Improving the *Whelk* System: a type-theoretic reconstruction

Geraint A. Wiggins
Department of Artificial Intelligence
University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN
Scotland
geraint@ai.ed.ac.uk

April 30, 1998

Abstract

I present a reformulation of the *Whelk* system [Wiggins 92b], as a higher-order type theory. The theory is based on that of [Martin-Löf 79], adapted to facilitate the extraction of logic programs from proof objects. A notion of normalization is used to ensure that the extracted program is executable by standard logic-programming methods. The extension admits specifications over types and programs, and so allows modularity and the construction of program combinators. In doing so, it demonstrates that logic program synthesis techniques have potential for solving "industrial-strength" problems.

1 Introduction

In this paper, I present a reconstruction of the *Whelk* proof development and program construction system which was explained in [Wiggins *et al.* 91, Wiggins 92a, Wiggins 92b]. The reconstruction uses a higher order logic which admits quantification over types, and, following the style of Martin-Löf Type Theory [Martin-Löf 79], views sentences in the logic as types. Proofs of such sentences are then thought of as members of the types. The addition of the higher order features means we can now synthesise meta-programs and program modules. Making such a gain means that the system is more complicated than before, because type membership is undecidable. However, much of the extra proof obligation entailed may be dealt with automatically, and good heuristics exist for those parts which are not straightforward. The correctness argument for the new system is greatly simplified over the original *Whelk*, because it can be given largely by reference to Martin-Löf's existing theory.

The paper is structured as follows. In section 2, I remind the reader of the *modus operandi* of the original *Whelk* system, and illustrate the disadvantages which led to the reconstruction explained here; in section 3 I explain the shift from the first order logic to the type theory and outline the changes necessary. In section 4, I give those inference figures of the reconstructed system necessary for an example. Correctness is shown by reference to Martin-Löf original calculus, but there is not space to cover it in depth here. Section 5 shows the synthesis of the `subset/2` predicate for comparison with the original system [Wiggins 92b]. Finally, because of space constraints, I merely sketch the normalization process for generating the runnable programs in section 6.

2 Background

The *Whelk* system is an attempt to bring ideas used in the domain of type theory and functional program synthesis to bear on the synthesis and/or transformation of logic programs. We wish to give the specification of a program in terms only of the logical relation between its arguments; the construction of a suitable algorithm is then to be left to a process of proof that the specification can be realised. The notion on which this is based is the *proofs-as-programs* paradigm of [Constable 82], in which a constructive proof of an existentially quantified conjecture is encoded as an *extract term*, a function, expressed as a λ-term, embodying an algorithm which implements the input/output relation specified in the conjecture. The proofs-as-programs idea has been adapted [Bundy *et al.* 90b] to synthesise logic programs. This is achieved by viewing logic programs as functions on to the Boolean type. The synthesised relations are then Boolean-valued functions (expressed as logic programs, called in the all-ground mode).

In [Wiggins *et al.* 91, Wiggins 92b], the original version of the logic system designed for this purpose is explained. While the system does its job effectively, and is in use for teaching and research at various institutions, the background theory is not as tidy as it might be. In particular, the means of specifying what constitutes a *synthesis conjecture* leads us to convoluted correctness proofs, which are less than elegant. Further, the original system is restricted to first order specifications, with no meta-language. Therefore, it is not possible to synthesise program modules, or to specify meta-programs, simply because quantification over mathematical structures other than the natural numbers and parametric lists is not allowed. (In fact, it is not clear that the modularity issue is as important in the context of program synthesis as elsewhere; however, the fact remains that the expressive power of modularity is desirable.)

In *Whelk*, the fact that we wish to synthesise a program is expressed by means of a *decidability* operator, ∂, which is applied to the specification of the desired program. By proving constructively that a specification is decidable, we synthesise the decision procedure – in the form of a logic program – for that specification. In earlier versions of the system [Wiggins 92b], ∂ was defined by reference to the existence of the extracted program, and not *via* sequent calculus rules, like the other logical connectives. Therefore, it behaved rather like a macro, defined at the meta-level with respect to the sequent calculus. This approach was taken because it allowed certain restrictions to be placed on the form of the programs derived from *Whelk* proofs, so that no program components would be generated by parts of the proof concerned purely with verification of correctness. In particular, the meta-flavour of the ∂ definition allowed certain restrictions on the applicability of inference rules to be hidden in a way which seems, in retrospect, undesirable. In the reconstruction explained here, these meta-level definitions and restrictions have been abandoned in favour of logical elegance, their respective effects being reproduced by a more conventional sequent calculus definition and application of a simple heuristic during proof construction.

In the following sections of this paper, I will discuss the use of the reconstructed calculus, which I will call WTT, for program manipulation, with reference to the existing example from [Wiggins 92b], showing how the program can be modularised by type. The *Whelk* conjecture, which clearly has types built in, is as follows:

$$\vdash \forall k{:}list(\mathbb{N}).\forall l{:}list(\mathbb{N}).\partial\,(\forall x{:}\mathbb{N}.x \in k \to x \in l)$$

where $\in$ is the usual list membership; it specifies the list inclusion relation. Proof of this synthesis conjecture is fairly straightforward, and will be outlined again in section 5. The extracted program, expressed as a logical equivalence, may be automatically converted the Gödel code, as shown in [Wiggins 92b].

The new higher-order calculus presented here is also designed for describing and manipulating logic programs. However, because of the first order nature of logic programs themselves, it is clear that programs parameterised by module or type will not be expressible as such, in a strictly first-order way. In what follows, therefore, I assume that the parameters of the modules I synthesise will be instantiated *within* WTT, and the resulting Gödel program extracted subsequently, rather than the converse.

3 Reconstruction

There are two main aspects to the type-theoretic reconstruction of *Whelk*. First, the decidability operator, ∂, on which the construction of logic programs rests, has been integrated into the sequent calculus as an operator in its own right, rather than as a set of operators related to the standard ones, as is the case in *Whelk*. This results in a considerable reduction in the number of rules (roughly half), which necessarily makes things less complicated.

The second change is the raising of the calculus to higher order. This is carried out by following the rules of Martin-Löf's Type Theory [Martin-Löf 79], and adapting and extending them where necessary. This approach has the advantage that correctness can be argued in terms of the original system. The introduction of higher-order structures raises the possibility that functional terms and variables might appear in the extracted programs, and we will need to be careful that this does not prove to be the case – the programs would not then be logic programs. This point is the main motivation for the building of a new calculus, from the bottom up, as it were, rather than simply implementing the *WTT* logic on top of an existing type theory. In the latter of these options, it is not clear how (or indeed if) we could guarantee the first order structures we need.

4 *Whelk* Type Theory

4.1 Introduction

In order to specify our type theory, we must specify what is the syntactic form of each type, and what is the syntactic form of objects of each type. In general, there will be two forms of objects: the *non-canonical* and *canonical* forms – *i.e.* forms which may be further evaluated and forms which may not. The calculus must also supply a means of computation to enable the derivation of canonical forms from non-canonical ones. It will be clear from the expression of the evaluation strategy below, which follows Martin-Löf's description closely, that the theory is extensional – that is to say, equality of functions is expressed in terms of equality of corresponding inputs and outputs.

Like Martin-Löf's type theory, *WTT* admits four kinds of "judgement" – a judgement being essentially what we can prove is true (*i.e.,* the logical sentence to the right of the sequent symbol). The four kinds of judgement I will use here relate to the formation of ***types*** and the ***objects*** which inhabit them. In each case, entities themselves, and equality between entities is covered.

Judgement	**Form**
A is a type	A *type*
A and B are equal types	$A = B$
a is an object of type A	$a \in A$
a and b are equal objects of type A	$a = b \in A$

4.2 Preamble

(First order) *Whelk* uses distinct languages for expression of its specification ($\mathcal{L}_{\mathcal{S}}$) and of its programs ($\mathcal{L}_{\mathcal{P}}$). A mapping connects the two, and allows us to say what the synthesised program means in terms of the specification – for more details, see [Wiggins 92b]. In *WTT*, a similar arrangement pertains: loosely, types correspond with $\mathcal{L}_{\mathcal{S}}$-formulæ and objects with $\mathcal{L}_{\mathcal{P}}$-formulæ, and type membership replaces the interpretation. However, there is a significant difference: in *WTT*, we have a type of types, known as a *universe*, so it is possible to view a type as an object also. For the purposes of the examples here, however, these higher-order complications are irrelevant. (Incidentally, there is a hierarchy of universes, each containing the one "below" it, which avoids Russell's Paradox, while still giving all the types we need.)

The type language admits the familiar connectives (though for reasons of limited space I have given only those necessary for the example): $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies), $\forall$ (for all). Contradiction is denoted by $\{\}$ (the empty type), and negation by implication, so the negation of A is $A \rightarrow \{\}$. We also have identity within a type, $\equiv_T$, (NB, this is not the same as the object equality judgement) and the decidability operator, ∂. The quantifiers are typed in the usual way, : denoting type membership. The operators of the object language are the constructors of the members of the types. They are best given in terms of the types they construct and destruct. To facilitate comparison, these constructions are given in the style of [Martin-Löf 79].

Type	**Canonical Form**	**Non-canonical Form**
$\forall x\!:\!A.B$	$\amalg x\!:\!A.b$	$c(a)$
$A \rightarrow B$	$x\!:\!A \mapsto b$	$c\{a\}$
$A \wedge B$	$a \sqcap b$	$\mathcal{L}_\wedge(\dashv), \mathcal{R}_\wedge(\lfloor)$
$A \vee B$	$\mathcal{L}_\sqcup(\dashv), \mathcal{R}_\sqcup(\lfloor)$	$\mathcal{D}_\vee(\S, \dagger)(\rfloor, \lceil, \rceil)$
$\partial\, A$	$\mathcal{L}_\partial(\dashv), \mathcal{R}_\partial(\dashv)$	$\mathcal{D}_\partial\,(\S, \dagger)(\rfloor, \lceil, \rceil)$
$a \equiv_A b$	$c \overset{A}{\triangleq} d$	$\mathcal{I}(\rfloor \overset{A}{\triangleq} \lceil, \rceil)$
$\{\}$		$false$
$A\ list$	$[\,]_A, [a \mid b]_A$	$p(x); p(x\!:\!A\ list) = List\mathcal{C}(\S, \lceil, \S_\prime, \S_\infty, \sqsubseteq, \rceil)$

The non-canonical forms operate as follows:

1. If c has canonical form $\amalg x\!:\!A.b$ and $a \in A$ then $c(a)$ is $b[a/x]$.
2. If c has canonical form $x\!:\!A \mapsto b$ and $a \in A$ then $c\{a\}$ is b.
3. If c has canonical form $a \sqcap b$ then $\mathcal{L}_\wedge(\rfloor)$ is a and $\mathcal{R}_\wedge(\rfloor)$ is b.
4. If c has canonical form $\mathcal{L}_\vee(\dashv)$ then $\mathcal{D}_\vee(\S, \dagger)(\rfloor, \lceil, \rceil)$ is $d[a/x]$; if c has canonical form $\mathcal{R}_\vee(\lfloor)$ then $\mathcal{D}_\vee(\S, \dagger)(\rfloor, \lceil, \rceil)$ is $e[b/y]$.
5. If b has canonical form $\mathcal{L}_\partial(\dashv)$ then $\mathcal{D}_\partial\,(\S, \dagger)(\lfloor, \rfloor, \lceil)$ is $c[a/x]$; if b has canonical form $\mathcal{R}_\partial(\dashv)$ then $\mathcal{D}_\partial\,(\S, \dagger)(\lfloor, \rfloor, \lceil)$ is $d[a/y]$.
6. $a \equiv_A b$ expresses the identity relation between a and b, and is syntactically distinct from the equality judgement $a = b \in A$. If c has canonical form $p \overset{A}{\triangleq} q$ then $\mathcal{J}(\rfloor, \lceil)$ is $d[q/p]$.
7. $\{\}$ is uninhabited; it has no constructors.
8. If x is $[\,]_A$ then $List\mathcal{C}(\S, \lceil, \S_\prime, \S_\infty, \sqsubseteq, \rceil)$ is d;
   if x is $[x_0 \mid x_1]$ then $List\mathcal{C}(\S, \lceil, \S_\prime, \S_\infty, \sqsubseteq, \rceil)$ is $e[p(x_1)/v]$.

The appearance of the non-canonical forms in synthesised programs corresponds with the application of elimination rules in the proof. Similarly, the application of introduction rules in the proof corresponds with the appearance of object constructors in the program. This will become obvious on inspection of the example inference figures, below.

The sequent symbol is written $\vdash$. Non-empty formulæ are represented by upper case Roman letters (A,B,C,D); variables by lower case (x,y,z); types by lower case Greek letters (τ); and sequences of formulæ by upper case. Hypothesis management is implicit, so the order of the hypotheses is insignificant; unchanging hypotheses are not shown in the figures below. As the calculus is constructive, there is only one formula on the right of $\vdash$, so no management is needed there. Because this is a sequent calculus, the introduction and elimination rules of natural deduction correspond with operations on the right and left of the sequent symbol, respectively.

The construction proving each formula is associated with it by $\in$, which may be thought of as type membership. Thus, a proof/program may be viewed as inhabiting the type which is its specification. Note that program constructions in the inference figures, in particular those associated with hypotheses, may contain uninstantiated (meta-)variables; the construction process may be seen as a process of instantiation. Some cases (*e.g.* proofs of implications not in the scope of ∂) will lead to finished constructions containing uninstantiated variables. These constructions may be thought of as program transformers: they take an "input", by unification, and return a new structure related to that input in a way described by the conjecture whose proof produced the transformer. These ideas will be explored elsewhere.

In order to verify or synthesise a WTT program, we prove a conjecture of the general form

$$\Lambda \vdash \phi \in \forall \overline{a\!:\!\vec{\tau}}.\partial\, S(\vec{a})$$

where Λ is an initial theory providing the definitions and axioms necessary for the proof; S is the specification of the program; $\overline{a\!:\!\vec{\tau}}$ is a vector of typed arguments; and ϕ is the (synthesised) program, which includes an initial goal. The difference between verification and synthesis is the instantiation level of the object ϕ – if it is fully ground, we are verifying, if it is a pure variable we are synthesising, and correspondingly for any case in between. This is essentially the same as *Whelk*. However, given the improved uniformity of the reconstruction, we can now say precisely what an object/program means for any type membership judgement which is part of a proof. If we have a complete proof of the conjecture

$$\Lambda \vdash \phi \in \Phi$$

so that ϕ is ground, then we know that ϕ explicitly encodes a proof of Φ. The computation rules of the calculus allow us to manipulate this proof – instantiating arguments, for example – in a way which models the execution of the program in a higher-order functional programming language. This is relevant to logic programs because, when executed in the all-ground mode, they are equivalent to boolean functions. We can therefore read a subset of the proof objects produced by the system – that subset whose member are members of ∂ types – as logic programs.

4.3 General Rules

Reflexivity

$$\frac{\vdash a \in A}{\vdash a = a \in A} \qquad \frac{\vdash A \; type}{\vdash A = A}$$

Symmetry

$$\frac{\vdash a = b \in A}{\vdash b = a \in A} \qquad \frac{\vdash B = A}{\vdash A = B}$$

Transitivity

$$\frac{\vdash a = b \in A \quad \vdash b = c \in A}{\vdash a = c \in A} \qquad \frac{\vdash A = B \quad \vdash B = C}{\vdash A = C}$$

Equality of Types

$$\frac{\vdash a \in A \quad \vdash A = B}{\vdash a \in B} \qquad \frac{\vdash a = b \in A \quad \vdash A = B}{\vdash a = b \in B}$$

Substitution

$$\frac{\vdash a \in A \quad x{:}A \vdash B[a/x] \; type}{\vdash B \; type} \qquad \frac{\vdash a \in A \quad x{:}A \vdash b[a/x] \in B[a/x]}{\vdash b \in B}$$

$$\frac{\vdash a = c \in A \quad x{:}A \vdash B[a/x] = D[c/x]}{\vdash B = D}$$

$$\frac{\vdash a = c \in A \quad x{:}A \vdash b[a/x] = d[c/x] \in B[a/x]}{\vdash b = d \in B}$$

Axiom

$$\frac{}{a{:}A \vdash a \in A}$$

4.4 Dependent Function Type

Formation

$$\frac{\vdash A \; type \quad x{:}A \vdash B \; type}{\vdash \forall x{:}A.B \; type} \qquad \frac{\vdash A = C \quad x{:}A \vdash B = D}{\vdash (\forall x{:}A.B) = (\forall x{:}C.D)}$$

Construction

$$\frac{x\!:\!A \vdash b \in B}{\vdash (\amalg x\!:\!A.b) \in (\forall x\!:\!A.B)} \qquad \frac{x\!:\!A \vdash b = c \in B}{\vdash (\amalg x\!:\!A.b) = (\amalg x\!:\!A.c) \in (\forall x\!:\!A.B)}$$

Selection

$$\frac{\vdash t \in B \quad b\!:\!B[t/x] \vdash g \in G}{c\!:\!(\forall x\!:\!A.B) \vdash g[c(t)/b] \in G}$$

$$\frac{\vdash c \in (\forall x\!:\!A.B) \quad \vdash t \in B \quad b\!:\!B[t/x] \vdash g \in G}{\vdash g[c(t)/b] \in G}$$

Evaluation

$$\frac{\vdash a \in A \quad x\!:\!A \vdash b \in B}{\vdash ((\amalg x\!:\!A.b)(a)) = b[a/x] \in B[a/x]} \qquad \frac{\vdash c \in (\forall x\!:\!A.B)}{\vdash ((\amalg x\!:\!A.c)(x)) = c \in (\forall x\!:\!A.B)}$$

4.5 Function Type

Formation

$$\frac{\vdash A \textit{ type} \quad \vdash B \textit{ type}}{\vdash A \to B \textit{ type}} \qquad \frac{\vdash A = C \quad \vdash B = D}{\vdash (A \to B) = (C \to D)}$$

Construction

$$\frac{a\!:\!A \vdash b \in B}{\vdash (x\!:\!A \mapsto b) \in (A \to B)} \qquad \frac{x\!:\!A \vdash b = d \in B}{\vdash (x\!:\!A \mapsto b) = (x\!:\!A \mapsto d) \in (A \to B)}$$

Selection

$$\frac{\vdash a\!:\!A \quad b\!:\!B \vdash g \in G}{c\!:\!(A \to B) \vdash g[c\{a\}/b] \in G} \qquad \frac{\vdash c \in (A \to B) \quad \vdash a \in A \quad b\!:\!B \vdash g \in G}{\vdash g[c\{a\}/b] \in G}$$

Evaluation

$$\frac{\vdash a \in A \quad \vdash b \in B}{\vdash ((x\!:\!A \mapsto b)\{a\}) = b \in B} \qquad \frac{\vdash b \in (x\!:\!A \mapsto B)}{\vdash ((x\!:\!A \mapsto b)\{x\}) = b \in (x\!:\!A \mapsto B)}$$

4.6 Product Type

Formation

$$\frac{\vdash A \textit{ type} \quad \vdash B \textit{ type}}{\vdash A \wedge B \textit{ type}} \qquad \frac{\vdash A = C \quad B = D}{\vdash (A \wedge B) = (C \wedge D)}$$

Construction

$$\frac{\vdash a \in A \quad \vdash b \in B}{\vdash a \sqcap b \in A \wedge B} \qquad \frac{\vdash a = c \in A \quad \vdash b = d \in B}{\vdash (a \sqcap b) = (c \sqcap d) \in (A \wedge B)}$$

Selection

$$\frac{a\!:\!A, b\!:\!B \vdash g \in G}{c\!:\!(A \wedge B) \vdash g[a, b/\mathcal{L}_{\wedge}(\rfloor), \mathcal{R}_{\wedge}(\rfloor)] \in \mathcal{G}} \qquad \frac{\vdash c\!:\!(A \wedge B) \quad a\!:\!A, b\!:\!B \vdash g \in G}{\vdash g[a, b/\mathcal{L}_{\wedge}(\rfloor), \mathcal{R}_{\wedge}(\rfloor)] \in \mathcal{G}}$$

Evaluation

$$\frac{\vdash a \sqcap b \in A \wedge B}{\vdash \mathcal{L}_{\wedge}(\dashv \sqcap \lfloor) = \dashv \in \mathcal{A}} \qquad \frac{\vdash a \sqcap b \in A \wedge B}{\vdash \mathcal{R}_{\wedge}(\dashv \sqcap \lfloor) = \lfloor \in \mathcal{B}}$$

4.7 Disjoint Union Type

Formation

$$\frac{\vdash A\ \mathit{type} \quad \vdash B\ \mathit{type}}{\vdash A \vee B\ \mathit{type}} \qquad\qquad \frac{\vdash A = C \quad \vdash B = D}{\vdash (A \vee B) = (C \vee D)}$$

Construction

$$\frac{\vdash a \in A}{\vdash \mathcal{L}_{\vee}(\dashv) \in \mathcal{A} \vee \mathcal{B}} \qquad\qquad \frac{\vdash a = c \in A}{\vdash \mathcal{L}_{\vee}(\dashv) = \mathcal{L}_{\vee}(\rfloor) \in (\mathcal{A} \vee \mathcal{B})}$$

$$\frac{\vdash b \in B}{\vdash \mathcal{R}_{\vee}(\lfloor) \in \mathcal{A} \vee \mathcal{B}} \qquad\qquad \frac{\vdash b = d \in B}{\vdash \mathcal{R}_{\vee}(\lfloor) = \mathcal{R}_{\vee}(\lceil) \in (\mathcal{A} \vee \mathcal{B})}$$

Selection

$$\frac{x{:}A \vdash d{:}G \quad y{:}B \vdash e{:}G}{c{:}(A{\vee}B) \vdash \mathcal{D}_{\vee}(\S,\dagger)(\rfloor,\lceil,\rceil) \in \mathcal{G}} \qquad \frac{\vdash c{\in}(A{\vee}B) \quad x{:}A{\vdash}d{\in}G \quad y{:}B{\vdash}e{\in}G}{\vdash \mathcal{D}_{\vee}(\S,\dagger)(\rfloor,\lceil,\rceil){\in}\mathcal{G}}$$

Evaluation

$$\frac{\vdash a \in A \quad x{:}A \vdash d \in C \quad y{:}B \vdash e \in C}{\vdash \mathcal{D}_{\vee}(\S,\dagger)(\mathcal{L}_{\vee}(\dashv),\lceil,\rceil) = \lceil[\dashv/\S] \in \mathcal{C}}$$

$$\frac{\vdash a \in A \quad x{:}A \vdash d \in C \quad y{:}B \vdash e \in C}{\vdash \mathcal{D}_{\vee}(\S,\dagger)(\mathcal{R}_{\vee}(\lfloor),\lceil,\rceil) = \rceil[\lfloor/\dagger] \in \mathcal{C}}$$

4.8 Decision Type

Formation

$$\frac{\vdash A\ \mathit{type}}{\vdash \partial\, A\ \mathit{type}} \qquad\qquad \frac{\vdash A = B}{\vdash (\partial\, A) = (\partial\, B)}$$

Construction

$$\frac{\vdash a \in A}{\vdash \mathcal{L}_{\partial}(\dashv) \in \partial\,\mathcal{A}} \qquad\qquad \frac{\vdash a = b \in A}{\vdash \mathcal{L}_{\partial}(\dashv) = \mathcal{L}_{\partial}(\lfloor) \in \partial\,\mathcal{A}}$$

$$\frac{\vdash a \in (A{\to}\{\})}{\vdash \mathcal{R}_{\partial}(\dashv) \in \partial\,\mathcal{A}} \qquad\qquad \frac{\vdash a = b \in (A{\to}\{\})}{\vdash \mathcal{R}_{\partial}(\dashv) = \mathcal{R}_{\partial}(\lfloor) \in \partial\,\mathcal{A}}$$

Selection

$$\frac{x{:}A{\vdash}d{\in}G \quad y{:}(A{\to}\{\}){\vdash}e{\in}G}{a{:}\partial\,A \vdash \mathcal{D}_{\partial}\,(\S,\dagger)(\dashv,\lceil,\rceil){\in}\mathcal{G}} \qquad \frac{\vdash a{\in}\partial\,A \quad x{:}A{\vdash}d{\in}G \quad y{:}(A{\to}\{\}){\vdash}e{\in}G}{\vdash \mathcal{D}_{\partial}\,(\S,\dagger)(\dashv,\lceil,\rceil){\in}\mathcal{G}}$$

Evaluation

$$\frac{\vdash a \in A \quad x{:}A \vdash d \in C \quad y{:}(A{\to}\{\}) \vdash e \in C}{\vdash (\mathcal{D}_{\partial}\,(\S,\dagger)(\mathcal{L}_{\partial}(\dashv),\lceil,\rceil)) = \lceil[\dashv/\S] \in \mathcal{C}}$$

$$\frac{\vdash a \in A \quad x{:}A \vdash d \in C \quad y{:}(A{\to}\{\}) \vdash e \in C}{\vdash (\mathcal{D}_{\partial}\,(\S,\dagger)(\mathcal{R}_{\partial}(\dashv),\lceil,\rceil)) = \rceil[\dashv/\dagger] \in \mathcal{C}}$$

4.9 Identity Relation

Formation

$$\frac{\vdash A\ type \quad \vdash a\!:\!A \quad \vdash b\!:\!A}{\vdash a \equiv_A b\ type} \qquad \frac{\vdash A = C \quad \vdash a = c\!:\!A \quad \vdash b = d\!:\!A}{\vdash (a \equiv_A b) = (c \equiv_C d)}$$

Construction

$$\frac{\vdash a = b \in A}{\vdash (c \overset{A}{=} d) \in (a \equiv_A b)} \qquad \frac{\vdash a = b \in A}{\vdash (c \overset{A}{=} d) = (c \overset{A}{=} d) \in (a \equiv_A b)}$$

Selection

$$\frac{\vdash e \in G[b/a]}{c\!:\!(a \equiv_A b) \vdash \mathcal{I}(\rfloor, \rceil) \in \mathcal{G}} \qquad \frac{\vdash c \in (a \equiv_A b) \quad \vdash e \in G[a/b]}{\vdash \mathcal{I}(\rfloor, \rceil) \in \mathcal{G}}$$

Evaluation

$$\frac{\vdash a = b \in A \quad \vdash e \in C[a/b]}{\vdash \mathcal{J}(\dashv \overset{A}{=} \lfloor, \rceil) = \rceil \in \mathcal{C}}$$

4.10 Void Type

Formation

$$\frac{}{\vdash \{\}\ type} \qquad \frac{}{\vdash \{\} = \{\}}$$

Selection

$$\frac{}{a\!:\!\{\} \vdash false \in G}$$

4.11 Parametric Lists

Formation

$$\frac{\vdash A\ type}{\vdash A\ list\ type} \qquad \frac{\vdash A\ type}{\vdash (A\ list) = (A\ list)}$$

Construction

$$\frac{\vdash A\ type}{\vdash [\,]_A\!:\!A\ list} \qquad \frac{\vdash A\ type}{\vdash [\,]_A = [\,]_A \in A\ list}$$

$$\frac{\vdash a\!:\!A \quad \vdash b \in A\ list}{\vdash [a|b]_A \in A\ list} \qquad \frac{\vdash a = c \in A \quad \vdash b = d \in A\ list}{\vdash [a|b]_A = [c|d]_A \in A\ list}$$

Selection

$$\frac{\vdash d\!:\!G[[\,]_A/x] \quad x_0\!:\!A, x_1\!:\!A\ list, y\!:\!G[x_1/x] \vdash e \in G[[x_0|x_1])/x]}{x\!:\!A\ list \vdash (p(x); p(x\!:\!A\ list) =\ list\mathcal{C}(\S, \lceil, \S\prime, \S_\infty, \sqsubseteq, \rceil)) \in \mathcal{G}}$$

$$\frac{\vdash x \in A\ list \quad \vdash d\!:\!G[[\,]_A/x] \quad x_0\!:\!A, x_1\!:\!A\ list, y\!:\!G[x_1/x] \vdash e \in G[[x_0|x_1])/x]}{\vdash (p(x); p(x\!:\!A\ list) =\ list\mathcal{C}(\S, \lceil, \S\prime, \S_\infty, \sqsubseteq, \rceil)) \in \mathcal{G}}$$

Evaluation

$$\frac{\vdash d \in G[[\,]_A/x] \quad x_0\!:\!A, x_1\!:\!A\ list, y\!:\!G[x_1/x] \vdash e \in G[[x_0|x_1]/x]}{x\!:\!A\ list \vdash (p(x); p(x\!:\!A\ list) =\ list\mathcal{C}([\,]_A, \lceil, \S\prime, \S_\infty, \sqsubseteq, \rceil) = \lceil) \in \mathcal{G}}$$

$$\frac{\vdash d \in G[[\,]_A/x] \quad x_0\!:\!A, x_1\!:\!A\ list, y\!:\!G[x_1/x] \vdash e \in G[[x_0|x_1]/x]}{\begin{array}{l} x\!:\!A\ list \vdash (p(x); p(x\!:\!A\ list) =\ list\mathcal{C}([\ddagger\prime|\ddagger_\infty], \lceil, \S\prime, \S_\infty, \sqsubseteq, \rceil)) = \\ \hfill e[z_0, z_1, p(x_1)/x_0, x_1, v] \in G \end{array}}$$

4.12 Universes

Formation

$$\frac{}{\vdash U_n \ \mathit{type}} \qquad\qquad \frac{}{\vdash U_n = U_n}$$

Construction (similarly for all types except U_m)

$$\frac{\vdash A{:}U_n \quad x{:}A \vdash B{:}U_n}{\vdash (\forall x{:}A.B){:}U_n} \qquad\qquad \frac{\vdash A =\!\!:U_n \quad x{:}A \vdash B = D{:}U_n}{\vdash (\forall x{:}A.B) = (\forall x{:}C.D){:}U_n}$$

Selection

$$\frac{\vdash A \ \mathit{type}}{\vdash A{:}U_n} \qquad\qquad \frac{\vdash A = B}{\vdash A = B{:}U_n}$$

$$\frac{\vdash A{:}U_n}{\vdash A{:}U_{n+1}} \qquad\qquad \frac{\vdash A = B{:}U_n}{\vdash A = B{:}U_{n+1}}$$

5 Example: `subset/2`

For this example, I use the conjecture which specifies the `subset/2` predicate using lists as a representation for sets – that is, the predicate which succeeds when all the members of the list given as its first argument are members of that given as its second. The specification is parameterised by base type (*i.e.,* by the type of the list elements), and, necessarily, by a decision procedure for equality for that type. In the event that such a decision procedure is not supplied, the proof process will not yield a logic program. I assume a background theory defining the $\in_\tau$ (typed `member/2`), as follows:

$$\begin{array}{l}
\amalg\tau{:}U_0.\ \amalg d{:}(\forall a{:}\tau.\forall b{:}\tau.\partial\,(a \equiv_\tau b)).\\
\qquad \amalg x{:}\tau.\ \amalg y{:}\tau\ \mathit{list}.(m(y);\\
\qquad\qquad m(y{:}\tau\ \mathit{list}) =\\
\qquad\qquad\quad \mathit{list}\mathcal{C}(\dagger,\\
\qquad\qquad\qquad \mathcal{R}_\partial(\sqsubseteq{:}\{\} \mapsto \{\dashv\updownarrow\int\rceil),\\
\qquad\qquad\qquad y_0, y_1, v,\\
\qquad\qquad\qquad \mathcal{D}_\partial\,(\sqrt{\prime}, \sqrt{\infty})(\lceil(\tau)(\S)(\dagger_\prime), \mathcal{L}_\vee(\mathcal{L}_\partial(\sqrt{\prime})), \mathcal{R}_\vee(\sqsubseteq))))\\
\qquad\qquad\qquad\qquad\qquad\qquad \in \forall\tau{:}U_0.\forall d{:}(\forall a{:}\tau.\forall b{:}\tau.\partial\,(a \equiv_\tau b)).\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \forall x{:}\tau.\forall y{:}\tau\ \mathit{list}.\partial\,(x \in_\tau y)
\end{array} \qquad (1)$$

This definition is the statement that a program implementing `member` inhabits the appropriate type. It is worth mentioning at this stage that a user of WTT will not normally have to deal with such complex constructions. Instead, we also have two logical equivalences (*i.e.,* type equalities):

$$\begin{array}{rcll}
\forall\tau{:}U_0.\forall x{:}\tau.(x \in_\tau [\,]_\tau) & = & \{\} & (2)\\
\forall\tau{:}U_0.\forall x{:}\tau.\forall y_0{:}\tau.\forall y_0{:}\tau\ \mathit{list}.(x \in_\tau [y_0\,|\,y_1]) & = & (x \equiv_\tau y_0 \vee x \in_\tau y_1) & (3)
\end{array}$$

It is, however, necessary to show that the type defined using $\in_\tau$ is inhabited – so we must construct the object; in fact, this turns out to be done *via* a simple proof.

Returning to the example conjecture, we start with

$$\begin{array}{ll}
\vdash & \forall\tau{:}U_0.\forall d{:}(\forall a{:}\tau.\forall b{:}\tau.\partial\,(a \equiv_\tau b)).\\
& \qquad\qquad \forall x{:}\tau\ \mathit{list}.\forall y{:}\tau\ \mathit{list}.\partial\,(\forall z{:}\tau.z \in_\tau x \rightarrow z \in_\tau y)
\end{array} \qquad (4)$$

The proof proceeds by primitive induction on lists, first on x and then on y. Note that the proof is presented in refinement style, with the rules applied "backwards", and that I have omitted unchanging hypotheses unless they are used in the current proof step. Further, since we begin with a completely uninstantiated proof object, I have omitted it and the $\in$ symbol from the sequents – otherwise the sequents would be completely illegible.

The first move is to introduce the universally quantified type parameters, and the quantifier of x in 4. This yields the conjecture

$$\begin{array}{rcl} \tau : U_0, d : (\forall a : \tau . \forall b : \tau . \partial\, (a \equiv_\tau b)), & & \\ x : \tau \; list & \vdash & \forall y : \tau \; list . \partial\, (\forall z : \tau . z \in_\tau x \to z \in_\tau y) \end{array} \tag{5}$$

and the program

$$\sqcup \tau : U_0 . \sqcup d : (\sqcup a : \tau . \sqcup b : \tau . \partial\, (a \equiv_\tau b)) . \sqcup x : \tau \; list . \phi_5$$

where ϕ_5 is a meta-variable which will be instantiated during the rest of the proof. Note the difference from the original *Whelk* here: the $\sqcup$s were previously represented by a predicate definition head and arguments. Also, we have two (syntactically detectable) higher-order arguments, whose presence indicates that what we are synthesising here is not a logic program, but a logic-program-valued function - *i.e.,* a module. These, then, are the first two significant differences between this presentation of the example and that of [Wiggins 92b].

Next, we apply *list* selection to x. This gives us two subconjectures:

$$\vdash \; \forall y : \tau \; list . \partial\, (\forall z : \tau . z \in_\tau [\,]_\tau \to z \in_\tau y) \tag{6}$$

$$\begin{array}{rcl} x_0 : \tau, x_1 : \tau \; list, & & \\ v : (\forall y : \tau . z \in_\tau x_1 \to z \in_\tau y) & \vdash & \forall y : \tau \; list . \partial\, (\forall z : \tau . z \in_\tau [x_0 \,|\, x_1] \to z \in_\tau y) \end{array} \tag{7}$$

and the following synthesised program, where ψ_6 and ψ_7 are the program constructions corresponding with (6) and (7), respectively:

$$\begin{array}{l} \sqcup \tau : U_0 . \sqcup d : (\sqcup a : \tau . \sqcup b : \tau . \partial\, (a \equiv_\tau b)) . \\ \quad \sqcup x : \tau \; list . (p(x) ; p(x : \tau \; list) = \; listC(\S, \psi_{/} \S_{/}, \S_\infty, \sqsubseteq, \psi_{\imath})) \end{array}$$

Here is another difference between *Whelk* and *WTT*: the choice between [] and non-[] lists is built into the language as a type-selector, so no explicit disjunction appears here.

To prove the base case, (6), observe that the expression within the scope of ∂ is true, because the antecedent of the implication is always false. We can now use the new decision type rules to introduce the ∂ and then show that the resulting statment is indeed true. We introduce y and then use the first ∂ construction, to yield the following conjecture:

$$y : \tau \; list \vdash \forall z : \tau . z \in_\tau [\,]_\tau \to z \in_\tau y \tag{8}$$

and the program

$$\begin{array}{l} \sqcup \tau : U_0 . \sqcup d : (\sqcup a : \tau . \sqcup b : \tau . \partial\, (a \equiv_\tau b)) . \\ \quad \sqcup x : \tau \; list . (p(x) ; p(x : \tau \; list) = \; listC(\S, \mathcal{L}_\partial (\sqcup \dagger : \tau \updownarrow \rangle \int \sqcup . \psi_\forall), \S_{/}, \S_\infty, \sqsubseteq, \psi_{\imath})) \end{array}$$

where ψ_8 is the program corresponding with (8) as before. It is the $\mathcal{L}_\partial()$ operator, appearing in the [] part of the *listC* selector which tells us that, when we reach this point in execution, we have succeeded; the argument of *listC* is just verification proof (maybe with some embedded identity relations, which can be extracted easily).

Proof of the remaining conjecture, (8), is trivial (from the definitions of $\in_\tau$ in 2) and is omitted here for lack of space. When it is finished, we have the following program.

$$\begin{array}{l} \sqcup \tau : U_0 . \sqcup d : (\sqcup a : \tau . \sqcup b : \tau . \partial\, (a \equiv_\tau b)) . \\ \quad \sqcup x : \tau \; list . (p(x) ; \\ \qquad\qquad\qquad p(x : \tau \; list) = \\ \qquad\qquad\qquad\quad listC(\S, \\ \qquad\qquad\qquad\qquad \mathcal{L}_\partial (\sqcup \dagger : \tau \updownarrow \rangle \int \sqcup . \sqcup \ddagger : \tau \updownarrow \rangle \int \sqcup . \sqsubseteq : (\ddagger \in_\tau [\,] \tau \mapsto \{ \dashv \updownarrow \int \rceil)), \\ \qquad\qquad\qquad\qquad x_0, x_1, v, \psi_7)) \end{array}$$

The step case of the induction on x, (7) is harder. First, we use the definition (3) of $\in_\tau$ rule to unfold the leftmost occurrence in the conclusion. This gives

$$\vdash \forall y{:}\tau\ \mathit{list}.\partial\,(\forall z{:}\tau.(z \equiv_\tau x_0 \vee z \in_\tau) \rightarrow z \in_\tau y) \tag{9}$$

Note here that I am using the type equality rules to perform and prove correct these rewrites, and not the evaluation rules. As such, the rewrites are effectively under equivalence, and the program does not change.

Now, we introduce the universal quantifier of y and rewrite under logical equivalence, again proving correctness by reference to type equality, to get:

$$y{:}\tau\ \mathit{list} \vdash \partial\,(\forall z{:}\tau.(z \equiv_\tau x_0 \rightarrow z \in y) \wedge \forall z{:}\tau.(z \in x_1 \rightarrow z \in y))$$

As with *Whelk*, the rewriting can be performed automatically, via the *rippling* paradigm of [Bundy *et al.* 90a, Bundy *et al.* 90c]. Again, this step does not change the structure of the synthesised program.

One more logical rewrite gives us

$$\vdash \partial\,(\forall z{:}\tau.(z \equiv_\tau x_0 \rightarrow z \in y)) \wedge \partial\,(\forall z{:}\tau.(z \in x_1 \rightarrow z \in y))$$

again, not contributing to the program structure. The next step, however, $\wedge$ construction, does contribute. Its sub-sequents are:

$$\vdash \quad \partial\,(\forall z{:}\tau.z \equiv_\tau x_0 \rightarrow z \in y) \tag{10}$$
$$\vdash \quad \partial\,(\forall z{:}\tau.z \in_\tau x_1 \rightarrow z \in y) \tag{11}$$

and the program is:

$$\begin{array}{l}
\text{ᑎ}\tau{:}U_0.\,\text{ᑎ}d{:}(\,\text{ᑎ}a{:}\tau.\,\text{ᑎ}b{:}\tau.\partial\,(a \equiv_\tau b)).\\
\quad \text{ᑎ}x{:}\tau\ \mathit{list}.(p(x);\\
\qquad\qquad p(x{:}\tau\ \mathit{list}) =\\
\qquad\qquad\quad \mathit{list}\mathcal{C}(\S,\\
\qquad\qquad\qquad \mathcal{L}_\partial(\,\text{ᑎ}\dagger{:}\tau\ \updownarrow\rangle\int\sqcup.\,\text{ᑎ}\ddagger{:}\tau\ \updownarrow\rangle\int\sqcup.\sqsubseteq{:}(\ddagger \in_\tau [\,]\tau \mapsto \{\dashv\updownarrow\int\rceil)),\\
\qquad\qquad\qquad x_0, x_1, v,\\
\qquad\qquad\qquad \psi_{10} \sqcap \psi_{11}))
\end{array}$$

We show (10) by first simplifying its conclusion:

$$\vdash \partial\,(x_0 \in_\tau y)$$

and then applying induction on y. This can now be demonstrated by appeal to the definition of $\in_\tau$, whose inhabitant proof object instantiates the program. The program now looks like this:

$$\begin{array}{l}
\text{ᑎ}\tau{:}U_0.\,\text{ᑎ}d{:}(\,\text{ᑎ}a{:}\tau.\,\text{ᑎ}b{:}\tau.\partial\,(a \equiv_\tau b)).\\
\quad \text{ᑎ}x{:}\tau\ \mathit{list}.(p(x);\\
\qquad\qquad p(x{:}\tau\ \mathit{list}) =\\
\qquad\qquad\quad \mathit{list}\mathcal{C}(\S,\\
\qquad\qquad\qquad \mathcal{L}_\partial(\,\text{ᑎ}\dagger{:}\tau\ \updownarrow\rangle\int\sqcup.\,\text{ᑎ}\ddagger{:}\tau\ \updownarrow\rangle\int\sqcup.\sqsubseteq{:}(\ddagger \in_\tau [\,]\tau \mapsto \{\dashv\updownarrow\int\rceil)),\\
\qquad\qquad\qquad x_0, x_1, v,\\
\qquad\qquad\qquad \mathcal{M}(\tau)(\S_\prime)(\ddagger) \sqcap \psi_{\infty\infty\infty}))
\end{array}$$

where $\mathcal{M}$ is the proof object of definition (1).

Finally, we appeal to the induction hypothesis of x (called v) to fill in the uninstantiated ϕ_{11} and complete the program.

$$
\begin{array}{l}
\amalg\tau : U_0.\ \amalg d : (\ \amalg a : \tau.\ \amalg b : \tau. \partial\,(a \equiv_\tau b)). \\
\quad \amalg x : \tau\ \mathit{list}.(p(x); \\
\qquad\qquad\qquad p(x : \tau\ \mathit{list}) = \\
\qquad\qquad\qquad\quad \mathit{list}\mathcal{C}(\S, \\
\qquad\qquad\qquad\qquad \mathcal{L}_\partial(\ \amalg\dagger : \tau\ \updownarrow\rangle\int\sqcup.\ \amalg\ddagger : \tau\ \updownarrow\rangle\int\sqcup.\sqsubseteq : (\ddagger \in_\tau [\,]\tau \mapsto \{\dashv\updownarrow\int\rceil)), \\
\qquad\qquad\qquad\qquad x_0, x_1, v, \\
\qquad\qquad\qquad\qquad (\mathcal{M}(\S,))(\ddagger) \sqcap \sqsubseteq))
\end{array}
$$

6 Normalization

We now have our *WTT* module for parameterised `subset`. However, (12) is clearly a higher-order structure, since it has two arguments which are types, τ and d. It also has embedded non-canonical forms, such as the application of $\mathcal{M}$ to its arguments. Before we can convert this program into one runnable directly as a "normal" logic program, we must evaluate and instantiate these structures, respectively. Evaluation of M, as far as the instantiation of its variables may be performed immediately, in the obvious way, as licensed by the $\forall$ type evaluation rules. To instantiate τ and d, we need another type. I will use $\mathbb{N}$ here, although I have omitted its proof rules for reasons of restricted space. All we need to know is that $\mathbb{N} \in U_0$, and that the type $\forall a \in \mathbb{N}.\forall b \in \mathbb{N}.\partial\,(a \equiv_{\mathbb{N}} b)$ is inhabited, thus:

$$
\begin{array}{l}
\amalg a \in \mathbb{N}.\ \amalg b \in \mathbb{N}.e(a); \\
\qquad\qquad e(a{:}\mathbb{N}) = \mathbb{N}\mathcal{C}(\dashv, \\
\qquad\qquad\qquad\qquad f(b); \\
\qquad\qquad\qquad\qquad f(b{:}\mathbb{N}) = \mathbb{N}\mathcal{C}(\lfloor, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{L}_\partial(/ \overset{\mathbb{N}}{=} /), \\
\qquad\qquad\qquad\qquad\qquad\qquad b_0, w, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{R}_\partial(\sqcap : (/ \overset{\mathbb{N}}{=} \int(/) \mapsto \{\dashv\updownarrow\int\rceil)), \\
\qquad\qquad\qquad\qquad x_0, v, \\
\qquad\qquad\qquad\qquad g(b); \\
\qquad\qquad\qquad\qquad g(b{:}\mathbb{N}) = \mathbb{N}\mathcal{C}(\lfloor, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{R}_\partial(\sqcap : (\int(/) \overset{\mathbb{N}}{=} / \mapsto \{\dashv\updownarrow\int\rceil)), \\
\qquad\qquad\qquad\qquad\qquad\qquad b_0, w, \\
\qquad\qquad\qquad\qquad\qquad\qquad v(b_0)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \vdash \forall a{:}\mathbb{N}.\forall b{:}\mathbb{N}.\partial\,(a \equiv_{\mathbb{N}} b)
\end{array}
$$

Again, it is worth emphasising that the production of such an algorithm is a straightforward, and in this case trivially automated, proof. This version of the equality predicate works by counting down the natural numbers; a better version would use a decision procedure built in as a basic type, which would yield a more efficient program.

Given this definition, the evaluation rules of the calculus may be used to reduce the original modular specification to the following:

$$\forall x : \mathbb{N}\ \mathit{list}.\forall y : \mathbb{N}\ \mathit{list}.\partial\,(\forall z : \mathbb{N}.z \in_{\mathbb{N}} \rightarrow z \in_{\mathbb{N}} y)$$

Such an evaluation results in parallel evaluation of the proof object, so the proof object we end up with is convertible into the `subset` relation for lists of naturals as required.

7 Conclusion

In this paper I have discussed how a higher-order extension of the *Whelk* system can help us synthesise modular programs. The example has shown that the system works for programs which are parameterised both in type and in sub-module. I have sketched the outline of an example proof, and shown that it is essentially the same as that for the same example in the *Whelk* system, though the modularity of *W*TT makes things slightly easier, in that it is not necessary to re-synthesised the `member` predicate – we can simply plug in an existing definition as a module.

The consequences of all this are significant. It has been shown elsewhere that logic program synthesis techniques work well for compiling naïve or non-executable specifications into comparatively efficient programs. The question has always been: "what about scaling up?". The modularity of the *W*TT system is one answer to this important question.

8 Acknowledgements

This work has been supported by ESPRIT basic research project #6810, "Compulog II". I am very grateful to my colleagues in the *DR^{E}AM* group at Edinburgh, in particular Alan Smaill, and in the Compulog project for their advice and support. I also thank Frank Pfenning, Andrei Voronkov and David Basin for their comments on the original version of *Whelk*.

References

[Bundy *et al.* 90a] A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L.H. Clarke, editor, *Proceedings of UK IT 90*, pages 221–6. IEE, 1990. Also available from Edinburgh as DAI Research Paper 448.

[Bundy *et al.* 90b] A. Bundy, A. Smaill, and G. A. Wiggins. The synthesis of logic programs from inductive proofs. In J. Lloyd, editor, *Computational Logic*, pages 135–149. Springer-Verlag, 1990. Esprit Basic Research Series. Also available from Edinburgh as DAI Research Paper 501.

[Bundy *et al.* 90c] A. Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 132–146. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 459.

[Constable 82] R. L. Constable. Programs as proofs. Technical Report TR 82-532, Dept. of Computer Science, Cornell University, November 1982.

[Martin-Löf 79] Per Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hanover, August 1979. Published by North Holland, Amsterdam. 1982.

[Wiggins 92a] G. A. Wiggins. Negation and control in automatically generated logic programs. In A. Pettorossi, editor, *Proceedings of META-92*. Springer Verlag, Heidelberg, 1992. LNCS Vol. 649.

[Wiggins 92b] G. A. Wiggins. Synthesis and transformation of logic programs in the Whelk proof development system. In K. R. Apt, editor, *Proceedings of JICSLP-92*, pages 351–368. M. I.T. Press, Cambridge, MA, 1992.

[Wiggins *et al.* 91] G. A. Wiggins, Alan Bundy, I. Kraan, and J. Hesketh. Synthesis and transformation of logic programs through constructive, inductive proof. In K-K.

Lau and T. Clement, editors, *Proceedings of LoPSTr-91*, pages 27–45. Springer Verlag, 1991. Workshops in Computing Series.