

# Translating Refined Logic Programs to Mercury

R. Colvin\*<sup>†</sup>

I. Hayes<sup>†</sup>

D. Hemer\*<sup>†</sup>

P. Strooper<sup>†</sup>

\*Software Verification Research Centre, University of Queensland

<sup>†</sup>School of Information Technology and Electrical Engineering, University of Queensland  
Email: {robert, ianh, hemer, pstroop}@csee.uq.edu.au

## Abstract

A refinement calculus provides a method for transforming specifications to executable code, maintaining the correctness of the code with respect to its specification. In this paper we investigate the use of Mercury as the target implementation language for a refinement calculus for logic programs. We describe a prototype tool for translating programs in our specification language to Mercury code. More generally, we investigate the advantages that Mercury has over standard Prolog, with respect to developing correct programs from specifications.

**Keywords:** Logic programming, refinement, Mercury, code generation

## 1 Introduction

The logic programming refinement calculus [Hayes et al., 2000] provides a method for systematically deriving logic programs from formal specifications. It is based on: a *wide-spectrum language* that can express both specifications and executable programs; a *refinement relation* that models the notion of correct implementation; and a collection of *refinement laws* providing the means to refine specifications to code in a stepwise fashion.

The wide-spectrum language includes assumptions and specification constructs that may not be executable, as well as a subset that corresponds to Horn clauses. The refinement relation is defined so that an implementation must produce the same set of solutions as the specification it refines, but it need do so only when the assumptions hold. There are refinement laws for manipulating assumptions and specifications, and for introducing code constructs.

In the logic programming paradigm, a first-order logical formula is said to be executable if it is in Horn-clause form, the atoms of which are language primitives or defined predicates. However the situation is more complicated than that, since in logic programming languages such as Prolog, the ordering of clauses and goals (disjuncts and conjuncts) can affect the correctness of a program. Thus determining exactly what is “code” requires knowledge of the underlying execution mechanism of the implementation language. This is in contrast to the development of code in the imperative refinement calculus [Morgan, 1994], where one just needs to refine to implementation language constructs.

In the refinement calculus for logic programs, the task of generating code from our language, which we refer to as translation, has been separated from

$\langle P \rangle$	-	specification
$\{A\}$	-	assumption
$(S \vee T)$	-	disjunction
$(S \wedge T)$	-	parallel conjunction
$(S, T)$	-	sequential conjunction
$(\exists X \bullet S)$	-	existential quantification
$(\forall X \bullet S)$	-	universal quantification
$pc(K)$	-	procedure call
$re\ p \bullet V \text{ :- } \mathcal{C}(p)\ er$	-	recursion block

Figure 1: Summary of wide-spectrum language

the task of refining to a pure logic program. Translation is easier if the semantics of the target language are not affected by the ordering of the program. The compiler of the logic programming language Mercury [Henderson et al., 2000] can perform a significant amount of goal ordering transparently to the developer. Mercury is also a purely declarative language, unlike many Prolog implementations which include language constructs that can break the declarative interpretation of a program.

In this paper we discuss Mercury as a target implementation language for a refinement calculus for logic programs. We present a translator, written in Mercury, that translates a (subset of all possible) wide-spectrum programs into compilable and runnable Mercury code. The translator is part of the logic program refinement tool Marvin [Hemer et al., 2001]. In Sect. 2 we present a summary of the refinement calculus for logic programs and Marvin. In Sect. 3 we discuss some of the features of Mercury that are relevant to using Mercury as a formal development language. We discuss the extensions to Marvin for translating wide-spectrum programs into Mercury code in Sect. 4.

## 2 Refinement calculus for logic programs

In this section we briefly describe the refinement calculus for logic programs. The specification language we use is described in Sect. 2.1, and the semantics of the language and refinement is outlined in Sect. 2.2. We demonstrate the calculus in Sect. 2.3, by presenting a specification of a program that checks for membership in an ordered binary tree, and the corresponding recursive implementation of the specification that may be obtained via refinement. We describe tool support for the calculus in Sect. 2.4.

### 2.1 Wide-spectrum language

In our wide-spectrum language we can write both specifications as well as executable programs. This has the benefit of allowing stepwise refinement within a single notational framework. A summary of the language is shown in Fig. 1.

**Specifications and assumptions.** A specification  $\langle P \rangle$ , where  $P$  is a predicate, represents a set of instantiations of the free variables of the program that satisfy  $P$ . For example, the specification  $\langle X = 5 \vee X = 6 \rangle$  represents the set of instantiations  $\{5, 6\}$  for  $X$ . An assumption  $\{A\}$ , where  $A$  is a predicate, allows us to state formally what a program fragment assumes about the context in which it is used. For example, some programs may require that an integer parameter be non-zero, expressed as  $\{X \neq 0\}$ . We assume a rich set of mathematical operators are available in our predicate language.

**Program Operators.** The disjunction of two programs  $(S \vee T)$  computes the union of the results of the two programs. There are two forms of conjunction: a parallel version  $(S \wedge T)$ , where  $S$  and  $T$  are evaluated independently and the intersection of their respective results is formed on completion; and a sequential form  $(S, T)$ , where  $S$  is evaluated before  $T$ . In the sequential case,  $T$  may assume the context established by  $S$ .

**Quantifiers.** The existential quantifier  $(\exists X \bullet S)$  generalises disjunction, computing the union of the results of  $S$  for all possible values of  $X$ . Similarly, the universal quantifier  $(\forall X \bullet S)$  computes the intersection of the results of  $S$  for all possible values of  $X$ .

**Procedures.** A procedure definition has the form  $pc \hat{=} F :- S$ , where  $S$  is a wide-spectrum program. It defines the procedure called  $pc$  with a list of formal parameters  $F$  and body  $S$ . A call on the procedure  $pc$  is of the form  $pc(K)$ , where  $K$  is a list of actual parameters.

**Recursion.** A recursion block has the form  $\mathbf{rep} \bullet V :- \mathcal{C}(p) \mathbf{er}$ . The body of the block,  $\mathcal{C}$ , encodes zero or more recursive calls to  $p$ , the actual parameters of which must all be less than  $V$  according to some well-founded relation.

**Types and modes.** We specify types using first-order predicates, e.g., we define the natural number type via the predicate  $isNat$ , using the usual successor notation.

$$isNat(N) == \\ N = 0 \vee \\ (\exists M \bullet isNat(M) \wedge N = suc(M))$$

This defines a natural number to be either 0 or the successor of another natural number.

The type of a variable can be declared either within an assumption, e.g.,  $\{isNat(L)\}$ , or within a specification, e.g.,  $\langle isNat(L) \rangle$ . A type assumption acts as a precondition to a program, and is similar to the condition that a variable be an input (in logic programming terminology, it has the mode “ground”). A type specification establishes the type within a program, and indicates that the parameter is to be an output. This information may help in the refinement process, as well as constrain the way a calling program can use the procedure (the calling program must ensure that the assumptions are satisfied). A more detailed examination of types in the refinement calculus for logic programs is presented in [Colvin et al., 2000].

$$\{isIntTree(T) \wedge isordered(T)\}, \\ \langle isInt(Y) \rangle \wedge \langle Y \in members(T) \rangle$$

Figure 2: Definition of ordered tree membership.

## 2.2 Semantics

In this section we present an informal exposition of the wide-spectrum language semantics and refinement. More detailed treatment of the semantics is presented in [Hayes et al., 1997, Hayes et al., 2000].

Each program (fragment)  $S$  in our language is defined by two components: first, an assumption component,  $ok.S$ , which is a predicate describing the conditions under which  $S$  is guaranteed not to abort, and second, the effect the program has on its free variables,  $ef.S$ . An assumption  $\{A\}$  in our language may abort if  $A$  does not hold, but has no effect on its free variables. A specification  $\langle P \rangle$  in our language will not abort<sup>1</sup>, but constrains its free variables to satisfy  $P$ ; if it cannot, the program fails (i.e., the set of answers is empty). A program in our language is then made up of these two basic building blocks, composed via the operators and quantifiers.

The refinement relation,  $\sqsubseteq$ , is defined with respect to the predicates  $ok$  and  $ef$ .

$$S \sqsubseteq T == ok.S \Rightarrow \left( \begin{array}{c} ok.T \\ ef.S \Leftrightarrow ef.T \end{array} \right)$$

We say that  $S$  is refined by  $T$  (written  $S \sqsubseteq T$ ), if  $T$  terminates whenever  $S$  terminates, and when  $S$  terminates the effect that  $T$  has on its free variables must be *equivalent* to the effect  $S$  has on its free variables. This is in contrast to the imperative refinement calculus [Morgan, 1994], where a program’s post-condition, corresponding loosely to our  $ef$  predicate, may be strengthened to reduce nondeterminism. In logic programming, a program must deliver all possible answers.

Refinement laws have been proven in terms of the semantic framework, including: algebraic properties of the language; monotonicity rules; and rules for introducing, eliminating and manipulating specifications and assumptions. As an example, consider the following law which may be used to weaken an assumption.

$$\frac{A \Rightarrow B}{\{A\} \sqsubseteq \{B\}}$$

We may refine an assumption  $\{A\}$  to an assumption  $\{B\}$ , if  $A \Rightarrow B$  holds. By applying such laws, we successively introduce implementation constructs into the program, until the program is in executable form.

## 2.3 Example

Consider the specification given in Fig. 2. It specifies the relation that  $Y$  is an element of the ordered binary tree  $T$ . The  $isInt(Y)$  and  $isIntTree(T)$  predicates define the type of the free variables  $Y$  and  $T$  to be an integer and a tree of integers respectively. The predicate  $isordered(T)$  states that  $T$  is ordered, and  $members(T)$  returns the set of elements contained in the tree  $T$ . Note that we *assume* that  $T$  is a tree, while we establish that  $Y$  is an integer via a specification. Thus we expect the implementation to treat  $T$  as an input, and  $Y$  as an output.

<sup>1</sup>A specification  $\langle P \rangle$  may abort if  $P$  is not well-defined; see [Hayes et al., 2000] for more details. In this paper we only use well-defined predicates.

```

re treeMember • (Y, T):-
  {isIntTree(T)}, ((isInt(Y)) ∧
  (∃ L, X, R • (T = tree(L, X, R)),
  ((X < Y) ∧ treeMember(Y, L)) ∨
  (Y = X) ∨
  ((X > Y) ∧ treeMember(Y, R))))
er

```

Figure 3: Recursive procedure *treeMember*

We may define *isIntTree* as follows.

```

isIntTree(T) ==
  T = empty ∨
  (∃ L, X, R • isInt(X) ∧
  isIntTree(L) ∧ isIntTree(R) ∧
  T = tree(L, X, R))

```

Given this definition, and assuming appropriate definitions for the predicate *isordered* and the function *members*, the specification in Fig. 2 may be refined to the program in Fig. 3. (Some changes in syntax aside, the details of the refinement may be found in [Hayes et al., 1997].) This is a recursive program that traverses the left and right branches of *T*, searching for *Y*. The assumption  $\{isIntTree(T)\}$  and the specification  $\{isInt(Y)\}$  have been left in the program, even though they may be eliminated. This is because the types of the formal parameters are required when we translate to Mercury code. We discuss the translation of this program to Mercury code in Sect. 4.

## 2.4 Tool support

Refinement of logic programs is supported by the Marvin tool [Hemer et al., 2001]. Marvin includes an embedding of the wide-spectrum language and its semantics in Isabelle/HOL [Paulson, 1994], an interactive theorem prover based on higher-order logic. Marvin includes a collection of refinement laws, proven with respect to the embedded semantics within the formal framework of Isabelle/HOL.

Starting with a top-level specification of their desired program, the user can progressively refine to an executable program by applying a series of refinement laws. Application of refinement laws often results in proof obligations that need to be discharged. Within Marvin these proof obligations can usually be discharged automatically by the underlying theorem prover engine. To assist the user in performing refinement, tactics have been implemented that allow the user to: refine a program by focusing on a subcommand; close on a subcommand; and apply refinement laws.

Having refined the top-level specification to an “executable” wide-spectrum program, the Marvin tool can generate implementable Mercury code. This part of the functionality of Marvin is represented by the translator described in this paper. The translator works in two stages: the first stage, written in Tcl, converts the wide-spectrum program from Isabelle syntax into a form readable by Mercury; the second stage, written in Mercury, takes the converted program and generates the final code. The first stage is a change in concrete syntax only; the second stage is described in Sect. 4.

## 3 Mercury

Mercury [Henderson et al., 2000] is a “logic/functional programming language, which combines the clarity and expressiveness of declarative programming

with advanced static analysis and error detection features”. Syntactically it is similar to Prolog, with a few additions. The most noticeable difference from Prolog is that all procedures must have associated type and mode declarations for their formal parameters. Using this information, the compiler can infer the types of local variables, and ensure that the program is well-typed and well-moded. An example Mercury program is given in Fig. 4. It is an implementation of the standard *append* relation between three lists, where *append*(*X*, *Y*, *Z*) holds iff *Z* is the catenation of *X* and *Y*.

```

:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out).
:- mode append(out, out, in).

append([], Y, Y).
append([H|T], Y, [H|V]) :- append(T, Y, V).

```

Figure 4: Implementation of *append* in Mercury

### 3.1 Types

The Mercury library includes support for types such as integers, strings and lists. Mercury also allows user-defined types. For example, the declaration for the natural numbers type (*nat*) using the successor notation would be

```

:- type nat ---> zero; suc(nat).

```

The type *nat* has a constant *zero* and a constructor *suc*/1, whose parameter must also be of type *nat*.

The first line of Fig. 4,

```

:- pred append(list(T), list(T), list(T)).

```

states that the three parameters to *append* are lists of the same (unspecified) type. The parameter *T* to *list* is a type variable, which may be instantiated to integers, strings etc. The code for *append* is the usual Prolog implementation. However, in Prolog, a call *append*([], 0, 0) would succeed, since the first clause does not require that *Y* be a list. Such problems are avoided in Mercury by explicit type declarations.

### 3.2 Modes

Mercury also requires mode declarations be provided for procedures. Mode declarations determine which parameters are to be inputs and which are outputs, defined by *in* and *out* respectively. For example, the second line of Fig. 4,

```

:- mode append(in, in, out).

```

is a mode declaration specifying that when the first two parameters are inputs, the last parameter is an output. This is the usual functional use of *append*, where two lists are catenated to form a third. The third line of Fig. 4,

```

:- mode append(out, out, in).

```

states that if the third parameter is input, the first two parameters are output. This mode declaration corresponds to decomposing a list into two sublists. Other mode declarations are possible for *append*, though for brevity we have omitted them from the definition.

The Mercury compiler can develop the code for multiple mode declarations from the one procedure definition, by reordering the goals to satisfy the mode

declaration (internally, a separate procedure is generated for each different mode). To achieve the same result in Prolog, a separate procedure may have to be written for each required mode (though some Prolog procedures work with multiple modes).

The Mercury compiler also generates “implied modes” for procedures. If a parameter is declared as an output in a mode, an implied mode will be generated which is the same as the original except with that parameter declared as an input. For instance, the compiler would generate (among others) the implied mode `append(in, in, in)` from the declarations given above.

### 3.3 Determinism

Associated with mode declarations are *determinism* declarations, which define the number of answers a procedure will produce. For instance, in the first mode of `append` we can include the determinism declaration `det`, meaning that mode of `append` always produces exactly one answer. This is because there can only be one result from catenating two lists.

```
:- mode append(in, in, out) is det.
```

The second mode,

```
:- mode append(out, out, in) is multi.
```

can produce many answers (mode `multi`), since any non-empty list can be decomposed in several ways (e.g., `[1,2]` can be decomposed into `([1,2], [])`, `([1], [2])`, and `([], [1,2])`).

The Mercury compiler can infer the determinism of a procedure, and thus the programmer need not supply a determinism declaration. Because the declarations are optional and we do not have determinism information in our wide-spectrum language, we avoid generating determinism declarations. However, if such declarations are desired (for instance, procedures that are exported from a Mercury module must have an explicit determinism declaration), the Mercury compiler could be used by the translation process to generate the declaration.

### 3.4 Goal ordering

From the perspective of the refinement calculus, the most attractive feature of Mercury is that the compiler can, in the majority of cases, determine an ordering of goals that ensures a procedure correctly implements the desired mode. The tradeoff is that the compiler needs type and mode declarations for every procedure. In a formal methods setting, however, this is a small price to pay for correct code. There is no guarantee that the compiler *can* find an ordering that will satisfy the declaration, though if it cannot the compiler will report an error.

As a simple example of where goal ordering is important, consider the following query for some procedure `p`.

```
t :- p(X), X = 0.
```

If `p` goes into an infinite loop when passed an uninstantiated parameter, the procedure `t` will not terminate. However, by swapping the order of the conjuncts thus:

```
t :- X = 0, p(X).
```

a call to `t` will terminate, assuming that `p` terminates when passed a ground parameter. Mercury will perform this ordering internally; to the programmer, either ordering of goals is fine.

Another problem with termination may arise in logic programming languages depending on the ordering of clauses (disjuncts), since disjunction is not commutative in the operational semantics. It is possible for the ordering of clauses to cause a procedure to go into an infinite loop without producing any answers when passed an uninstantiated variable, though with some other ordering the procedure would work correctly. In our calculus, however, the rules for introducing recursion prohibit the derivation of programs that would generate infinite loops of this kind. To satisfy the well-founded ordering inherent in the laws for recursion, the recursive parameter must be ground. This guarantees that the recursion will terminate regardless of the ordering of the disjuncts.

## 4 The translation tool

To bridge the gap between an “executable” wide-spectrum program and code, we developed a tool that translates a (subset of all possible) wide-spectrum programs into compilable and runnable Mercury code. The translator itself is written in Mercury, and may be invoked from Marvin or via a command-line interface.

We discuss what it means for a wide-spectrum program to be “executable”, that is, in a form amenable to translation, in Sect. 4.1. The translation of type definitions and programs are discussed in Sections 4.2 and 4.3 respectively. We provide examples in Sect. 4.4.

### 4.1 Executable wide-spectrum programs

The refinement of a wide-spectrum program ends when the program contains only implementation constructs. However, determining when a logic program is executable (via say the Prolog interpreter) is complicated by the ordering of goals. At the logical level, goals are joined via conjunction, which is commutative. The Prolog implementation of conjunction is not, in general, commutative (we demonstrated this with the `t` program in Sect. 3.4). Thus the refinement process (or, alternatively, the translation process), must take into account the operational semantics of Prolog, if Prolog is to be the implementation language.

By choosing an implementation language such as Mercury, however, this complication is avoided. The Mercury compiler can order the goals of a program to ensure correct execution, if there is such an ordering.

The overhead of using Mercury is that the types and modes of formal parameters must be explicitly declared. Thus we require that a program that is to be translated to Mercury include the types of the parameters, with the mode information determined by whether the types are assumed (via assumptions) or guaranteed (via specifications). We therefore impose structure on programs that are to be translated to Mercury code. For simplicity we assume that all type predicates are called “*isT*”, where `T` is the name of the type (e.g., `isIntTree`). A program with body `S`, with the types of the input parameters `I` given by the predicate  $\mathcal{T}_i(I)$ , and the types of the output parameters `O` given by the predicate  $\mathcal{T}_o(O)$ , must be of the following form:

$$\{\mathcal{T}_i(I)\}, ((\mathcal{T}_o(O)) \wedge S)$$

Either of the type declarations may be omitted if not needed. The types declared in  $\mathcal{T}_i$  and  $\mathcal{T}_o$  must either be defined in Mercury, or have their definitions translated to Mercury (as described in Sect. 4.2).

For example, the body of the recursive block for *treeMember* from Fig. 3 is in this form.

$$\{isIntTree(T)\}, ((isInt(Y)) \wedge \dots)$$

The input parameter  $T$  is a tree, and the output parameter  $Y$  is an integer.

Within the body of the program ( $S$ ), the goals of the program may be composed from procedure calls or specifications. Assumptions are not translated to code, since they do not have any effect on free variables. Procedure calls are straightforward to translate, since the syntax is the same in both languages. Specifications are translated to their defining predicate, which must be an equality or an inequality, and which again are straightforward to translate.

## 4.2 Type translation

As an example of the translation of types to Mercury definitions, consider the definition of *isIntTree* (a tree of integers) below.

$$\begin{aligned} isIntTree(T) == \\ & T = empty \vee \\ & (\exists L, X, R \bullet isInt(X) \wedge \\ & \quad isIntTree(L) \wedge isIntTree(R) \wedge \\ & \quad T = tree(L, X, R)) \end{aligned}$$

We translate this to a Mercury “discriminated union”. The name of the Mercury type is derived from the name of the type predicate, by stripping the *is* and uncapitalising the first letter. Each disjunct in the predicate type corresponds to a constructor of the discriminated union. The formal parameter (in the above case,  $T$ ) is instantiated to some term, e.g., *empty* in the first disjunct of *isIntTree*, and any existentially quantified variables are declared to be of some type, e.g.,  $L$  and  $R$  are trees of integers, and  $X$  is an integer.

We construct a mapping from existentially quantified variables to their types, and use this mapping to substitute variables with their types in any compound terms. For example, the term *tree(L, X, R)* in the second disjunct is translated to the term `tree(intTree, int, intTree)`.

The translator produces the following output from the predicate *isIntTree*.

```
:- type intTree --->
    empty
    ; tree(intTree, int, intTree).
```

## 4.3 Program translation

The translator uses Mercury parsing procedures to construct an abstract syntax tree of the wide-spectrum program, on which a sequence of six translation steps are performed, described below. Steps 1 and 2 make some minor modifications to the wide-spectrum program, before a Mercury program is generated by Steps 3 and 4. Some simplifications are made to the program to make it more readable in Steps 5 and 6.

1. Rename local variables so that all variable names are unique.

The wide-spectrum language contains explicit existential quantification of local variables, but in Mercury such quantification is optional (and is generally omitted for readability). Thus we must ensure that any name clashes are resolved, e.g., where a local variable name clashes with a parameter name. This is done by appending an integer onto the variable name that ensures it is unique.

For instance the program (fragment):

$$(\exists X \bullet p(I, X)) \wedge (\exists X \bullet q(I, X))$$

is translated to

$$(\exists X \bullet p(I, X)) \wedge (\exists X1 \bullet q(I, X1))$$

2. Extract the type and mode information for the parameters.

Variables in the wide-spectrum language are not explicitly typed, thus type information must be extracted from the program.

- (a) Parameters appearing in assumptions become input parameters (Mercury mode `in`).
- (b) Parameters appearing in specifications become output parameters (Mercury mode `out`).

The type predicates that have been used for type and mode declarations are removed from the wide-spectrum program. Lists that contain the Mercury types and modes which correspond to the list of formal parameters are constructed, forming the type and mode declarations of the final program.

For example, from the body of the recursive block for *treeMember* from Fig. 3, we determine that  $T$  is of type `intTree` and is an input (mode `in`), and that  $Y$  is an integer output (mode `out`). This generates the following type and mode declarations.

```
:- pred treeMember(int, intTree).
:- mode treeMember(out, in).
```

Note that the compiler will generate implied modes for the procedure if any of the parameters are outputs. For example, the implied mode `treeMember(in, in)` would be generated from the mode `treeMember(out, in)`.

3. Translate the parameters into Mercury terms. This is the first step that translates wide-spectrum constructs into Mercury constructs. There is a simple one-to-one correspondence between wide-spectrum and Mercury terms.
4. Translate the procedure body into a Mercury program. If a wide-spectrum construct (program, predicate, or term) has no Mercury equivalent, an error is reported.

- (a) Syntactic translation of terms. We allow arbitrary terms in prefix notation.
- (b) Syntactic translation of specifications to their defining predicates. We allow equalities and inequalities as atomic Mercury predicates. Logical conjunction and disjunction within specifications are translated to Mercury conjunction and disjunction (see step 4d below).
- (c) Syntactic translation of procedure calls. A procedure call in the wide-spectrum language translates to a procedure call in Mercury.
- (d) Syntactic translation of program operators. Parallel and sequential conjunction are translated to Mercury conjunction (`‘, ’`) and disjunction to Mercury disjunction (`‘; ’`).

```

treeMember(Y,T) :- T = tree(L,X,R),
  (X < Y, treeMember(Y,L)) ;
  Y = X ;
  (X > Y, treeMember(Y,R)).

```

Figure 5: The *treeMember* program after step 4

- (e) Existential quantification is made implicit. Since step 1 has been performed, we may remove existential quantification from the program.

- (f) Recursion is made implicit.

A recursion block in the wide-spectrum language is of the form **re**  $p \bullet V :- \mathcal{C}(p)$  **er**. We remove the **re..er** delimiters and use the name  $p$  as the name of the procedure. For example, the recursion block

```
re treeMember • (Y, T) :- ... er
```

is translated to

```
treeMember(Y,T) :- ...
```

At the end of step 4 the program is a valid Mercury program. The result of translating the *treeMember* program after step 4 is shown in Fig. 5.

5. Split disjunctions into separate clauses.

A wide-spectrum procedure does not have separate clauses, though it is common programming style to separate top-level disjunctions into clauses. For example, the program in Fig. 5 is translated to the following:

```

treeMember(Y,T) :- T = tree(L,X,R),
  X < Y, treeMember(Y,L).
treeMember(Y,T) :- T = tree(L,X,R),
  Y = X.
treeMember(Y,T) :- T = tree(L,X,R),
  X > Y, treeMember(Y,R).

```

6. Head unification and simplification.

In this step we simplify the clauses, based on variables appearing in equalities. We rename variables that appear in equalities in the body and the head, and remove the equality goals.

For example, we translate *treeMember* to the following.

```

treeMember(Y, tree(L, X, R)) :-
  X < Y, treeMember(Y, L).
treeMember(X, tree(L, X, R)).
treeMember(Y, tree(L, X, R)) :-
  X > Y, treeMember(Y, R).

```

We have replaced the parameter  $T$  with  $\text{tree}(L,X,R)$  in all the clauses, and also replaced  $Y$  with  $X$  in the second clause.

The program may now be output to a file. In the printing of the program, some final translation is performed to take advantage of Mercury functions. For example, a term  $\text{add}(X,Y)$  is displayed as  $X + Y$ .

## 4.4 Examples

### 4.4.1 treeMember

Recall the program *treeMember* (Fig. 3).

```

re treeMember • (Y, T):-
  {isIntTree(T)}, ((isInt(Y)) ∧
  (∃ L, X, R • (T = tree(L, X, R)),
  ((X < Y) ∧ treeMember(Y, L)) ∨
  {Y = X} ∨
  ((X > Y) ∧ treeMember(Y, R))))
er

```

The program in internal translator syntax is given in Fig. 6. The syntax **pand**, **sand**, **por**, and **exists** correspond to program parallel conjunction, sequential conjunction, disjunction, and existential quantification respectively. Specifications and assumptions are modeled using **spec** and **assert** respectively, and recursive blocks and procedure calls using **rec** and **wspcall**. The **varT** and **funT** syntax distinguishes terms that are variables and functions. Following usual conventions, we begin variable and functor names with upper and lower-case letters respectively, though the translator will perform any necessary lower to upper case adjustments before outputting the program.

As shown in Sect. 4.3, the translator produces the following output from the input given in Fig. 6.

```

:- pred treeMember(int, intTree).
:- mode treeMember(out, in).

treeMember(Y, tree(L, X, R)) :-
  X < Y, treeMember(Y, L).
treeMember(X, tree(L, X, R)).
treeMember(Y, tree(L, X, R)) :-
  X > Y, treeMember(Y, R).

```

The code for *treeMember* is runnable in Prolog. However, the ordering of the goals is important; in Prolog, this code may not be used to output all the elements in the tree (that is, it may not be used with the mode *treeMember(out, in)*). The problem arises since the parameters to the primitives ' $<$ ' and ' $>$ ' cannot be treated as outputs, which will occur with this ordering of goals. Hence the naive translation to Prolog will only work correctly with  $Y$  as an input, even though our specification contains no such constraint. Such problems are avoided by using Mercury, as goal ordering to satisfy mode declarations is done transparently by the compiler.

### 4.4.2 reverse

In this section we present an example where we are unable to generate Mercury code from a specification. Consider the following procedure that defines the relationship between the lists  $L$  and  $R$  where  $L$  is the reverse of  $R$ .

```

re rev • (L, R):-
  {isList(L) ∧ isList(R)} ∧
  ((L = [] ∧ R = []) ∨
  (∃ H, T • (L = [H | T]),
  (∃ RT • append(RT, [H], R) ∧
  rev(T, RT))))
er

```

We assume the predicate *isList* is true iff its parameter is a list, and we assume the existence of a procedure *append/3*.

We have declared that  $L$  and  $R$  are lists by use of a specification, indicating that both may be outputs. Given the above program, the translator produces the following:

```

wsp_program_def("treeMember", rec("treeMember", [varT("Y"), varT("T")],
  assert(isIntTree(varT("T"))) 'sand'
  (spec(isInt(varT("Y"))) 'pand'
  exists([varT("L"), varT("X"), varT("R")],
  spec(varT("T") == funT("tree", [varT("L"), varT("X"), varT("R")]))
  'sand'
  (
    ((spec(varT("X") << varT("Y")) 'pand'
    wpcall("treeMember", [varT("Y"), varT("L")]))
    'por'
    spec(varT("Y") == varT("X")))
    'por'
    (spec(varT("X") >> varT("Y")) 'pand'
    wpcall("treeMember", [varT("Y"), varT("R")]))
  )
  )))

```

Figure 6: Translator syntax for *treeMember*

```

:- pred reverse(list(T), list(T)).
:- mode reverse(out, out).

reverse([], []).
reverse([H|T], R) :-
  append(RT, [H], R), reverse(T, RT).

```

The above mode does not compile – Mercury does not generate code for lists whose elements are unspecified. The specification of the program is too weak, and we must strengthen the program so that at least one of the lists is input in order to generate compilable Mercury code. For example, if the *reverse* program was specified with *list(L)* inside an assumption rather than a specification, the same Mercury code would be produced, but with the (compilable) mode:

```

:- mode reverse(in, out).

```

## 5 Conclusions

In this paper we have introduced a prototype tool for the translation of programs in a wide-spectrum specification language into executable Mercury code. This tool forms part of the Marvin refinement tool, providing the step from refined logic programs to compilable code. We presented a brief description of the refinement calculus for logic programs, and an overview of the logic programming language Mercury. The translation process was described, and issues relating to generating logic program code from specifications were discussed. It was found that Mercury has many desirable features as a target implementation language, particularly since goal ordering is performed by the compiler. In general the process is a straightforward syntactic translation, complicated by the extraction of type and mode declarations from the specifications.

### 5.1 Related work

Other logic programming languages may be chosen as target implementation languages. Our earlier work investigated Prolog as an implementation language. However, issues such as clause and goal ordering meant that the translation process would be complex. Either more effort would need to be spent to match the semantics of the refinement calculus with those of the procedural semantics of Prolog, or the translator would need to be able to do the ordering itself (using a similar approach as that of the Mercury compiler).

Another logic programming language that is more attractive than Prolog as target implementation language is Gödel [Hill and Lloyd, 1994]. Gödel is similar to Mercury in that it is (almost) purely declarative

– for instance the programmer need not be concerned with goal ordering. (The non-declarative aspects of Gödel include committed-choice operators that can prune the search tree.) Gödel has a module system, and the type system is based on many-sorted logic, similar to Mercury. Gödel does not seem to be as well-known as Mercury, and for this reason we chose Mercury over Gödel for code generation.

Logic program synthesis [Deville and Lau, 1994] is a method of deriving logic programs from specifications, similar to that of the refinement calculus. Typically synthesis works directly in first-order logic, using Prolog-like syntax. If the synthesis process takes the operational semantics of Prolog into account, the final program should be directly executable, removing the need for a separate translation process. The tradeoff is the complexity of using the operational semantics of Prolog during the synthesis process. If the operational semantics of Prolog are not taken into account during synthesis, then a separate process must be undertaken to manipulate the program into executable form (cf., logic program transformation [Pettorossi and Proietti, 2000]).

Our approach of separating translation from the task of deriving a logic description of the problem is similar to the approach of Deville [Deville, 1990]. Deville includes transformation laws for improving the efficiency of the final program, which we may consider for the translator.

### 5.2 Future work

If the refiner wishes to derive a Mercury procedure from a specification, the final program must include the types of the parameters in order to generate a Mercury type declaration. Similar restrictions would be required for other logic programming languages; for example if a procedure in a constraint language was desired, the refinement process would be different in order to generate a constraint-based program. The translator could be extended to include several different styles of translation, which may be selected by the user, with the core syntactic code-translation process the same.

We have taken the approach of translating to Mercury from the refinement calculus without making any changes to the calculus itself. However if Mercury was to be the sole target implementation language of the calculus, several measures could be taken to make the wide-spectrum language more compatible with Mercury. For instance, we would use a type system similar to that of Mercury, and include mode (and determinism) declarations as part of a procedure definition. Such additions would make the translation to Mer-

cury simpler, however, the extra baggage may be a hindrance if another implementation language is desired.

## Acknowledgements

We would like to thank Richard Hagen for early work on code generation for the refinement calculus for logic programs, and members of the Mercury development team and mailing lists for help with Mercury (the Mercury home page is at <http://www.cs.mu.oz.au/research/mercury/index.html>). This work reported in this paper is supported by Australian Research Council grant number A49937007: *Refinement Calculus for Logic Programming*.

## References

- [Colvin et al., 2000] Colvin, R., Hayes, I. J., and Strooper, P. (2000). Refining logic programs using types. In Edwards, J., editor, *Australasian Computer Science Conference (ACSC 2000)*, pages 43–50. IEEE Computer Society.
- [Deville, 1990] Deville, Y. (1990). *Logic Programming: Systematic Program Development*. International series in logic programming. Addison-Wesley.
- [Deville and Lau, 1994] Deville, Y. and Lau, K.-K. (1994). Logic program synthesis. *Journal of Logic Programming*, 19,20:321–350. Special Issue: Ten Years of Logic Programming.
- [Hayes et al., 1997] Hayes, I., Nickson, R., and Strooper, P. (1997). Refining specifications to logic programs. In Gallagher, J., editor, *Logic Program Synthesis and Transformation. Proc. of the 6th Int. Workshop, LOPSTR'96, Stockholm, Sweden, August 1996*, volume 1207 of *Lecture Notes in Computer Science*, pages 1–19. Springer.
- [Hayes et al., 2000] Hayes, I., Nickson, R., Strooper, P., and Colvin, R. (2000). A declarative semantics for logic program refinement. Technical Report 00-30, Software Verification Research Centre, The University of Queensland.
- [Hemer et al., 2001] Hemer, D., Hayes, I., and Strooper, P. (2001). Refinement Calculus for Logic Programming in Isabelle/HOL. In Boulton, R. and Jackson, P., editors, *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 249–264. Springer.
- [Henderson et al., 2000] Henderson, F., Conway, T., Somogyi, Z., Jeffery, D., Schachte, P., Taylor, S., and Speirs, C. (2000). *The Mercury Language Reference Manual*. [http://www.cs.mu.oz.au/research/mercury/information/doc-release/reference\\_manual\\_toc.html](http://www.cs.mu.oz.au/research/mercury/information/doc-release/reference_manual_toc.html).
- [Hill and Lloyd, 1994] Hill, P. and Lloyd, J. (1994). *The Gödel Programming Language*. MIT Press, 1994.
- [Morgan, 1994] Morgan, C. (1994). *Programming from Specifications*. Prentice Hall, second edition.
- [Paulson, 1994] Paulson, L. C. (1994). *Isabelle: a generic theorem prover*. Springer Verlag.
- [Pettorossi and Proietti, 2000] Pettorossi, A. and Proietti, M. (2000). Automatic derivaton of logic programs by transformation. In *Lecture Notes for the 2000 European Summer School on Logic, Language, and Information (ESSLLI'2000)*, Birmingham.