Refinement of Higher-Order Logic Programs

Robert Colvin[1], Ian Hayes[2], David Hemer[1], and Paul Strooper[2]

University of Queensland, Brisbane, Australia, 4072
{ianh,pstroop}@itee.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 extend the refinement calculus for logic programs to include higher-order programming capabilities in specifications and programs, such as procedures as terms and lambda abstraction. We use a higher-order type and term system to describe programs, and provide a semantics for the higher-order language and refinement. The calculus is illustrated by refinement examples.

1 Introduction

The logic programming refinement calculus [5] provides a method for systematically deriving logic programs from formal specifications. It is based on: a *wide-spectrum language* [12] 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, 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 specifications, and for introducing code constructs. The decision of which refinement law to apply at each step is determined by the developer. The calculus could be used as the basis for a program synthesis system [3] that would allow the developer or refiner to obtain some degree of automation, depending on the problem and the synthesis scheme chosen.

In this paper we extend the refinement calculus for logic programs so that variables may range over procedures, i.e., procedures become terms in our language. We can then make procedure calls on variables that represent procedures, and pass procedures as parameters. We may also construct procedures anonymously (lambda abstraction). We achieve this by embedding the specification language in a type system developed by Nadathur and Miller [10] for λProlog [9]. The semantics of the language and refinement are then extended to include typed variables, and procedures become special types of functions in the term language.

The paper is structured as follows. In Sect. 2 we give an overview of the type and term system presented in [10]. In Sect. 3 we extend the type system with the type *Cmd*

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 126–143, 2003.

representing programs, and present our wide-spectrum language. We also give the intuition behind the refinement relation (the notion of implementation), and some example programs. In Sect. 4 we present some examples of refining higher-order procedures. In Sections 5 - 8 we present the semantics of refinement and our higher-order wide-spectrum language. In Sect. 9 we discuss problems with allowing equality on procedure-valued terms in the context of refinement and higher-order logic programing.

2 Type System

In this section we present a general introduction to the type system described by Nadathur and Miller [10] for λProlog. In Sect. 3 we discuss how the type system is used to represent commands in our wide-spectrum language. The system is based on Church's Simple Theory of Types. It includes base types as well as functional types, and allows abstraction and application as terms.

2.1 Types

We use the following definitions.

Name	Represents	Examples
$\mathcal{S}$	base types	$\mathbb{Z}, \mathbb{B}$
$\mathcal{C}$	type constructors	$list/1, set/1, \rightarrow /2$
Types	all allowed types	$\mathbb{Z}, list(\mathbb{Z}), \mathbb{Z} \rightarrow \mathbb{B}$

The set $\mathcal{S}$ includes all the base types, such as integers ($\mathbb{Z}$) and booleans ($\mathbb{B}$). The set $\mathcal{C}$ is the set of all type constructors, that is, functors which take types as arguments and return a type. For example, *list* is a constructor which takes a type, e.g., $\mathbb{Z}$, and returns a type representing a list of elements of $\mathbb{Z}$. Typically we only need to be concerned with the set *Types*. This includes every base type and every possible application of the type constructors. In particular, it includes every possible function from a type to a type, for instance $\mathbb{Z} \rightarrow list(\mathbb{Z})$, and $list(\mathbb{Z}) \rightarrow list(list(\mathbb{Z} \rightarrow \mathbb{Z})) \rightarrow \mathbb{B}$. The function type constructor '$\rightarrow$' associates to the right.

We assume a set of variables, *Var*, which have an associated type, and a set of constants (including functions) of given types. There is at least one variable and constant of each type.

We assume a base type *Pred*, representing first-order predicates. The operators for the type *Pred* include conjunction, disjunction, etc., and the existential and universal quantifiers. We assume a rich set of mathematical operators, including equality and arithmetic operators, is available in our predicate language. The semantics of predicates is discussed in more detail in Sect. 5.2.

2.2 Terms

We define a *term* in this language as follows.

1. a constant or variable of type σ is a term of type σ.

2. $(\lambda X{:}\,\sigma \bullet E)$, where X is a variable of type σ and E is a term of type τ, is a term of type $\sigma \to \tau$.
3. $F(E)$, where F is a term of type $\sigma \to \tau$ and E is a term of type σ, is a term of type τ.

Thus a term can be a variable or a constant, a lambda abstraction, or an application. For example, given the integers as constants of type $\mathbb{Z}$, the (constant) empty list of integers $[\,]_{\mathbb{Z}}$, and the constructor $[.\mid .]_{\mathbb{Z}}$ for lists of integers, we can write terms such as $[1 \mid [2 \mid [\,]_{\mathbb{Z}}]_{\mathbb{Z}}]_{\mathbb{Z}}$. In the rest of the paper we drop the subscript on the list constants and use the usual list notation for lists of any type, though in practice there are different constructors for each distinct list type. Note that badly typed function applications are not terms.

3 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. To make the use of higher-order constructs easier, we have embedded our language in the type system described in Sect. 2. We introduce a new base type, *Cmd*, representing all possible commands that may be constructed in our language. We describe the basic constructors of the language in the next section, then discuss how procedures are treated in the framework, allowing reasoning about higher-order constructs. Our base language is similar to that presented in [5], except that in the new type system, quantified variables and parameters to procedures must be typed.

3.1 Basic Constructs

A summary of the basic constructs of the language is shown in Fig. 1.

Cmd	Type		Example
$\{\,\}$	$\mathit{Pred} \to \mathit{Cmd}$	assumption	$\{X \neq 0\}$
$\langle\,\rangle$	$\mathit{Pred} \to \mathit{Cmd}$	specification	$\langle Y = 2 * X\rangle$
$\vee$	$\mathit{Cmd} \to \mathit{Cmd} \to \mathit{Cmd}$	disjunction	$S \vee T$
$\wedge$	$\mathit{Cmd} \to \mathit{Cmd} \to \mathit{Cmd}$	parallel conj.	$S \wedge T$
,	$\mathit{Cmd} \to \mathit{Cmd} \to \mathit{Cmd}$	sequential conj.	S, T
$\exists_\sigma$	$(\sigma \to \mathit{Cmd}) \to \mathit{Cmd}$	existential quant.	$(\exists X{:}\,\mathbb{Z} \bullet S)$
$\forall_\sigma$	$(\sigma \to \mathit{Cmd}) \to \mathit{Cmd}$	universal quant.	$(\forall X{:}\,\mathbb{Z} \bullet S)$

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.

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 $\{X \neq 0\}$.

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. For brevity, the existential quantifier $(\exists_\sigma(\lambda X{:}\sigma \bullet S))$ will be written in the usual way, i.e., $(\exists X{:}\sigma \bullet S)$. It generalises disjunction, computing the union of the results of S for all possible values of X of type σ. Similarly, the universal quantifier $(\forall X{:}\sigma \bullet S)$ computes the intersection of the results of S for all possible values of X of type σ. Note that there are an infinite number of quantifiers, as there is an $\exists_\sigma$ and $\forall_\sigma$ for each type σ.

The following *Cmd* is an example of a program that can be constructed in our language.

$$\{X, Z \in \mathbb{Z} \wedge X \neq 0\}, \langle Y = Z \operatorname{div} X \rangle$$

The program assumes that the variables X and Z are bound to integers, and that X is non-zero, then establishes the relation that Y is the whole number division of Z by X.

3.2 Procedures

A procedure in the wide-spectrum language is a function whose result type is *Cmd*, and whose argument types are *not* of type *Cmd* or *Pred* (i.e., the constructors in Fig. 1 are not procedures). In addition, a procedure must be a closed term, that is, contain no free variables.

A summary of some relevant procedure-related constructs is given in Fig. 2. We discuss each below, and describe the method we use for procedure definition.

Syntax		Type
$pc(T)$ or $P(T)$	procedure call	*Cmd*
$(\lambda V{:}\mathit{list}(\mathbb{Z}) \bullet S)$	non-recursive proc.	$\mathit{list}(\mathbb{Z}) \rightarrow \mathit{Cmd}$
$\mu P \bullet (\lambda V{:}\mathbb{Z} \bullet \ldots P(X) \ldots)$	recursive proc.	$\mathbb{Z} \rightarrow \mathit{Cmd}$
$id \mathrel{\widehat{=}} proc$	procedure definition	

Fig. 2. Procedure-based constructs

Procedure Call. A procedure call is the application of a procedure to parameters, and is a term of type *Cmd*. Note that we allow application of procedure variables; i.e., if P is a procedure variable, $P(T)$ is a *Cmd* (variables are disallowed as functors in some versions of Prolog).

Non-recursive Procedures. A non-recursive procedure is a term of the form $(\lambda X{:}\sigma \bullet S)$, where S is a wide-spectrum program and X is a parameter to the procedure of type σ (a procedure may have multiple parameters, expressed in the usual way).

Recursive Procedures. We use the least fix-point operator μ to define the meaning of a recursive procedure. We use the following notation:

$$\mu P \bullet (\lambda X{:}\sigma \bullet \mathcal{C}(P))$$

The body of the procedure, $\mathcal{C}$, encodes zero or more recursive calls to P.

Procedure Definition. A procedure definition has the form $p \mathrel{\widehat{=}} proc$, where p is some name and *proc* is a procedure. For example, we may define a procedure *double* that doubles an integer:

$$double \mathrel{\widehat{=}} (\lambda N{:}\mathbb{Z}, N'{:}\mathbb{Z} \bullet \{N \in \mathbb{Z}\}, \langle N' = N * 2\rangle)$$

Note that the syntax for a procedure definition ($\widehat{=}$) just introduces a shorthand for the procedure itself; the names of the procedures are not semantic entities in our system. The type of *double* is $\mathbb{Z} \to (\mathbb{Z} \to Cmd)$.

We do not allow terms representing commands to appear inside predicates. This does not remove the ability to do any of the things we would normally like in our higher-order programming language – it is our wide-spectrum programming language that we are making higher-order, not the predicate language (allowing higher-order predicates, that is, predicate variables to range over predicates, is different from allowing predicate variables to range over commands). Allowing the wide-spectrum language constructs to appear inside predicates would unnecessarily complicate the predicate language and its semantics.

3.3 Refinement

To model the notion of implementation, we define the refinement operator, '$\sqsubseteq$'. We say a program S refines to a program T, written $S \sqsubseteq T$, if T terminates more often that S (w.r.t. its *assumptions*), and if T preserves the same effect on its free variables. A formal definition of refinement is presented in Sect. 7.

We present a number of derived refinement laws below. Each law represents a refinement (synthesis/transformation) that may be made. Where a law is divided into two parts divided by a horizontal line, the part above the line is the proof obligation that must be satisfied for the refinement below the line to be applied.

Law 1 *Equivalent specifications*

$$\frac{P \equiv Q}{\langle P\rangle \sqsubseteq \langle Q\rangle}$$

We can refine a specification by transforming its predicate under logical equivalence.

Law 2 *Weaken assumption*

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

We may weaken an assumption by transforming its predicate under implication.

Law 3 *List case analysis*

$$\begin{array}{l} \{L \in list(\sigma)\}, S \\ \sqsubseteq \{L \in list(\sigma)\}, ((\langle L = [\,]\rangle \wedge S) \vee \\ \qquad (\exists H{:}\,\sigma, T{:}\,list(\sigma) \bullet \langle L = [H \mid T]\rangle \wedge S)) \end{array}$$

A program S which has an associated assumption that variable L is a list may be split into the cases where L is empty and non-empty. This law has no proof obligations.

Law 4 *Recursion introduction*
Where $(\sigma, \prec)$ *is a well founded set,*

$$\frac{\forall X{:}\,\sigma \bullet \big(\forall Y{:}\,\sigma \bullet \{Y \prec X\}, pc(Y) \sqsubseteq id(Y)\big) \Rightarrow pc(X) \sqsubseteq \mathcal{C}(id)}{pc \sqsubseteq (\mu\, id \bullet (\lambda X{:}\,\sigma \bullet C(id)))}$$

This law is similar to the one presented and proved in [5], and follows from the principle of well-founded induction.

In order to apply Law 4, we must prove that the predicate above the line holds. This is done by showing that $pc(X) \sqsubseteq \mathcal{C}(id)$, assuming that

$$(\forall Y{:}\,\sigma \bullet \{Y \prec X\}, pc(Y) \sqsubseteq id(Y)) \tag{1}$$

We call (1) the inductive hypothesis. It permits refinements to recursive calls as long as the parameter Y is less than X according to some well-founded relation $\prec$ on σ, ensuring that the recursion will terminate. The program $\mathcal{C}(id)$ is just some program that may involve calls on the procedure id. Note that if $pc \mathrel{\widehat{=}} (\lambda X{:}\,\sigma \bullet S)$ then $pc(X)$ is equivalent to the procedure's body, S.

The law is used in the following steps:

1. Focus on the body of pc, S.
2. Refine S, possibly using the inductive hypothesis to introduce a call to id.
3. Call this refined program $\mathcal{C}(id)$.
4. Then, the proof obligation for the law has been proved (by instantiating $\mathcal{C}$), and the original non-recursive procedure pc has been refined to the recursive procedure $(\mu\, id \bullet (\lambda X{:}\,\sigma \bullet C(id)))$.

The second step will in general be complex, involving user direction as with most program derivations. The other steps in the process are trivial, including the construction of the inductive hypothesis, which is determined by the syntactic form of the specification.

3.4 Example Specification and Implementation

Consider the following specification of the standard higher-order procedure *map*, that applies a procedure P to all the elements in a list L, returning the list L'.

Definition 1 *Map*

$$\begin{array}{l} map \mathrel{\widehat{=}} \lambda P{:}\,\sigma \to \tau \to Cmd, L{:}\,list(\sigma), L'{:}\,list(\tau) \bullet \\ \quad \{L \in list(\sigma)\}, \\ \quad \langle \#L = \#L' \rangle \wedge (\forall i{:}\,1..\#L \bullet P(L(i), L'(i))) \end{array}$$

Note the assumption that L is a list. The type of L is given by its type declaration in the parameter list, but to guarantee that L is instantiated (i.e., not unbound) the assumption must be included. This allows us to implement the procedure using recursion. For the well-founded ordering we use $<$ on the length of the list parameter L. The assumption $\{L \in \mathit{list}(\sigma)\}$ ensures L is bound; without the assumption, L could be unbound, and the recursion would not terminate.

We briefly outline the refinement of *map* below. We wish to refine it to a recursive procedure, and therefore use Law 4. Thus we must discharge the proof obligation, which involves refining the body of *map* assuming the following inductive hypothesis. We use m as the name of the recursive call.

$$\begin{array}{l}(\forall T: \mathit{list}(\sigma), T': \mathit{list}(\sigma), P: \sigma \to \tau \to \mathit{Cmd} \bullet \\ \quad \{\#T < \#L\}, \\ \quad \{T \in \mathit{list}(\sigma)\}, \\ \quad \langle \#T = \#T' \rangle \land (\forall i: 1..\#T \bullet P(T(i), T'(i))) \\ \quad \sqsubseteq m(P, T, T'))\end{array} \tag{2}$$

First we refine the body of *map* (Definition 1) by splitting into the cases where L is empty and non-empty (Law 3) and simplifying.

$$\begin{array}{l}\{L \in \mathit{list}(\sigma)\}, \\ \langle L = [\,] \land L' = [\,] \rangle \lor \\ (\exists H: \sigma, H': \tau, T: \mathit{list}(\sigma), T': \mathit{list}(\tau) \bullet \\ \quad \langle L = [H \mid T] \land L' = [H' \mid T'] \rangle, \\ \quad \langle \#T = \#T' \rangle \land (\forall i: 1..\#L \bullet P(L(i), L'(i))))\end{array}$$

Now we split the universal quantification over i in the range $1..\#L$ into the cases where $i = 1$ and i is in the range $2..\#L$. We note that $L(1) = H$ and $L'(1) = H'$, and that indexing L in the range $2..\#L$ is equivalent to indexing its tail T in the range $1..\#T$ (and similarly for L' and T'). We also add some assumptions about T, which we do by noting that $L \in \mathit{list}(\sigma)$ and $L = [H \mid T]$.

$$\begin{array}{l}\{L \in \mathit{list}(\sigma)\}, \\ \langle L = [\,] \land L' = [\,] \rangle \lor \\ (\exists H: \sigma, H': \tau, T: \mathit{list}(\sigma), T': \mathit{list}(\tau) \bullet \\ \quad \langle L = [H \mid T] \land L' = [H' \mid T'] \rangle, \\ \quad P(H, H') \land \\ \quad \{\#T < \#L\}, \\ \quad \{T \in \mathit{list}(\sigma)\}, \\ \quad \langle \#T = \#T' \rangle \land (\forall i: 1..\#T \bullet P(T(i), T'(i))))\end{array}$$

Note that the bottom three lines match the left side of the inductive hypothesis (2), and therefore we can use it to introduce a call to m.

$$\begin{array}{l}\{L \in \mathit{list}(\sigma)\}, \\ \langle L = [\,] \land L' = [\,] \rangle \lor \\ (\exists H: \sigma, H': \tau, T: \mathit{list}(\sigma), T': \mathit{list}(\tau) \bullet \\ \quad \langle L = [H \mid T] \land L' = [H' \mid T'] \rangle, \\ \quad P(H, H') \land m(P, T, T')\end{array}$$

The above refinement steps comprise the proof obligation for Law 4, and thus we may refine the original procedure *map* to:

$$\begin{array}{l}\mu\, m \bullet \lambda P{:}\,\sigma \to \tau \to \mathit{Cmd}, L{:}\,\mathit{list}(\sigma), L'{:}\,\mathit{list}(\tau) \bullet \\ \quad \langle L = [\,] \land L' = [\,]\rangle \lor \\ \quad (\exists H{:}\,\sigma, H'{:}\,\tau, T{:}\,\mathit{list}(\sigma), T'{:}\,\mathit{list}(\tau) \bullet \\ \qquad \langle L = [H \mid T] \land L' = [H' \mid T']\rangle, \\ \qquad P(H, H') \land m(P, T, T'))\end{array}$$

Now that we have an implementation for *map*, we want to be able to refine programs like

$$\{X \in \mathit{list}(\mathbb{Z})\}, \langle \#X = \#X'\rangle \land (\forall i{:}\,1..\#X \bullet \mathit{double}(X(i), X'(i)))$$

to $\mathit{map}(\mathit{double}, X, X')$. This refinement is trivial by folding, i.e., pattern matching with the specification of *map* (though in general, higher-order matching is non-trivial [2]).

More generally, we have the following refinement law.

Law 5 *Parameter application.* Given $pc \mathrel{\widehat{=}} (\lambda X{:}\,\sigma \bullet S)$ then

$$S[\tfrac{Y}{X}] \sqsubseteq pc(Y)$$

4 Higher-Order Refinement

In this section we provide a more complex example, in which we use some general algebraic properties to match a specification with a recursive procedure definition. Consider a procedure *foldR*, where a call $\mathit{foldR}(P, \mathit{Base})(L, \mathit{Result})$ applies procedure P, representing a binary operator, right-associatively to the list L, starting with base element *Base* (typically the identity of the binary operator represented by P), producing the answer *Result*. For example, assuming *plus* implements binary addition, i.e.,

$$\mathit{plus} \mathrel{\widehat{=}} (\lambda X, Y, Z{:}\,\mathbb{Z} \bullet \{X, Y \in \mathbb{Z}\}, \langle Z = X + Y\rangle)$$

$\mathit{foldR}(\mathit{plus}, 0)([1, 2, 3], X)$ would bind X to 6 (the result of 1+(2+(3+0))).

As a second example, given the definition

$$\mathit{snoc} \mathrel{\widehat{=}} (\lambda A{:}\,\sigma, B{:}\,\mathit{list}(\sigma), C{:}\,\mathit{list}(\sigma) \bullet \langle C = B \frown [A]\rangle)$$

a list may be reversed (inefficiently) using *foldR*:

$$\mathit{reverse}(R, R') \sqsubseteq \mathit{foldR}(\mathit{snoc}, [\,])(R, R')$$

or more succinctly,

$$\mathit{reverse} \sqsubseteq \mathit{foldR}(\mathit{snoc}, [\,])$$

This may be transformed to a more efficient version using a similar higher-order procedure *foldL*, that captures left-associativity, as shown in [14].

We define *foldR* as a procedure that takes a procedure and a base value and returns a recursive procedure.

$$\begin{array}{l} \mathit{foldR} \mathrel{\widehat{=}} (\lambda P{:}\,\sigma \to \tau \to \tau \to \mathit{Cmd}, \mathit{Base}{:}\,\tau \bullet \\ \quad (\mu \mathit{fr} \bullet \\ \quad\quad (\lambda L{:}\,\mathit{list}(\sigma), \mathit{Result}{:}\,\tau \bullet \\ \quad\quad\quad \{L \in \mathit{list}(\sigma)\}, \\ \quad\quad\quad (\langle L = [\,] \land \mathit{Result} = \mathit{Base}\rangle \lor \\ \quad\quad\quad (\exists H{:}\,\sigma, T{:}\,\mathit{list}(\sigma) \bullet \langle L = [H \mid T]\rangle, \\ \quad\quad\quad\quad (\exists R{:}\,\tau \bullet \mathit{fr}(T,R) \land P(H,R,\mathit{Result})))))) \end{array}$$

We prove a general theorem for using *foldR* to implement relations that satisfy certain properties.

Theorem 1.
Given constants Q: $\mathit{list}(\sigma) \to \tau \to \mathit{Pred}$, B: τ, *and* P: $\sigma \to \tau \to \tau \to \mathit{Pred}$, *that satisfy:*

$$Q([\,], B) \tag{3}$$

$$Q([H \mid T], N) \Leftrightarrow (\exists R{:}\,\tau \bullet Q(T,R) \land P(H,R,N)) \tag{4}$$

and a procedure op that implements P, i.e.,

$$\mathit{op} \mathrel{\widehat{=}} (\lambda H{:}\,\sigma, R{:}\,\tau, N{:}\,\tau \bullet \langle P(H,R,N)\rangle) \tag{5}$$

then the following refinement holds:

$$(\lambda L{:}\,\mathit{list}(\sigma), N{:}\,\tau \bullet \{L \in \mathit{list}(\sigma)\}, \langle Q(L,R)\rangle) \sqsubseteq \mathit{foldR}(\mathit{op}, B)$$

Note that right-associativity is encoded into properties (3) and (4). In the summation example above, Q is the relation between a list of numbers and its sum, B is 0, and P is binary addition. In the reverse example, Q is the reverse relation between lists, B is [ ], and P is the predicate

$$(\lambda A{:}\,\sigma, B{:}\,\mathit{list}(\sigma), C{:}\,\mathit{list}(\sigma) \bullet C = B \frown [A])$$

which is implemented by the procedure *snoc*.

Proof. We use Law 4, and therefore assume the following inductive hypothesis.

$$(\forall T{:}\,\mathit{list}(\sigma), N'{:}\,\tau \bullet \{\#T < \#L\}, \{T \in \mathit{list}(\sigma)\}, \langle Q(T,N')\rangle \sqsubseteq \mathit{fr}(T,N')) \tag{6}$$

We begin the refinement of the body of the procedure on the left-hand side of the refinement.

$$\begin{array}{ll} & \{L \in \mathit{list}(\sigma)\}, \langle Q(L,N)\rangle \\ \sqsubseteq & \text{case analysis on } L \text{ using Law 3 and simplifying } Q([\,],N) \text{ using (3)} \\ & \{L \in \mathit{list}(\sigma)\}, \\ & (\langle L = [\,]\rangle \land \langle N = B\rangle \lor \\ & (\exists H{:}\,\sigma, T{:}\,\mathit{list}(\sigma) \bullet \langle L = [H \mid T]\rangle \land \langle Q(L,N)\rangle)) \\ \sqsubseteq & \text{focus on second disjunct} \end{array}$$

$$
\begin{array}{l}
1 \bullet\ (\exists H{:}\,\sigma, T{:}\,\mathit{list}(\sigma) \bullet \langle L = [H \mid T]\rangle \wedge \langle Q(L,N)\rangle) \\
\quad \sqsubseteq \text{ expand } Q \text{ using (4)} \\
\qquad (\exists H{:}\,\sigma, T{:}\,\mathit{list}(\sigma) \bullet \langle L = [H \mid T]\rangle \wedge \\
\qquad\quad \langle(\exists R{:}\,\tau \bullet Q(T,R) \wedge P(H,R,N))\rangle) \\
\quad \sqsubseteq \text{ Lift quantification and conjunction} \\
\qquad (\exists H{:}\,\sigma, T{:}\,\mathit{list}(\sigma) \bullet \langle L = [H \mid T]\rangle \wedge \\
\qquad\quad (\exists R{:}\,\tau \bullet \langle Q(T,R)\rangle \wedge \langle P(H,R,N)\rangle)) \\
\quad \sqsubseteq \text{ implement using } \mathit{op} \text{ from (5)} \\
\qquad (\exists H{:}\,\sigma, T{:}\,\mathit{list}(\sigma) \bullet \langle L = [H \mid T]\rangle, \\
\qquad\quad (\exists R{:}\,\tau \bullet \langle Q(T,R)\rangle \wedge \mathit{op}(H,R,N))) \\
\quad \sqsubseteq \text{ include assumptions about } T \text{ (from } L \in \mathit{list}(\sigma) \text{ and } L = [H \mid T]) \\
\qquad (\exists H{:}\,\sigma, T{:}\,\mathit{list}(\sigma) \bullet \langle L = [H \mid T]\rangle, \\
\qquad\quad (\exists R{:}\,\tau \bullet (\{\#T < \#L\}, \{T \in \mathit{list}(\sigma)\}, \langle Q(T,R)\rangle) \\
\qquad\qquad \wedge \mathit{op}(H,R,N))) \\
\quad \sqsubseteq \text{ introduce recursive call from (6)} \\
\qquad (\exists H{:}\,\sigma, T{:}\,\mathit{list}(\sigma) \bullet \langle L = [H \mid T]\rangle, \\
\qquad\quad (\exists R{:}\,\tau \bullet \mathit{fr}(T,R) \wedge \mathit{op}(H,R,N)))
\end{array}
$$

The above refinement steps complete the proof obligation for Law 4, resulting in the procedure:

$$
\begin{array}{l}
(\mu \mathit{fr} \bullet \lambda L{:}\,\mathit{list}(\sigma), N{:}\,\tau \bullet \\
\quad \{L \in \mathit{list}(\sigma)\}, \\
\quad (\langle L = [\,]\rangle \wedge \langle N = B\rangle \vee \\
\quad (\exists H{:}\,\sigma, T{:}\,\mathit{list}(\sigma) \bullet \langle L = [H \mid T]\rangle, \\
\qquad (\exists R{:}\,\tau \bullet \mathit{fr}(T,R) \wedge \mathit{op}(H,R,N)))))
\end{array}
$$

This is equivalent to $\mathit{foldR}(\mathit{op}, B)$. □

To apply this theorem to the summation example we need the definitional properties of *sum* corresponding to (3) and (4) respectively:

$$
\begin{array}{l}
\mathit{sum}([\,], 0) \\
\mathit{sum}([H \mid T], N) \Leftrightarrow (\exists R{:}\,\mathbb{Z} \bullet \mathit{sum}(T,R) \wedge N = H + R)
\end{array}
$$

and a procedure that implements binary addition

$$
\mathit{plus} \mathrel{\widehat{=}} (\lambda X, Y, Z{:}\,\mathbb{Z} \bullet \{X \in \mathbb{Z} \wedge Y \in \mathbb{Z}\}, \langle Z = X + Y\rangle)
$$

With Q, B and P as *sum*, 0 and binary addition respectively, we can apply Theorem 1 to deduce

$$
(\lambda L{:}\,\mathit{list}(\mathbb{Z}), N{:}\,\mathbb{Z} \bullet \{L \in \mathit{list}(\mathbb{Z})\}, \langle \mathit{sum}(L,R)\rangle) \sqsubseteq \mathit{foldR}(\mathit{plus}, 0)
$$

For the example above $\mathit{op}(H,R,N)$ represented a binary function from H and R to N. Now we consider an example where *op* represents a relation where there may be more than one value of N for a given pair of values for H and R. The permutation relation on lists may be formulated to correspond with the properties (3) and (4).

$$
\begin{array}{l}
\mathit{permutation}([\,], [\,]) \\
\mathit{permutation}([H \mid T], P) \Leftrightarrow \\
\quad (\exists R{:}\,\mathit{list}(\sigma) \bullet \mathit{permutation}(T,R) \wedge \mathit{interleave}(H,R,P))
\end{array}
$$

where

$$interleave(H, R, P) \Leftrightarrow \\ (\exists F, B: list(\sigma) \bullet R = F \frown B \wedge P = F \frown [H] \frown B)$$

Given some implementation *interleaveOp* of *interleave*, we can apply Theorem 1 to deduce

$$(\lambda L: list(\sigma), P: list(\sigma) \bullet \{L \in list(\sigma)\}, \langle permutation(L, P)\rangle) \\ \sqsubseteq foldR(interleaveOp, [\,])$$

and hence for appropriately typed L and P,

$$\{L \in list(\sigma)\}, \langle permutation(L, P)\rangle \sqsubseteq foldR(interleaveOp, [\,])(L, P)$$

5 Semantics

In this section we ascribe a meaning to terms of type *Cmd*, and define formally the refinement relation. The semantics we use are a modified version of earlier work [5,4]. We begin our formal treatment of the semantics by defining the domains over which our semantics of programs are given. We present an abridged version, which we modify to cope with changes for higher-order features. The main changes occur at the lowest level, where we define the set of values in our universe, and the evaluation of a term. Once these changes are in place, the semantics for the language constructs and refinement are similar to the semantics described in [5].

5.1 Variables and Values

For each type σ (from Sect. 2.1) we assume an associated set of values Val_σ. We assume that no value is a member of two different value sets, i.e.,

$$(\forall \sigma, \tau: Types \bullet \sigma \neq \tau \Rightarrow Val_\sigma \cap Val_\tau = \{\})$$

We define *Val* as the union of all sets Val_σ.

$$Val = \bigcup \{\sigma: Types \bullet Val_\sigma\}$$

Similarly we assume a unique set of variables Var_σ for each type σ, and define *Var* as the union of these sets.

5.2 Bindings, States and Predicates

A *binding* is a total function that maps every variable to a value of the correct type.

$$Bnd == \{b: Var \rightarrow Val \mid (\forall \sigma: Types \bullet (\forall V: Var_\sigma \bullet b(V) \in Val_\sigma))\}$$

Each binding corresponds to a single ground answer to a Prolog-like query. The mechanism for representing "unbound" variables is described below.

A *state* is a set of bindings:

$$State == \mathbb{P}\, Bnd$$

A state corresponds to our usual notion of a predicate with some free variables, which is true or false once provided with a binding for those variables, i.e., for a binding in the state. Given a predicate P, we write pred P to denote the set of bindings satisfying P. A completely unbound variable of type σ is represented by a (possibly infinite) state that has one binding to each element of Val_σ.

5.3 Term Evaluation

A term has a value relative to some binding. For a term T and binding b,

1. if T is a variable V, $\mathsf{eval}\ b\ T$ is simply the value for V in b, i.e., $b(V)$;
2. if T is a function application $F(X)$, $\mathsf{eval}\ b\ T$ is the evaluation of F applied to the evaluation of X, i.e., $(\mathsf{eval}\ b\ F)(\mathsf{eval}\ b\ X)$;
3. if T is a function $(\lambda X{:}\,\sigma \bullet e)$ where e is of type τ, $\mathsf{eval}\ b\ T$ is a function from elements of Val_σ to elements of Val_τ, defined as follows.

   $$(\lambda V{:}\, Val_\sigma \bullet eval\ (b \oplus \{X \mapsto V\})\ e)$$

   The binding $b \oplus \{X \mapsto V\}$ is the binding that maps all variables as b maps them, but with X mapped to V. In general the function $f \oplus g$, where f and g are functions, behaves as g for all elements in the domain of g, and as f for elements in the domain of f that are not in the domain of g.

6 Program Execution

In this section we model programs as functions from state to state. The notation $\{X{:}\,T \mid P \bullet E\}$ describes the set of values of the expression E, for each X of type T for which predicate P holds (when P is *true* we may omit it, i.e., $\{X{:}\,T \bullet E\}$). The notation $\{X{:}\,T \mid P\}$ filters the elements of T to leave only those that satisfy P.

6.1 Executions

We define the semantics of our language in terms of *executions*, which are mappings from initial states to final states. The mapping is partial because the program is only well-defined for those initial states that guarantee satisfaction of all the program's assumptions. Executions satisfy three healthiness properties, which restrict executions to model pure logic programs.

$$\begin{aligned} Exec == \{e{:}\, State \nrightarrow State \mid \\ \quad \operatorname{dom} e = \mathbb{P}\{b{:}\ Bnd \mid \{b\} \in \operatorname{dom} e\} \wedge & \quad (7) \\ \quad (\forall s{:}\operatorname{dom} e \bullet e(s) \subseteq s) \wedge & \quad (8) \\ \quad (\forall s{:}\operatorname{dom} e \bullet e(s) = \{b{:}\, s \mid e(\{b\}) = \{b\}\})\} & \quad (9) \end{aligned}$$

The notation '$\nrightarrow$' denotes a partial function. An execution, e, maps a state, s (a set of bindings or possible answers), to a new state, $e(s)$. Because execution of a command always constrains the set of possible answers, $e(s)$ must be a subset of s (property (8)). For a pure logic program, the new state can be constructed by considering whether each binding b in s is kept or not (property (9)); note that because of property (8), $e(\{b\})$ is either $\{b\}$ or $\{\}$. For a pure logic program, the domain of execution e can be determined by considering for each binding b whether or not $\{b\}$ is in the domain of e. The states in the domain of e are then all possible subsets of the set of all such bindings (property (7)). The healthiness properties are discussed in more detail in [5].

6.2 Semantic Function for Commands

We define the semantics of the commands in our language via a function that takes a command and returns the corresponding execution.

$$\mid \mathsf{exec}: \mathit{Cmd} \to \mathit{Exec}$$

The semantics of the basic commands (excluding recursion, which is treated in Section 8) is shown in Figure 3. In the remainder of this section, we explain the definitions. In [4], we show that all executions constructed using the definitions satisfy the healthiness properties of executions.

$$\begin{aligned}
&\mathsf{exec}(\langle P \rangle) = (\lambda\, s : \mathit{State} \bullet s \cap \overline{P}) \\
&\mathsf{exec}(\{A\}) = (\lambda\, s : \mathbb{P}\, \overline{A} \bullet s) \\
&\mathsf{exec}(c_1 \vee c_2) = \mathsf{exec}\, c_1 \cup \mathsf{exec}\, c_2 \\
&\mathsf{exec}(c_1 \wedge c_2) = \mathsf{exec}\, c_1 \cap \mathsf{exec}\, c_2 \\
&\mathsf{exec}(c_1 \,;\, c_2) = \mathsf{exec}\, c_1 \fatsemi \mathsf{exec}\, c_2 \\
&\mathsf{exec}(\exists\, V : \sigma \bullet c) = \mathsf{exists}_\sigma\, V (\mathsf{exec}\, c) \\
&\mathsf{exec}(\forall\, V : \sigma \bullet c) = \mathsf{forall}_\sigma\, V (\mathsf{exec}\, c)
\end{aligned}$$

Fig. 3. Execution semantics of basic commands

Specifications and Assumptions. The result of executing specification $\langle P \rangle$ consists of those bindings in s that satisfy P.

An assumption $\{A\}$ is defined for all states s such that A holds for all bindings in s; the result of executing assumption $\{A\}$ has no effect (the set of bindings remains unchanged).

Propositional Operators. Disjunction and parallel conjunction are defined as point-wise union and intersection of the corresponding executions. We present the definitions as Z axiomatic definitions [16]; the signatures are given above the line, and the definitions in the form of predicates are given below the line.

$$\_ \cap \_ : \mathit{Exec} \times \mathit{Exec} \to \mathit{Exec}$$
$$\_ \cup \_ : \mathit{Exec} \times \mathit{Exec} \to \mathit{Exec}$$
$$(e_1 \cap e_2) = (\lambda\, s : \mathrm{dom}\, e_1 \cap \mathrm{dom}\, e_2 \bullet (e_1\ s) \cap (e_2\ s))$$
$$(e_1 \cup e_2) = (\lambda\, s : \mathrm{dom}\, e_1 \cap \mathrm{dom}\, e_2 \bullet (e_1\ s) \cup (e_2\ s))$$

For a conjunction $(c_1 \wedge c_2)$, if a state s is mapped to s' by $\mathsf{exec}\, c_1$ and s is mapped to s'' by $\mathsf{exec}\, c_2$, then $\mathsf{exec}(c_1 \wedge c_2)$ maps s to $s' \cap s''$. Disjunction is similar, but gives the union of the resulting states instead of intersection.

Sequential conjunction (c_1 , c_2) is defined as function composition of the corresponding executions.

Quantifiers. For a type σ, variable V, and a state s, we define the state '$\mathsf{unbind}_\sigma\ V\ s$' as one whose bindings match those of s in every place except V, which is mapped to all values of type σ.

$$\mathsf{unbind}_\sigma : \mathit{Var}_\sigma \to \mathit{State} \to \mathit{State}$$
$$\mathsf{unbind}_\sigma\ V\ s = \{b : s;\ x : \mathit{Val}_\sigma \bullet b \oplus \{V \mapsto x\}\}$$

Execution of an existentially quantified command $(\exists\, V : \sigma \bullet c)$ from an initial state s is defined if executing c is defined in the state s', which is the same as s except that V is unbound. Since executions either keep or discard individual bindings (property (8)), the execution of $(\exists\, V : \sigma \bullet c)$ keeps a binding b if there exists some value x such that b, with V mapped to x, would be kept by the execution of c. A binding b is kept, therefore, if $e(\{b \oplus \{V \mapsto x\}\}) \neq \varnothing)$, where e is the execution of c. We thus make the following definition of the existential quantifier for executions.

$$\mathsf{exists}_\sigma : \mathit{Var}_\sigma \to \mathit{Exec} \to \mathit{Exec}$$
$$\mathsf{exists}_\sigma\ V\ e = (\lambda\, s : \mathit{State} \mid \mathsf{unbind}_\sigma\ V\ s \in \mathrm{dom}\, e \bullet \{b : s \mid (\exists\, x : \mathit{Val}_\sigma \bullet e(\{b \oplus \{V \mapsto x\}\}) \neq \varnothing)\})$$

Universal quantification behaves in a similar fashion, except that to retain a binding b, execution of e must retain $b \oplus \{V \mapsto x\}$ for all values x of type σ.

$$\mathsf{forall}\sigma : \mathit{Var}_\sigma \to \mathit{Exec} \to \mathit{Exec}$$
$$\mathsf{forall}\sigma\ V\ e = (\lambda\, s : \mathit{State} \mid \mathsf{unbind}_\sigma\ V\ s \in \mathrm{dom}\, e \bullet \{b : s \mid (\forall\, x : \mathit{Val}_\sigma \bullet e(\{b \oplus \{V \mapsto x\}\}) \neq \varnothing)\})$$

7 Refinement

An execution e_1 is refined by an execution e_2 if and only if e_2 is defined wherever e_1 is and they agree on their outputs whenever both are defined. This is the usual "definedness" order on partial functions, as used, for example, by Manna [7]: it is simply defined by the subset relation of functions viewed as sets of pairs, i.e.,

$$e_1 \sqsubseteq_{\mathit{Exec}} e_2 \Leftrightarrow e_1 \subseteq e_2$$

Thus, if (s_1, s_2) is in e_1, then it must also be in e_2. Since both e_1 and e_2 are functions, there can be no other state associated with initial state s_1. This ensures that the set of answers is preserved by refinement, when the assumptions associated with e_1 hold. For some state $s' \notin \mathrm{dom}\, e_1$, (s', s'') may be in e_2 for any s''; in this case, the assumptions for e_1 do not hold (in s'), and thus e_2 may choose any answer (as long as the properties for executions are maintained).

Refinement is a pre-order — a reflexive and transitive relation — because subset is a pre-order on sets. Refinement for commands is defined in terms of refinement of executions.

$$c_1 \sqsubseteq c_2 \Leftrightarrow \mathsf{exec}(c_1) \sqsubseteq_{Exec} \mathsf{exec}(c_2)$$

Refinement equivalence ($\sqsupseteq\sqsubseteq$) is defined for *Cmd* and *Exec* as refinement in both directions.

8 Recursion

In this section we discuss the semantics of recursion. The treatment is different to our earlier approaches, as we do not have a separate environment that maps procedure names to procedures, because this may now be represented as part of the state. In addition, the move to higher-order programs means that some programs we can construct are not continuous, and thus to construct the fix point of a recursive procedure we need to go past the first infinite ordinal.

Consider the following function which takes as its argument a procedure and returns a procedure.

$$Ctx \mathrel{\widehat{=}} (\lambda P{:}\,\sigma \to Cmd \bullet (\lambda X{:}\,\sigma \bullet \mathcal{C}(P)))$$

$\mathcal{C}(P)$ is some *Cmd* in our language involving calls the procedure P.

We define $\mathbf{abort}_\sigma$ to be the least defined procedure in our language, i.e., it always aborts for any input of type σ.

$$\mathbf{abort}_\sigma \mathrel{\widehat{=}} (\lambda X{:}\,\sigma \bullet \{false\})$$

Now, as all contexts definable in our language are montonic [4], the least fix-point $\mu\, Ctx$ of Ctx exists [1], and furthermore, there exists an ordinal γ such that

$$\mu\, .Ctx = Ctx^{\gamma}(\mathbf{abort}_\sigma)$$

Given the definition of Ctx above, we write $\mu\, .Ctx$ as

$$\mu P \bullet (\lambda X{:}\,\sigma \bullet \mathcal{C}(P))$$

9 Procedures and Equality

While a logic programming language can provide a primitive equality relation for terms composed from basic types, extending equality to procedures is problematic. Many versions of Prolog represent procedures as terms, but implement equality as syntactic equality on the terms. However two syntactically different procedures may be semantically

equivalent, and hence such an implementation does not reflect the desired semantics. Unfortunately determining whether two syntactically different procedures are semantically equivalent is an undecidable problem, and hence equality cannot be implemented on procedures.

One special case that avoids this problem is an equality of the form $P = proc$, where the variable P is unbound. P can be bound to $proc$ and no comparison of procedures is required. However, such an equality is still problematic in the context of the refinement calculus, because if $proc$ is refined by $proc'$, one would like to be able to replace $proc$ by $proc'$ in any context. However $proc'$ is not equivalent to $proc$, it is a refinement.

This issue also complicates the notion of monotonicity of the language when procedure equality is included. For instance, the program

$$P = proc \wedge P = proc \tag{10}$$

should refine to just $P = proc$. However, if we have $proc \sqsubseteq proc'$ and $proc \sqsubseteq proc''$ for some $proc'$ and $proc''$ which do not refine each other, the following is also a valid refinement.

$$P = proc' \wedge P = proc'' \tag{11}$$

The two bindings for P are both semantically and syntactically different, and thus the program should fail – yet a program that fails would not typically be regarded as a valid refinement of (10).

The problems with such a construct prompted us to disallow the use of procedure equality. The only mechanism we allow to bind a procedure value to a variable is via parameter passing. In this case the formal parameter variable is guaranteed to be unbound, and gets bound once at the time the procedure is called.

Note that a procedure passed as a parameter can be replaced by a refinement. For example, a call of the form $q(proc, X)$, where $proc$ is a procedure parameter, can be replaced by $q(proc', X)$ if $proc \sqsubseteq proc'$. This is guaranteed by the monotonicity under refinement of our language.

By disallowing procedure equality, we limit the programs we can describe. We cannot have a procedure that accepts an unbound procedure variable as an input and on completion binds the variable to some procedure. For example we cannot write a procedure that succeeds and binds P to $test$ if its other parameter satisfies $test$.

$$testBind \mathrel{\widehat{=}} (\lambda P{:}\sigma \to Cmd, X{:}\sigma \bullet \{X \in \sigma\}, test(X) \wedge P = test)$$

However, we may still achieve the same effect on non-procedure variables without the procedure equality command. For instance, for a program $\mathcal{C}(P, Y)$ containing calls to procedure variable P and references to some set of non-procedure variables Y (possibly containing X), a program

$$testBind(P, X), \mathcal{C}(P, Y)$$

may be rewritten as

$$test(X), \mathcal{C}(test, Y)$$

In both cases the bindings for the variables in Y will be the same, but the latter program does not constrain P in any way.

10 Conclusions

In this paper we have presented a semantics for refinement of higher-order logic programs. We have used Nadathur and Miller's semantics for higher-order logic programs [10] for the basis of our semantics. They describe an implementation of the semantics, involving hereditary Harrop formulas, in the logic programming language λProlog [9]. The calculus we describe here would be suitable for developing programs for higher-order languages such as λProlog and Mercury [15].

In general, higher-order programming is more developed in the functional programming community. Some examples of development of functional programs include a refinement calculus for nondeterministic expressions [17] and development of extended ML programs [13]. Lacey, Richardson, and Smaill [6] use higher-order techniques to synthesise both first- and higher-order logic programs. They develop an automatic synthesiser in λClam, which is a proof system written in λProlog. They adopt a proof-planning approach to the problem. The refinement calculus approach we take is similar to Morgan's approach for imperative programs [8], though he does not explicitly mention higher-order programming. Naumann [11] uses a predicate transformer semantics to give a semantics for a higher-order imperative programming language, including a procedure binding construct.

In this paper we have extended earlier work [5] by including a type system in our wide-spectrum language. The semantics of dealing with procedures has also been simplified, by eliminating the need for a mapping from procedure identifiers to procedures (an environment). We presented some examples of using higher-order features in refinement, and discussed some of the problems associated with a mechanism for binding a procedure to a variable in a logic program development framework. We have distinguished higher-order programming, where variables may take the value of procedures, from meta-programming, where variables may take the value of any command in our language, e.g., a procedure call rather than a procedure itself. Meta-programming by this definition is not dealt with in this paper – this is an avenue for future work.

References

1. B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
2. Oege de Moor and Ganesh Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269(1–2):135–162, 2001.
3. Y. Deville and K.-K. Lau. Logic program synthesis. *Journal of Logic Programming*, 19,20:321–350, 1994. Special Issue: Ten Years of Logic Programming.
4. I. Hayes, R. Nickson, P. Strooper, and R. Colvin. A declarative semantics for logic program refinement. Technical Report 00-30, Software Verification Research Centre, The University of Queensland, 2000.
5. I. J. Hayes, R. Colvin, D. Hemer, R. Nickson, and P. A. Strooper. A refinement calculus for logic programs. *Theory and Practice of Logic Programming*, 2(4–5):425–460, July–September 2002.
6. David Lacey, Julian Richardson, and Alan Smaill. Logic program synthesis in a higher-order setting. In John W. Lloyd et al., editor, *Computational Logic 2000*, volume 1861 of *LNAI*, pages 87–100. Springer-Verlag, 2000.

7. Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.
8. Carroll Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.
9. G. Nadathur and D. Miller. An overview of Lambda-PROLOG. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827. MIT Press, 1988.
10. G. Nadathur and D. Miller. Higher-order logic programming. In Dov M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logics for Artificial Intelligence and Logic Programming*, volume 5, chapter 8, pages 499–590. Clarendon Press, Oxford, 1998.
11. D. A. Naumann. Predicate transformer semantics of a higher order imperative language with record subtyping. *Science of Computer Programming*, 41(1):1–51, September 2001.
12. H. A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.
13. D. Sannella. Formal program development in Extended ML for the working programmer. In *Proc. 3rd BCS/FACS Workshop on Refinement*, Springer Workshops in Computing, pages 99–130. Springer, 1990.
14. S. Seres and M. Spivey. Higher-order transformation of logic programs. In K.-K. Lau, editor, *Proceedings of the Tenth International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR 2000)*, volume 2042 of *LNCS*, pages 57–68. Springer-Verlag, 2000.
15. Z. Somogyi, F.J. Henderson, and T.C. Conway. Mercury, an efficient purely declarative logic programming language. In R. Kotagiri, editor, *Proceedings of the Eighteenth Australasian Computer Science Conference*, pages 499–512, Glenelg, South Australia, 1995. Australian Computer Science Communications.
16. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
17. Nigel Ward. *A Refinement Calculus for Nondeterministic Expressions*. PhD thesis, Department of Computer Science, University of Queensland, 1994.