functions are:

```
:- impure func from_old_po_vertex(po_vertex::oo) = (vertex::out).
:- impure func to_old_po_vertex(vertex::in)      = (po_vertex::no).
```

where from_old_po_vertex takes an old po_vertex value and returns its internal ground vertex representation, while the dual function to_old_po_vertex takes a ground vertex value and returns the corresponding old po_vertex. Note that by default, internal representations are ground values. This can be overridden in the solver type declaration if, for example, the internal representation is defined in terms of another solver type.

The casting functions are impure because a semantically non-ground value of the external type may be (and typically is) represented by a ground value of the internal type. No declarative reading can be given to such a relationship. While the value of the external type may be further constrained, this does not affect the already ground value of the internal

[1] Languages like Haskell sidestep the same problem by treating the (possibly user-defined) equality relation == as having no relation to the equality = used for referential transparency. This is not possible in a relational language due to the pervasive, implicit use of equality.

type. In the internal view, this is usually reflected only in the constraint store, which is not an argument to the casting functions.

Operationally, cast functions are just the identity function. Calls to these functions are guaranteed to be optimized away, and thus have no performance cost. They exist only to bridge the "semantic gap" between a solver type and its internal representation.

4.4 The constraint store

The value of a solver variable cannot be understood in isolation from the constraint store of its solver. The

```
constraint_store is
    [ mutable(counter, int, 0),
      mutable(constraints, set(constraint), empty_set) ].
```

part of the declaration indicates that the constraint store for the po_vertex type is stored in two mutable global variables, one containing the id of the next vertex to be allocated (each new vertex is given a different integer identifier), the other containing the set of constraints in the store. In this case, the store may contain only lt and le constraints (this example represents equality constraints as a pair of le constraints). Initially no vertexes have been created and the set of constraints is empty. More complex solvers would have more sophisticated data structures.

The Mercury compiler automatically creates two access predicates for each mutable variable, which in this case will have the signatures

```
:- semipure pred get_counter(int::out) is det.
:- impure   pred set_counter(int::in) is det.
:- semipure pred get_constraints(set(constraint)::out) is det.
:- impure   pred set_constraints(set(constraint)::in) is det.
```

Exported solver predicates start by reading (parts of) the store from these global variables and finish by updating them, if necessary. Updates to the global variables are trailed to ensure they are automatically undone on backtracking.

4.5 Solver operations

The functionality of the solver derives from the functions and predicates that it exports. In practice, this is where the bulk of the solver code lies. This code uses predicates and functions created by the compiler from solver type declarations to map the external types and instantiations to internal ones and vice versa, and to lookup and modify information in the global solver state. For example, the less-than-or-equal-to constraint listed in the solver type interface might be defined as in figure 1.

```
A ≤ B :-
    promise_pure(
        impure X = from_old_po_vertex(A),
        impure Y = from_old_po_vertex(B),
        semipure get_constraints(Arcs0),
        ( if    path(X, Y, Arcs0, _)
          then true
          else  not path(Y, X, Arcs0, strict),
                Arcs = set.insert(Arcs0, le(X, Y)),
                impure set_constraints(Arcs)
        )
    ).
```

Fig. 1. The code of the predicate that adds a less-than constraint

If there is an existing path from X to Y in the constraint graph, then the constraint $X \leq Y$ is already entailed and we return. (The path predicate looks for acyclic paths in $Arcs_0$, and tracks whether the path traverses a strict constraint or not.) Otherwise, if there is an existing *strict* path from Y to X (implying $Y < X$), then we fail, since the constraint $X \leq Y$ is inconsistent with the current constraint graph. Otherwise we add le(X, Y) to the constraint graph and update the global constraint store.

4.6 Hybrid solvers

It is also possible to define a new solver type in terms of other solver types. Figure 2 shows how a lexicographically ordered solver type could be defined in terms of the po_vertex solver (this example is purely illustrative: in practice a user of the po_vertex type would just use a pair of po_vertexes directly rather than hiding the representation behind another solver type). The solver type declaration contains an extra attribute, any is bound(lex_rep(old, old)), which specifies that the instantiation state of the representation type that corresponds to the inst old of the external type is not ground, but rather the function symbol lex_rep wrapped around two old values.

5 Related work

Mercury's focus on purity while not neglecting efficiency (quite the contrary!) leaves very few directly competing logic programming languages. The closest relative is HAL [3], which itself used Mercury as an implementation language. The vast bulk of other logic programming systems supporting the implementation of constraint solvers can reasonably be described as Prolog extensions (e.g. [1, 5, 7, 11]). Oz [10] supports constraint programming, but any new constraint solvers have be written in a foreign language (C++).

```
:- solver type lex where
    representation is lex_rep,
    any            is bound(lex_rep(old, old)),
    equality       is eq_lex,
    initialisation is init_lex.

:- type lex_rep ⟶ lex_rep(po_vertex, po_vertex).

:- pred init_lex(lex::no) is det.
init_lex(A) :-
    promise_pure(
        init(A1), init(A2), impure A = to_old_lex(lex_rep(A1, A2))
    ).

:- pred eq_lex(lex::oo, lex::oo) is semidet.
eq_lex(A, B) :-
    promise_pure(
        impure lex_rep(A1, A2) = from_old_lex(A),
        impure lex_rep(B1, B2) = from_old_lex(B),
        A1 = B1, A2 = B2
    ).

:- pred '≤'(lex::oo, lex::oo) is nondet.
A ≤ B :-
    promise_pure(
        impure lex_rep(A1, A2) = from_old_lex(A),
        impure lex_rep(B1, B2) = from_old_lex(B),
        ( A1 < B1 ; A1 = B1, A2 ≤ B2 )
    ).
```

Fig. 2. Defining a solver type in terms of another solver type

5.1 Prolog-based systems

Key characteristics of Prolog-based systems are a dynamic type system, no mode checking, support for aliasing of all variables, and dependence on impure language features while lacking any mechanism for distinguishing pure code from impure code. The lack of a static type system means that program variables all have the same "universal type". Even in systems with optional type declarations, such as CIAO [7], the compiler cannot optimize the representation of a term to its type without breaking the assumptions of e.g. the debugger and the garbage collector. The compiler therefore cannot optimize the representation of solver types either. Because the absence of a mode system allows variables to become aliased before they become ground, every variable must be initialised before use. It also means that every time the system wants to look up the value of a variable, it needs to be prepared to follow a chain of aliasing pointers first.

These characteristics make it very difficult to build fast Prolog systems (e.g. implementors must write program analyses if they wants to optimize away dereferencing). Prolog systems that have been extended to support constraint programming typically use *attributed variables* [8] to associate solvers with variables. This complicates the representation of variables even further, and makes unification more complex and expensive due to the need to check at many steps whether any attributed solver goals have to be invoked. However, the biggest drawback of building constraints on top of attributed variables is that code built that way has no clear, well-defined boundary between the pure external view and the internal impure view, which makes programs harder to maintain and to optimize.

The Mercury compiler, by contrast, knows the type of every variable, and each type has a separately optimized low-level representation. Thanks to the mode system, the Mercury compiler also knows at each point whether a given variable is new, old, or ground. Consequently, new variables may contain just junk data, and do not require any kind of initialization. Only *solver type* variables need to be initialized, at the point where their instantiation state changes from new to old (or to ground). Because new Mercury variables cannot be aliased (Mercury uses code reordering to ensure that at least one side of a unification is old or ground before the unification is carried out), variables do not need to be dereferenced. A particular solver *might* use aliasing as part of its implementation of equality, but that is an implementation decision made by the solver programmer and can have no effect on the performance of variables with other types, which have separate representations. For example, a Herbrand solver type for terms with Prolog-style unification could use a WAM-style [13] representation, where variables may be aliased and would therefore need dereferencing. On the other hand, a solver type interfacing to a SAT solver might unify variables by simply adding clauses equating the two variables to its constraint store. Similarly, because solver variables are handled exclusively by solver implementations, solvers can immediately inspect their variables' values. There is no need for a general attribute variable mechanism, and thus no overhead is incurred by non-solver types.

Mercury also gets additional speed from the Mercury compiler's ability to optimize away some computations and to reorder some others (e.g. to make failure happen earlier). The Mercury compiler is allowed to exercise this ability only on pure code; removing or reordering impure code could change the program's output. Mercury programmers often write clear, maintainable code, even if it is inefficient, if they know the compiler

can eliminate the inefficiency. Our solver type design allows and indeed encourages solver writers to keep both solver interfaces and the codes of the solvers themselves as pure as possible, and requires them to cleanly separate out the impure code. This preserves maximum freedom for the compiler and allows programmers to maintain a declarative programming style. This has genuine advantages for both compiler implementors (optimizations are easier to implement) and constraint programmers (declarative code is more maintainable). These are real advantages not available with Prolog-based approaches.

5.2 HAL

Like Mercury, HAL also has external and internal views of solver types. But rather than making these genuinely distinct types, the external view is a specially handled renaming of the solver type representation. Outside the solver module, a HAL solver type is an abstract type whose values typically have instantiation state old and whose equality is defined by a programmer specified predicate in the solver module. Inside the solver module, the solver type is a concrete type, values of that type have a different instantiation state (usually ground), and the applicable equality semantics is structural equality rather than the equality predicate used for the external view.

The first problem with this approach is that referential transparency is much more complicated for solver types, since what equality means for such terms is different depending upon whether a unification occurs inside or outside the solver module. This means for example that code performing such unifications must not be subject to intermodule optimizations such as inlining. The second is that the solver type module cannot define predicates in terms of the external view. For example, in the Mercury po_vertex solver module the programmer can define equality of po_vertexes in terms of $\leq$ for po_vertexes:

$$\mathsf{eq}(\mathsf{A},\ \mathsf{B})\ \mathsf{:-}\ \mathsf{A} \leq \mathsf{B},\ \mathsf{B} \leq \mathsf{A}.$$

In a HAL solver module, however, A and B would be viewed as integers (the representation view), hence the integer version of $\leq$ would be used instead of the po_vertex version, which has quite different properties! To see the third problem, consider a showable class the debugger may use to print values. We would like different things printed in the internal and the external views, but with the HAL approach, this is not possible, since there is only one type.

The Mercury approach avoids these problems by making the external and internal views distinct types and requiring the programmer to ex-

Problem	Size	Reps	Mercury	HAL	Eclipse
serialize	10,000	2,000	63.5	98.9 (1.56)	164.9 (2.60)
	7,500	3,000	68.4	96.5 (1.41)	180.1 (2.63)
	5,000	4,000	58.8	69.0 (1.17)	152.6 (2.60)
warplan		100	8.8	8.2 (0.93)	12.0 (1.36)
hanoi	10	20,000	15.2	17.3 (1.14)	31.7 (2.09)
	13	2,000	12.5	18.6 (1.48)	25.5 (2.04)
	16	200	10.8	18.5 (1.72)	31.9 (2.95)
qsort	10,000	200	45.9	55.4 (1.21)	221.5 (4.83)
	7,500	400	52.5	63.2 (1.21)	241.2 (4.59)
	5,000	800	48.7	58.4 (1.20)	204.9 (4.21)
laplace	10	6,000	28.1	32.2 (1.15)	72.7 (2.59)
	20	400	20.0	32.7 (1.64)	47.8 (2.39)
	30	50	15.9	33.3 (2.01)	37.7 (2.37)
matmul	10	2,000	12.2	37.4 (3.07)	60.9 (4.99)
	20	200	14.8	33.5 (2.26)	52.1 (3.52)
	30	40	14.3	38.3 (2.68)	42.6 (2.98)
mortgage		1,000	26.9	19.4 (0.72)	1140.0 (42.0)
fib	12	1,000	104.9	100.1 (0.95)	1667.3 (15.9)

Table 1. Benchmark results.

plicitly cast between them. We feel the modest amount of extra typing required is more than compensated for by the increased flexibility, clarity, and protection from errors.

6 Experimental evaluation

The results presented here are solely presented to illustrate that the Mercury approach can be used to implement competitive solvers. The benchmark programs for each language are as similar as possible, although the solver type implementations are obviously different in each case. It is important to bear in mind that the performance of a given solver is determined much more by how it is implemented than in what language: the better algorithm will usually win! Using the exact same algorithm on all systems usually isn't feasible, and even when it is, a given algorithm may be a better fit for one system than for another. This is why despite our best efforts, the benchmarks here are far from being apples-to-apples comparisons. That said, it seems clear that Mercury is generally faster than HAL and Eclipse.

These benchmarks were run on a PC with dual 933MHz Pentium III CPUs and 2 GBytes of RAM running Linux kernel version 2.4.3. All times

are given in seconds and performance relative to Mercury is also given in parentheses for all other benchmark times. The compiler versions used were Mercury rotd-2005-08-21, the last development release of the HAL compiler (work on HAL ceased in 2004), and Eclipse 5.8. The CPLEX benchmarks were linked against CPLEX 7.0.

Table 1 gives the results. serialize, warplan, hanoi, and qsort are standard Prolog benchmarks to test performance on Herbrand constraints, i.e. Prolog-style unification. HAL's superior performance on warplan, the most challenging of this group of benchmarks, reflects the considerable effort expended by the HAL team on efficient Herbrand types.

laplace, matmul, fib, and mortgage test solver interfaces to CPLEX [2], an off-the-shelf linear constraint optimizer. laplace computes a matrix using Laplace's equation. matmul inverts a matrix of prime numbers by multiplying it with a matrix of variables and equating the result with the unit matrix. mortgage computes mortgage costs on a $120,000 dollar loan over 120 years at 1% interest and then runs the same computation backwards. fib takes a naive approach to computing Fibonacci numbers in the forward direction. In laplace and matmul, constraints are "batched" together and solved once at the end of the query. In mortgage and fib, constraints are incrementally checked for consistency because they control recursive loops. We believe HAL's superior performance on these two benchmarks is due simply to the fact we haven't had as much time to optimize this part of Mercury's interface to CPLEX.

7 Conclusions

We have extended Mercury with the instantiation old and solver types. When we began this work, we thought supporting solver types would be a straightforward and relatively uninteresting design problem. In the end it required several attempts and a great deal of careful thought to arrive at a clean design that could be implemented efficiently without sacrificing referential transparency.

So far, we have used these new Mercury features to implement a Herbrand solver, a propagation based finite domain solver, and a BDD-based set solver, as well as interfaces to CPLEX and SATZ (a SAT solver). We have found the solver type mechanisms to be easy to use, and in each case, the interface of the solver seen by its users is totally pure, with the exception of ask predicates.

We have evaluated the performance of some of the above solvers against comparable solvers in other languages. The results are very en-

couraging: in the benchmarks we have run, Mercury is the fastest system in almost all cases.

The system we have described is now available in releases-of-the-day from the Mercury web site. The full source code of our running example is also available from there, next to this paper on the papers page.

We would like to thank Fergus Henderson for many useful discussions, and NICTA and the Australian Research Council for their support.

References

1. B-Prolog. `www.sci.brooklyn.cuny.edu/~zhou/bprolog.html`.
2. ILOG CPLEX. www.cplex.com.
3. B. Demoen, M. Garcia de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. An overview of HAL. In *Proceedings of the Fourth International Conference on Principles and Practices of Constraint Programming*, pages 174–188, 1999.
4. G.J. Duck, M. García de la Banda, and P.J. Stuckey. Compiling ask constraints. In B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, LNCS, pages 105–119. Springer-Verlag, 2004.
5. ECLiPSe. http://www.icparc.ic.ac.uk/eclipse/.
6. F. Henderson, T. Conway, Z. Somogyi, D. Jeffery, P. Schachte, S. Taylor, and C. Speirs. The Mercury language reference manual. Available from http://www.cs.mu.oz.au/mercury/, 2000.
7. M. Hermenegildo, F. Bueno, D. Cabeza, M. Garcia de la Banda, P. López, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*. Nova Science, Commack, NY, USA, April 1999.
8. C. Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. In *Proceedings of PLILP '92*, number 631 in LNCS, pages 260–268. Springer-Verlag, 1992.
9. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of POPL '87*, pages 111–119, 1987.
10. Mozart. `http://www.mozart-oz.org`.
11. SICStus Prolog. http://www.sics.se/sicstus/.
12. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–64, 1996.
13. D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.