Modular Logic Program Refinement *

Robert Colvin, Ian Hayes and Paul Strooper

Software Verification Research Centre and
School of Computer Science and Electrical Engineering,
University of Queensland, Brisbane, Australia

Abstract. A refinement calculus provides a method for transforming specifications to executable code, maintaining the correctness of the code with respect to its specification. Modules allow one to group together data types with sets of procedures that manipulate the data types. In this paper we develop a technique for refining a module to one that uses a more efficient representation of the data type. The technique places restrictions on the way procedures in the module use the data type, and on the way a program uses the module. The restrictions allow a more efficient implementation to be developed.

1 Introduction

The logic programming refinement calculus [2] is a method for systematically deriving logic programs from formal specifications, in a way that guarantees correct implementations. It is based on: a *wide-spectrum language*, that may 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 general predicates (specification constructs), as well as a subset that corresponds to Horn clauses (code). 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 predicates, and for introducing code constructs.

In this paper we introduce *modules* into the wide-spectrum specification language. Modules are collections of procedures that operate on an associated data type. For example, a module *Set*, using the set data type, might include procedures to add an element to a set and to check set membership.

A program that uses the *Set* module will not be directly implementable, since the set data type is not supported by logic programming languages such as Prolog. We would like to be able to replace the calls to the *Set* module with calls

* Extended Abstracts of LOPSTR 2000, Tenth International Workshop on Logic-based Program Synthesis and Transformation, 24-28 July 2000, London, UK. Technical Report Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1. http://www.cs.man.ac.uk/cstechrep/titles00.html

to a module that implements the same operations, except on an implementable type, e.g., Prolog lists. We call the module that operates on sets the *abstract* module, and the module that operates on lists the *concrete* module. To reason about such situations, we introduce a notion of module refinement, which characterizes the conditions under which procedure calls to an abstract module may be systematically replaced by the corresponding procedure calls to a concrete module.

Sect. 2 presents a brief outline of the refinement calculus for logic programs. Sect. 3 introduces modules into the wide-spectrum language, and outlines how they may be used to construct programs. Sect. 4 discusses the relationship between modules that models module refinement.

2 Refinement Calculus for Logic Programs

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.

$\langle P \rangle$ - specification
$\{\mathrm{A}\}$ - 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

Fig. 1. Summary of wide-spectrum language

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. *Assumptions* allow 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 $\{X \neq 0\}$.

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.

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 $pc(F) \mathrel{\widehat{=}} S$, where S is a wide-spectrum program. It defines the procedure called pc with a list of formal parameter names F and body S. A call on the procedure pc is of the form $pc(K)$, where K is a list of actual parameters.

2.2 Semantics

To define the semantics of the extended language we consider the effect of a program if assumptions are ignored, and the conditions under which programs are guaranteed not to abort. We do not deal directly with termination in our specification language, leaving concerns such as clause ordering as a task for the actual translation of programs in the wide-spectrum language to Prolog (or any chosen logic programming language). Such translations may use already well-developed techniques for logic program transformation.

The function *ef* gives the effect of a program as a characteristic predicate of the results computed by the program, ignoring assumptions. The effect function for the basic constructs in our language is detailed in Fig. 2(a). The function *ok* defines under what circumstances a program is guaranteed not to abort. The details of *ok* for basic constructs are given in Fig. 2(b). To define the semantics of a procedure call $pc(K)$, we assume the existence of the procedure definition $pc(F) \mathrel{\widehat{=}} S$. We then define *ok* and *ef* by substituting actual parameters K for formal parameters F in the body S (notationally, $S\left[\frac{K}{F}\right]$).

$$\begin{aligned}
ef.\langle P\rangle &\mathrel{\widehat{=}} P \\
ef.\{A\} &\mathrel{\widehat{=}} true \\
ef.(S \lor T) &\mathrel{\widehat{=}} ef.S \lor ef.T \\
ef.(S \land T) &\mathrel{\widehat{=}} ef.S \land ef.T \\
ef.(S, T) &\mathrel{\widehat{=}} ef.S \land ef.T \\
ef.(\exists X \bullet S) &\mathrel{\widehat{=}} (\exists X \bullet ef.S) \\
ef.(\forall X \bullet S) &\mathrel{\widehat{=}} (\forall X \bullet ef.S) \\
ef.pc(K) &\mathrel{\widehat{=}} ef.(S\left[\tfrac{K}{F}\right])
\end{aligned}$$

(a) The effect of program constructs

$$\begin{aligned}
ok.\langle P\rangle &\mathrel{\widehat{=}} true \\
ok.\{A\} &\mathrel{\widehat{=}} A \\
ok.(S \lor T) &\mathrel{\widehat{=}} ok.S \land ok.T \\
ok.(S \land T) &\mathrel{\widehat{=}} ok.S \land ok.T \\
ok.(S, T) &\mathrel{\widehat{=}} ok.S \land (ef.S \Rightarrow ok.T) \\
ok.(\exists X \bullet S) &\mathrel{\widehat{=}} (\forall X \bullet ok.S) \\
ok.(\forall X \bullet S) &\mathrel{\widehat{=}} (\forall X \bullet ok.S) \\
ok.pc(K) &\mathrel{\widehat{=}} ok.(S\left[\tfrac{K}{F}\right])
\end{aligned}$$

(b) Non aborting conditions of program constructs

Fig. 2. Semantics

Refinement between programs is defined by reducing the circumstances under which abortion is possible — that is, by weakening *ok* — while maintaining the effect in those cases where abortion is not possible. Hence refinement of program S by program T, written $S \sqsubseteq T$, is defined by

$$S \sqsubseteq T \ \widehat{=} \ ok.S \Rightarrow (ok.T \wedge (ef.S \Leftrightarrow ef.T))$$

For example, assuming X and Y are integers,

$$\{X \geqslant 0\}, \langle Y = abs(X) \rangle \sqsubseteq \langle Y = X \rangle$$

because $X \geqslant 0 \Rightarrow (true \wedge (Y = abs(X) \Leftrightarrow Y = X))$.

The definition of refinement above does not allow the reduction of non-determinism that imperative refinement allows; in logic programming we are interested in *all* possible solutions, and hence any refinement must also return all of those solutions.

Given two procedures pa and pc we say that pa is refined by pc if for all arguments K, $pa(K) \sqsubseteq pc(K)$.

Based on the semantics, we may prove general refinement laws to allow program derivations. For example, consider:

Law 1 *Sequential conjunction monotonic*

$$\frac{S \sqsubseteq S'}{S, T \sqsubseteq S', T}$$

The predicate above the line represents the premisses that must be satisfied before the transformation below the line can be employed. In this case, the refinement of S to S' guarantees that the sequential conjunction (S, T) refines to (S', T). Monotonicity holds for all the operators and both quantifiers in the wide-spectrum language.

3 Modules

In this section we introduce modules in the wide-spectrum language and explain how they may be used. Our module specifications define an opaque data type, and provide the type name and procedures that operate on that data type. In logic programming languages such as Prolog there is no notion of a 'hidden' or 'local' state – the current value of the data type in question must always be passed as a parameter to a procedure in the module. In order to allow a change of representation of types declared within a module, we restrict programs so that the only operations that may be used on variables of such types are the procedures declared in the module. A module procedure may have any number of parameters of such types.

To allow more powerful refinement rules we split these parameters into two categories, input and output. Upon a procedure call, input parameters have already been instantiated to the data type, and output parameters are uninstantiated. In addition, procedures may have "auxiliary" parameters, which are not of the declared type.

Below is an example module, called *Set*, that defines operations on a set of natural numbers, described by $\mathbb{F}\,\mathbb{N}$ (the set of all finite subsets of $\mathbb{N}$). The input parameters, in this case called S, have their type assumed. The output parameters, S', have their type established by the procedure. Two of the procedures also make use of the auxiliary parameter E. Note that we do not distinguish the modes of the auxiliary parameters in our specifications. For clarity the type signature of each procedure is declared.

Definition 1 *Set module*

$$
\begin{array}{l}
\textbf{Module } \mathit{Set} \\
\quad \textbf{Type}\colon \mathsf{setnat} == \mathbb{F}\,\mathbb{N} \\
\quad \mathit{init}(S'\colon \mathsf{setnat}) \\
\quad \mathit{add}(E\colon \mathbb{N}, S\colon \mathsf{setnat}, S'\colon \mathsf{setnat}) \\
\quad \mathit{mem}(E\colon \mathbb{N}, S\colon \mathsf{setnat}) \\
\\
\quad \mathit{init}(S') \mathrel{\widehat=} \langle S' = \{\}\rangle \\
\quad \mathit{add}(E, S, S') \mathrel{\widehat=} \{S \in \mathsf{setnat} \wedge E \in \mathbb{N} \wedge E \notin S\}, \langle S' = S \cup \{E\}\rangle \\
\quad \mathit{mem}(E, S) \mathrel{\widehat=} \{S \in \mathsf{setnat}\}, \langle E \in S\rangle \\
\textbf{End}
\end{array}
$$

Typically a module's procedures fall into one of several categories, based on whether the procedure has an input or an output parameter or both. Following the terminology of [3], a procedure with no input is referred to as an *initialisation* procedure, e.g., *init*. A procedure with no output is referred to as an *observer*. It will typically provide information on the value of the input parameter, e.g., *mem*. A procedure with both input and output parameters is called a *constructor*. It can be likened to updating the state in an imperative module, e.g., *add*.

3.1 Using Modules

Our intuition is that a module is to be used in the construction and maintenance of some data structure throughout multiple procedure calls. We therefore restrict the way in which a procedure from the module may be called. For illustrative purposes, consider the implementation of a program that checks whether variable X is 1 or 2, using the *Set* module.

$$
\begin{array}{l}
(\exists\, S \bullet \mathit{init}(S), (\exists\, S' \bullet \mathit{add}(1, S, S'), \\
\qquad (\exists\, S'' \bullet \mathit{add}(2, S', S''), \mathit{mem}(X, S''))))
\end{array} \tag{1}
$$

The use of sequential conjunction reflects the notion of the changing state and also allows us to satisfy the type assumptions of the calls. Initially, S is instantiated to the empty set. The two calls to *add* update S to S' and then S''. Overall, the only variable we are interested in is X - the input and output parameters are local and are existentially quantified when they are first used as an output.

We focus on programs of this form, where the module procedure calls provide a context for the other module procedure calls. We can use this contextual information to derive more efficient implementations of the module. The form of programs we consider always existentially quantify the output of a procedure call p. We say such a program is in 'output quantified form', i.e.,

$$(\exists\, O \bullet p(V, I, O), \mathcal{C}) \tag{2}$$

where V, I, and O are the auxiliary, input, and output parameters respectively. $\mathcal{C}$ is any sub-program that also respects the form of (2) for calls on a module procedure. The input parameter appears free, and this implies that it must have already been used as the output parameter of another module procedure - in fact, the initialisation procedure must have been called to start with since it does not have an input parameter. In (1), *mem* does not have an output parameter, and therefore there is no quantification associated with it. Corresponding with our intuition about output parameters being uninstantiated before they are used, O must have no assumptions made about it by p. Our approach also allows more general programs, however, the interesting refinement cases arise with this more restrictive form. Hence, programs written in output quantified form are sufficient to illustrate our methods.

4 Module Refinement

The framework for modules, and the restrictions placed on their use, allows the development of a refinement rule that allows efficient implementations to be developed for such modules.

Consider the *Set* module in Definition 1. A program that uses it, e.g., the program in (1), is not directly implementable, since the *Set* module uses the abstract type setnat. We would like to be able to replace the calls to *init*, *add*, and *mem* from the *Set* module with corresponding calls from a module that implements the operations on an implementable data type, e.g., a Prolog list. Of course, replacing the references to the abstract module with references to the concrete module must result in a refinement of the program in question. We refer to the module with an abstract type as the abstract module, and the implementable module as the concrete module.

In order to relate the abstract and concrete types a coupling invariant is created, similar to the processes outlined in [1, 5]. For example, if S is a set and L is a list, one possible coupling invariant is $S = \operatorname{ran} L$, where ran (short for range) returns the set of elements in a list. Recall the *add* procedure from Definition 1. Assuming the existence of the type listnat, representing a list of natural numbers, a specification of procedure *add* using lists could be

$$add^{+}(E, L, L') \mathrel{\widehat{=}} \{L \in \mathsf{listnat} \wedge E \in \mathbb{N} \wedge E \notin \operatorname{ran} L\}, \langle L' = [E \mid L] \rangle$$

However, the concrete implementation is not a refinement of the abstract implementation. The definition of refinement says that the effects of the two program must be equivalent. Even assuming that the corresponding inputs and outputs are related via the coupling invariant, we must show:

$$\begin{array}{l}(S = \operatorname{ran} L \wedge S' = \operatorname{ran} L') \Rightarrow \\ \qquad S' = S \cup \{E\} \Leftrightarrow L' = [E \mid L]\end{array}$$

The equivalence does not hold — it holds only in the $\Leftarrow$ direction. There are many list representations for any non-empty set. The refinement relation requires that every one of them must be established (for example, we could have had E being appended to the end of L). Since we are not directly interested in the intermediate input and output parameters, we would prefer not to enumerate each possible list representation of a set.

Fortunately, for programs that are in output quantified form (2), we do not have to relate each pair of procedures so directly. Superscripting references to concrete variables and procedures with '+', we wish to show, for all sub-programs $\mathcal{C}$,

$$\frac{CI(I, I^+)}{(\exists\, O \bullet p(V, I, O), \mathcal{C}) \sqsubseteq (\exists\, O^+ \bullet p^+(V, I^+, O^+), \mathcal{C}^+)} \tag{3}$$

assuming a similar relationship holds between $\mathcal{C}$ and $\mathcal{C}^+$. Given that the coupling invariant (CI) holds between the abstract and concrete input variables, the abstract implementation of p may be replaced by the concrete implementation, p^+. Note that we have also changed the quantification to be over the concrete output parameter, and that $\mathcal{C}$ has been refined to $\mathcal{C}^+$, which is $\mathcal{C}$ with calls to the concrete module in place of calls to the abstract.

In order to prove (3) we make the following inductive assumption (4) that $\mathcal{C}$ is refined by $\mathcal{C}^+$ provided all free variables which appear as inputs are related to their concrete counterparts by the coupling invariant. Note that the inputs to $\mathcal{C}$ include the output, O, of p, and similarly for $\mathcal{C}^+$.

$$\frac{CI(I, I^+) \wedge CI(O, O^+)}{\mathcal{C} \sqsubseteq \mathcal{C}^+} \tag{4}$$

We outline a proof of (3) in Fig. 3, and discuss the proof obligations for each step.

Step 1. Starting with the abstract program $(\exists\, O \bullet p(V, I, O), \mathcal{C})$, we introduce the corresponding concrete procedure p^+. If p^+ has an output parameter it is existentially quantified. The proof obligations of this step require that if the coupling invariant holds for the inputs and p does not abort, then p^+ should not abort and the overall effect of the program should be unchanged, i.e., $ef.p(V, I, O) \Leftrightarrow (ef.p(V, I, O) \wedge (\exists\, O^+ \bullet ef.p^+(V, I^+, O^+)))$. Using the equivalence $(P \Leftrightarrow (P \wedge Q)) \Leftrightarrow (P \Rightarrow Q)$ this last condition can be rewritten as an implication.

$$\begin{array}{l}CI(I, I^+) \wedge ok.p(V, I, O) \Rightarrow \\ \qquad ok.p^+(V, I^+, O^+) \wedge \\ \qquad (ef.p(V, I, O) \Rightarrow (\exists\, O^+ \bullet ef.p^+(V, I^+, O^+)))\end{array} \tag{5}$$

$$
\begin{aligned}
&\exists\, O \bullet p(V,I,O), \mathcal{C}\\
\sqsubseteq\ & \text{Step 1. augment with concrete representation}\\
&\exists\, O \bullet (p(V,I,O) \wedge (\exists\, O^+ \bullet p^+(V,I^+,O^+))), \mathcal{C}\\
\sqsubseteq\ & \text{Step 2. extend scope of } O^+\\
&\exists\, O, O^+ \bullet (p(V,I,O) \wedge p^+(V,I^+,O^+)), \mathcal{C}\\
\sqsubseteq\ & \text{Step 3. apply (4)}\\
&\exists\, O, O^+ \bullet (p(V,I,O) \wedge p^+(V,I^+,O^+)), \mathcal{C}^+\\
\sqsubseteq\ & \text{Step 4. reduce scope of } O\\
&\exists\, O^+ \bullet (p^+(V,I^+,O^+) \wedge (\exists\, O \bullet p(V,I,O))), \mathcal{C}^+\\
\sqsubseteq\ & \text{Step 5. diminish abstract representation}\\
&\exists\, O^+ \bullet p^+(V,I^+,O^+), \mathcal{C}^+
\end{aligned}
$$

Fig. 3. Inductive refinement of (3)

Step 2. We extend the scope of O^+ to be the same as that of O. The new output, O^+, is assumed not to appear in $p(V,I,O)$ or in $\mathcal{C}$.

Step 3. We apply (4) to refine $\mathcal{C}$ to $\mathcal{C}^+$. The proof obligation for this step is that the combination of p and p^+ must establish the coupling invariant between the output parameters O and O^+, i.e.,

$$
\begin{aligned}
&CI(I,I^+) \wedge ok.p(V,I,O) \Rightarrow\\
&\qquad (ef.p(V,I,O) \wedge ef.p^+(V,I^+,O^+) \Rightarrow CI(O,O^+))
\end{aligned}
\tag{6}
$$

Step 4. Now the abstract variable O only appears in the call to p. We may reduce the scope of O.

Step 5. We can eliminate the abstract procedure call if, similar to the first step, we prove that the execution of p^+ guarantees that there exists a value for the output parameter for p that does not affect the constraints on V.

$$
\begin{aligned}
&CI(I,I^+) \wedge ok.p^+(V,I^+,O^+) \wedge ok.p(V,I,O) \Rightarrow\\
&\qquad (ef.p^+(V,I^+,O^+) \Rightarrow (\exists\, O \bullet ef.p(V,I,O)))
\end{aligned}
\tag{7}
$$

4.1 Example

In this section we consider a concrete representation of the *Set* module. We will use a list of natural numbers to represent the set.

Definition 2 *Concrete set module*

Module Set^+
$\quad$ **Type**: $\mathsf{listnat} == \{L: List \mid \operatorname{ran} L \subseteq \mathbb{N}\}$
$\quad$ $init^+(L\colon \mathsf{listnat})$
$\quad$ $add^+(E\colon \mathbb{N}, L\colon \mathsf{listnat}, L'\colon \mathsf{listnat})$
$\quad$ $mem^+(E\colon \mathbb{N}, L\colon \mathsf{listnat})$

$\quad$ $init^+(L') \mathrel{\widehat{=}} \langle L' = [\,]\rangle$
$\quad$ $add^+(E, L, L') \mathrel{\widehat{=}} \{L \in \mathsf{listnat} \wedge E \in \mathbb{N} \wedge E \notin \operatorname{ran} L\}, \langle L' = [E \mid L]\rangle$
$\quad$ $mem^+(E, L) \mathrel{\widehat{=}} \{L \in \mathsf{listnat}\}, \langle E \in \operatorname{ran} L\rangle$
End

As with the abstract *Set* module, E is an auxiliary parameter. Input parameters are called L and output parameters are called L'. To relate this representation to the abstract representation, we choose the coupling invariant $S = \operatorname{ran} L$.

To prove that calls to the procedures of module Set^+ may systematically replace calls to the procedures of module *Set* in any program that is in output quantified form (2), we need to prove that each pair satisfies proof obligations (5), (6) and (7). For instance, consider the proof obligation (6) for the *add* and add^+ procedures. We must show that

$$\begin{aligned} &S = \operatorname{ran} L \wedge S \in \mathsf{setnat} \wedge E \in \mathbb{N} \wedge E \notin S \Rightarrow \\ &\quad (S' = S \cup \{E\} \wedge L' = [E \mid L]) \Rightarrow S' = \operatorname{ran} L' \end{aligned}$$

which holds. In fact, the three pairs of related procedures in the two modules, *init*, *add*, and *mem*, can be shown to satisfy (5), (6) and (7). Therefore any program that is in output quantified form (2), may have all references to the *Set* module replaced systematically with the procedures from the Set^+ module.

5 Conclusion

We have introduced modules into the refinement calculus. Encapsulation of this sort has been developed for logic programs previously, e.g., [4, 6]. Our specification style is based on [3], where a module presents an interface of operations that can be performed on a data type. Our refinement approach is similar to that of Morgan [5], where non-executable constructs are allowed, i.e., specifications and assertions. This allows us to formally use the assumptions the module procedures make about their inputs during the refinement process.

In this paper we have outlined a technique for changing the representation of a data type defined by a module. The change in representation must preserve the meaning of any program that uses the module, and hence we restrict ourselves to a subset of programs that are in output quantified form (2). To prove that any such program that uses the abstract module is refined by the same program that uses the concrete module, we establish a relationship between the modules independent of the particular program. This is done by ensuring that the concrete module faithfully preserves the observable behaviour of the abstract module (via its effects on auxiliary parameters). This process allows the module to be implemented using a data structure that is more appropriate to a particular task.

The sequential nature of (2), and the resulting chain of input/output values, can be likened to the structural form of imperative programs, and therefore any program that may be implemented in this style can benefit from the techniques presented in the paper. However this subset of programs is more restrictive than necessary. For example, we can allow multiple output variables to be introduced in parallel. We may also allow input and output parameters to be passed to non-module procedures, as long as those procedures respect the restrictions also. These cases rely on proofs similar to that shown in Fig. 3.

References

1. R. Colvin, I. J. Hayes, and P. Strooper. Data refining logic programs. In Jim Grundy, Martin Schwenke, and Trevor Vickers, editors, *International Refinement Workshop and Formal Methods Pacific 1998*, pages 100–116. Springer-Verlag, 1998.
2. I. Hayes, R. Nickson, and P. Strooper. Refining specifications to logic programs. In J. Gallagher, editor, *Logic Program Synthesis and Transformation. Proc. of the 6th Int. Workshop, LOPSTR'96*, volume 1207 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 1997.
3. B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, third edition, 1986.
4. F. G. McCabe. *Logic and Objects.* Prentice Hall, 1992.
5. C. Morgan. *Programming from Specifications.* Prentice Hall, second edition, 1994.
6. M.G. Read and E. Kazmierczak. Formal program development in modular Prolog. In T.P. Clement and K.-K. Lau, editors, *Logic Program Synthesis and Transformation. Proc. of LOPSTR'91*, pages 69–93. Springer-Verlag, 1992.