

A Partial-Correctness Semantics for Modelling Assembler Programs

Geoffrey Watson and Colin Fidge

School of Information Technology and Electrical Engineering
The University of Queensland
Australia

Abstract

Previous work on formally modelling and analysing program compilation has shown the need for a simple and expressive semantics for assembler level programs.

Assembler programs contain unstructured jumps and previous formalisms have modelled these by using continuations, or by embedding the program in an explicit emulator. We propose a simpler approach, which uses techniques from compiler theory in a formal setting. This approach is based on an interpretation of programs as collections of program paths, each of which has a weakest liberal precondition semantics.

We then demonstrate, by example, how we can use this formalism to justify the compilation of block-structured high-level language programs into assembler.

1 Introduction

An assembler program is written as a sequence of instructions, some of which may be labelled. The ordering of instructions in the program source defines a default sequence for instruction execution. However jumps and branches may interrupt the flow and cause control to be transferred elsewhere. Whereas in block-structured high-level languages all non-linear control flow is encapsulated in the semantics of compound statements, such as alternations and iterations, in assembler programs control flow can be arbitrary.

Thus a major challenge in formalising assembler programs is how to handle this mixture of linear and non-linear control flow. One method is to explicitly model the instruction counter of a ‘virtual machine’ [16, 14, 20, 21, 18]. This simple model iterates over the whole list of instructions.

Each instruction has an address and, after it has executed, each instruction sets the program counter to the address of the next instruction to be executed. Normally the ‘next’ address is that of the next instruction in the sequence, but jumps and branches can set the ‘next’ address to be that of any other instruction. This approach has the virtue that it can model any assembler program, since it effectively implements an assembler code emulator. But it has the drawback that it does not describe any properties above the basic instruction level. It is no easier to use such a model to reason about a program than it is to argue from the original assembler source. In fact it is more difficult, for the latter has an explicit execution sequence, whereas the connection between sequential instructions is indirect in the emulation model. Even reasoning about linear instruction sequences must be done with respect to the emulator’s loop.

Another approach [5, 22] is derived from work on the semantics of older high-level languages that support GOTO statements [6]. This style of semantics is based on the use of continuations to give a meaning to programs that include jumps [11]. A continuation, which encapsulates the ‘effect’ of the rest of the program, is associated with each label, and a jump to a label replaces the current continuation with that associated with the label. Unfortunately continuations are difficult to reason with since the semantics, although theoretically elegant, requires the extra continuation term (which is a mapping from states to states) in addition to the program state. Similarly complex is the use of an exception-like mechanism instead of continuations to model GOTOS in VDM [4], and the use of a ‘moving pointer’ in the program text by Müller-Olm and Wolf [19] which requires the program text to be part of the state.

Therefore our goal is to define a semantics for assembler instructions directly, without an explicit emulator, and in a framework that relies on the pro-

gram state only, without the complication of continuations, or program-valued meta variables. To do this we combine established principles from the disciplines of compiler theory and programming language semantics.

2 Paths and Weakest Liberal Preconditions

The key difficulty in modelling assembler programs is with control flow, and in particular with handling non-local control flow in a simple way. However, compiler theorists [1] have decades of experience with identifying and reasoning about the control flow semantics of assembler programs.

A key concept is the *basic block* which is defined as, [1, page 528]:

a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

Taking the basic block as a fundamental entity in our modelling of assembler programs enables us to use standard sequential composition wherever this is permissible.

Traditionally in compiler theory, control flow is analysed using control flow graphs [13, 3], and the paths through them. We borrow this technique, defining the semantics of a program in terms of the set of all paths through that program. The components of these paths are just basic blocks.

However we combine these familiar ideas from the theory of compilation with a semantics founded on *weakest liberal preconditions* (wlp). Weakest liberal preconditions are a type of predicate transformer [7]. We denote the wlp of statement S with respect to an arbitrary postcondition R by $wlp.S.R$, which is a predicate characterising those initial states from which S will achieve R , provided S terminates.

Weakest liberal preconditions are adequate for reasoning about partial correctness, which ignores the possibility of non-termination, but the stricter *weakest precondition* semantics is normally required when considering total correctness. However, as described elsewhere [9], the two are equivalent for deterministic programs that are feasible, which is generally the case for assembler programs. Weakest liberal preconditions have the advantage that they avoid the need for fixpoint reasoning for loops. Instead they are based on an infinitary logic that allows infinite conjunctions (see,

for example, Definitions 2 and 8). The need for infinitary logics in this style of semantics is well known [2] (both fixpoints and infinite conjunctions require an infinitary logic).

2.1 Avoiding Dead Paths

We define the meaning of a program to be a choice between all possible paths through that program, which semantically is the conjunction of the wlp of all its paths. While it is a simple matter to generate all the *syntactically* valid paths through a program, some of these may not be possible execution paths because they cannot be followed at runtime. Such infeasible or *dead* paths do not contribute to the semantics of the program.

Weakest liberal preconditions handle this situation in an elegant way. Since *true* is the conjunctive identity, any path which has an overall wlp of *true* is effectively ignored. The wlp of a path is composed from the wlp of its component statements. Furthermore, it is a property of wlp that $wlp.S.true \equiv true$ for any statement S . (We represent logical equivalence by ' \equiv '.) So for any path, if at some point the wlp of the *rest* of the path evaluates to *true*, then the wlp of that *whole* path evaluates to *true*. This characteristic automatically ignores dead paths when using a weakest liberal precondition semantics, because the wlp of an un-executable statement is *true* [12].

Thus our aim below is to extract *all* the syntactic paths, give each a wlp semantics, and take the meaning of the program to be the conjunction of the wlp semantics of all of these paths. The key to this strategy is the combination of path extraction and weakest liberal precondition semantics.

2.2 Semantics of Assembler Code

Our semantic modelling of assembler programs using weakest liberal preconditions is done at a number of levels. We divide assembler instructions into those which manipulate program state and those which manage control flow. The former can be given a semantics at the level of individual instructions, but the meaning of the latter (jumps, branches and labels) is context-dependent — the context being provided by the control flow graph (that is, the set of program paths).

State update and sequencing

To illustrate our model we use a simple single-address symbolic assembler language, in which the operations reference a single accumulator. (A more

sophisticated assembler has been modelled elsewhere [10].) This language is listed in Table 1, which also gives assignments that describe the effect of the instructions on the state. In Table 1, c

Instruction	Description	Assignment
loadi c	load acc. with constant	$a := c$
load x	load acc. from store	$a := x$
store x	store a value from acc.	$x := a$
addi c	add a constant to acc.	$a := a + c$
subi c	subtract " "	$a := a - c$
add x	add from store to acc.	$a := a + x$
sub x	subtract " "	$a := a - x$
jmp l	unconditional jump	n/a
bze l	branch to l if acc. zero	n/a
blez l	branch to l if acc. less than or equal zero	n/a

Table 1. A Simple Assembler Language

represents a small constant, x an assembler-level variable, l a program label and a the single accumulator.

The atomic components of paths are instructions which modify the program state. These are modelled by assignments, and assigning a variable v the value of an expression E has the semantics

$$\text{wlp.}(v := E).R \equiv R[E/v]$$

where $R[E/v]$ denotes substitution of free occurrences of v by E in predicate R [7]. The degenerate assignment which changes nothing is represented by **skip** with wlp semantics $\text{wlp.}\text{skip}.R \equiv R$.

Assignments are combined within basic blocks by sequential composition. The semantics of sequential composition is just the functional composition of the wlp function [7]. Let S and T be statements, then

$$\text{wlp.}(S; T).R \equiv \text{wlp}.S.(\text{wlp}.T.R)$$

Thus the sequence of assembler instructions ‘load x ’ followed by ‘subi 1’ which loads the accumulator a with the value of variable x and then subtracts 1, is described by $a := x; a := a - 1$. This is given the semantics

$$\text{wlp.}(a := x).(\text{wlp.}(a := a - 1).R)$$

which reduces to $R[(x - 1)/a]$.

Control flow

Non-sequential control flow occurs when an unconditional jump (or a conditional branch with

condition *true*) is executed. We model the control flow of jumps by the construction of program paths. A path explicitly links the jump or branch with the label to which control is passed, and the transition itself is modelled by sequential composition.

A branch has two aspects, a conditional behaviour which selects between two possible control paths, and the transfer of control itself, either sequential or a jump to a label. The conditional aspect of a branch is modelled by deterministic choice, using a combination of the choice (\sqcap) [7] and coercion ($[G]$) [17] statements. These have the respective weakest precondition semantics

$$\text{wlp.}[G].R \equiv G \Rightarrow R$$

$$\text{and } \text{wlp.}(S \sqcap T).R \equiv \text{wlp}.S.R \wedge \text{wlp}.T.R$$

Note that the wlp of a *false* coercion evaluates to *true* regardless of the value of R . In isolation \sqcap describes nondeterministic choice, but we only use it in constructs where the components of the choice contain complementary coercions. This ensures that the choice is *deterministic*. For instance:

$$([G]; S) \sqcap ([\neg G]; T)$$

which describes the deterministic choice of alternatives based on the truth of predicate G .

Using this construct, the weakest precondition semantics of a branch which transfers control to label L if some predicate G is true is

$$(G \Rightarrow S) \wedge (\neg G \Rightarrow T)$$

where S is the meaning of the program at label L and T that of the program sequentially following the branch instruction. The presence of S and T in this semantics for a branch reveals the basic problem with trying to give a self-contained semantics for such instructions. In effect the branch semantics is parameterised by the meaning of the surrounding code. It is for this reason that we seek a model that avoids the need to invoke such ‘whole-program’ arguments.

Loops are effected by a jump back to previously executed instructions. A loop can only terminate if it contains a branch, and will only do so when the branch condition changes and redirects control flow out of the loop. In a path semantics a loop will be described by a (possibly infinite) *family* of paths which share a common structure. This is done by allowing a path to consist of repeated copies of an action and allowing a choice between an arbitrary number of possible paths. Definition 1 defines a repetition of sequential composition that will be used to describe a loop in Section 3.2.

Definition 1 (Repeated Sequential Composition)

$$S^n \stackrel{\text{def}}{=} \begin{cases} \text{skip}, & n = 0 \\ \underbrace{S; \dots; S}_{n \text{ times}}, & n > 0 \end{cases}$$

In the description of loops iterative sequential composition is usually bound by the choice quantifier which has a semantics given by Definition 2.

Definition 2 (Generalised Choice Quantifier)

$$\text{wlp}.\left(\prod_{n \in \mathbb{N}} S_n\right).R \stackrel{\text{def}}{=} \bigwedge_{n \in \mathbb{N}} (\text{wlp}.S_n.R)$$

This definition uses the infinitary conjunction quantifier.

3 Path Extraction

There are standard compiler techniques for determining the basic blocks in an assembler program. The conversion of an assembler program from a traditional list of instructions to a list of basic blocks is a straightforward algorithmic process [1, page 529].

A division between basic blocks occurs whenever there is a branch instruction (conditional or unconditional) since this may direct control elsewhere. The branch is the last instruction in one block and a new block starts with the following instruction. A division between basic blocks also occurs at any instruction to which control may pass from elsewhere in the program. Typically this is a labelled instruction, and in this case the labelled instruction is the first instruction in the new block.

The generation of the list of basic blocks in a program involves the identification of the *leader* of each block, which is its first instruction. Since we are constructing an abstract representation of a program we assign a unique *tag* to each basic block rather than identifying it by its first instruction.

The second concept we adopt from compiler technology is that of a *program path*. A program is assumed to have a distinguished instruction as its start point. A program *path* is a sequence of instructions which could possibly be executed during some run of the program, starting from the program's start point. Paths are maximal, in that a path is only completed when the program terminates.

The generation of program paths is, in principle, straightforward. Program paths can be described

by sequences of basic blocks, each block being followed by a block to which control can legitimately pass after completion of the first block. A conditional branch at the end of a block will result in the path 'splitting', that is, there are now two possible paths which share a common initial execution segment. This principle can be applied iteratively, starting from the start point of the program, to generate all the paths.

However the usefulness of this semantics is threatened by size issues. Path splitting increases the number of paths, but this is just a case of the size of the set of paths reflecting the complexity of the original program. More significantly, the presence of loops can explode both the number of paths and the length of individual paths in such a way that either may become infinite.

3.1 Path Segments

Most basic blocks can be described by a single *path segment*, which is a triple containing the start tag of the block, its body, and a next tag, which defines where control passes on completion of the block. However, where a block terminates in a conditional branch there are two possibilities for the next tag, so such a block corresponds to *two* path segments, each representing one of the possible behaviours.

We define the function \mathfrak{S} that converts a basic block to its corresponding path segment(s). The *body* of each of these segments is obtained from the body of the block by detaching any terminating jump or branch from the sequence of instructions, and conjoining the remaining instructions by sequential composition ';'. In the case where the block terminates in a conditional branch, the bodies of the path segments are extended by appending a coercion defining the condition necessary for that path to be taken.

Definition 3 (Path Segment Generation) *The function \mathfrak{S} generates the one or two path segments corresponding to a given basic block. \mathfrak{S} is of the type*

$$\text{tag} \times \text{seq inst} \times \text{tag} \rightarrow \mathbb{P}(\text{tag} \times \text{inst}' \times \text{tag})$$

where inst is the type of assembler instructions, and inst' is a sequential composition of assignment statements and coercions. That is, \mathfrak{S} maps basic blocks to sets of path segments.

If B is a sequence of state-modifying assembler instructions and B' the sequential composition of

the assignments corresponding to those instructions, a is the accumulator, and j, k, l are tags, then the function \mathfrak{S} for the language of Table 1 is defined by cases on the form of the instruction sequence as follows.

$$\begin{aligned}\mathfrak{S}((j, B, k)) &= \{(j, B', k)\} \\ \mathfrak{S}((j, B \text{ bze } l, k)) &= \{(j, B'; [a \neq 0], k), \\ &\quad (j, B'; [a = 0], l)\} \\ \mathfrak{S}((j, B \text{ blez } l, k)) &= \{(j, B'; [a > 0], k), \\ &\quad (j, B'; [a \leq 0], l)\} \\ \mathfrak{S}((j, B \text{ jmp } l, k)) &= \{(j, B', l)\}\end{aligned}$$

When B is the empty sequence then B' is **skip**.

Note that this construction embodies the notion that branches can be defined by $([G]; S) \sqcap ([\neg G]; T)$ (provided that we have access to code segments S and T) — the coercions are explicitly adjoined to the two paths generated for the branch, while the choice is implicit in the definition of the semantics of the program as the conjunction of all its paths.

3.2 Program Paths

We have shown how we can decompose an assembler program into a sequence of basic blocks, and then associate each basic block with either one or two path segments. This gives a collection of path segments from which we can construct the actual paths through the program. One way of achieving this using conventional compiler technology is via a control flow graph [13, 3].

Definition 4 (Control Flow Graph) *The control flow graph of a program P is a triple (s, V, E) , such that:*

- *The pair (V, E) defines a finite directed graph with a single source.*
- *The set V is the collection of vertices of the graph. Each vertex is labelled with one of the identifying tags in the list of basic blocks of P .*
- *The node s is the initial node of the graph, and is the start tag of the program P .*
- *The set E is the collection of edges of the graph. Each edge is labelled with the body of one of the path segments generated from program P .*
- *There is a one-to-one mapping between the edges E and the path segments of P such that:*

1. *The vertices defining an edge are labelled with the start and end tags of the corresponding path segment.*
2. *The direction of an edge is from the start to the end node.*
3. *Each edge is labelled with the statement in the body of the corresponding path segment.*

Thus all flow graphs are finite, and can be constructed in a simple fashion from the set of path segments derived from the program. Where a program loops, its flow graph will contain a cycle. We define the semantics of a program in terms of the paths through its flow graph. (A formal semantics of high-level program paths has been given elsewhere [15]. Here we address assembler programs, and are primarily concerned with using collections of paths to define assembler code control flow.)

Given an assembler program P , we can construct its control flow graph as per Definition 4, and then identify execution paths as walks through this graph.

Definition 5 (Program Execution Path) *Given an assembler program P , with start tag s , and a single exit, with tag e , an execution path through the control flow graph of P is a sequence of edges of this flow graph which satisfies the following conditions.*

- *The initial vertex of the first edge in the sequence is labelled with s , the start tag of P .*
- *For each consecutive pair of edges in the sequence the final vertex of the first is the same as the initial vertex of the second.*
- *The final vertex of the last edge in the sequence is labelled with e , the exit tag of P .*

An execution path defines a sequence of statements that we shall call the *program path*. The semantics of an execution path is just that of the program path that it defines.

Definition 6 (Program Path) *Given an assembler program P , with start tag s , and a single exit, with tag e , then, for each execution path of P (as defined by Definition 5), there corresponds a program path, which is the sequential composition of the statements labelling the edges of that execution path, taken in order.*

We can now define the semantics of a program as the conjunction of the semantics of all its program paths.

Definition 7 (Flow Graph Path Semantics) *The wlp of a program P with respect to an arbitrary postcondition R is given by*

$$\text{wlp}.P.R \equiv \bigwedge_{p \in \text{paths}(P)} \text{wlp}.p.R$$

where $\text{paths}(P)$ (which may be unbounded) is the set of all program paths defined by the flow graph of P as described in Definition 6.

As we have noted, a program path may be infinite, and in order to be able to reason about a program we want to express its paths in a succinct closed expression where possible. One way of managing this is by means of the iterative constructs given in Definitions 1 and 2. These constructs allow certain patterns that occur in families of paths representing loops to be described succinctly. Algorithms for generating expressions of this type from control flow graphs have been described [23], however in simple cases, such as the example in Section 4, the expression summarizing the control flow graph can readily be derived by inspection.

This formulation of path semantics limits us to that class of programs for which the program paths can be described by finite program path expressions. However, this class includes all programs compiled from conventional block-structured languages. For example the definition of the weakest liberal precondition semantics for the **while** construct [8] in this style is as follows.

Definition 8 (Partial-correctness loop semantics)

$$\text{wlp}.\langle \mathbf{while} \ B \ \mathbf{do} \ S \ \mathbf{end} \rangle.R \stackrel{\text{def}}{=} \bigwedge_{n \in \mathbb{N}} (\text{wlp}.\langle ([B]; S)^n; [\neg B] \rangle.R)$$

This definition provides an illustration of how weakest liberal preconditions can encapsulate loop semantics. If we consider the expression on the right-hand side, then, for any particular state one of the following two conditions holds.

1. There is some k such that, in the sequence $(\langle [B]; S \rangle)^k$, expression B is true before each of the k iterations of S and false after the k^{th} one.
2. There is no k for which condition 1 holds.

In the former case, if n is less than k the final coercion $[\neg B]$ is false, while if n is greater than k

the initial coercion $[B]$ is false for the $k + 1^{\text{th}}$ iteration. Either way the false coercion reduces the wlp for that path to *true* (for all R) and so the conjunction over all the paths collapses to the single term $\text{wlp}.\langle ([B]; S \rangle)^k; [\neg B] \rangle.R$, thus defining the loop semantics to consist of k iterations as expected. In case 2 the whole conjunction just collapses to *true*, which indicates an infinite loop.

4 An Example

In this section we give an example of the application of this semantic framework to program compilation. We ask the question ‘Is assembler code A a correct compilation of high-level language language program P ?’ This can be answered by showing that $\text{wlp}.P.R \equiv \text{wlp}.A.R'$, for all postconditions R formulated in the context of program P (that is, only referring to P ’s variables), and where postcondition R' is derived from R by replacing references to P ’s high-level language variables by references to the equivalent assembler-level variables in A .

For the present purpose we only consider the simple case where there is a direct one-to-one correspondence between the variables in the two programs (in the example all are integers). In this case we can use the same names for the corresponding integer variables in the high-level language and in the assembler code, and so we can identify R and R' since they are syntactically identical predicates. Obviously where more complex data types are used in the program P , for instance floating-point values, arrays or records, a more sophisticated mapping is needed.

The required behaviour is defined by the high-level program. A consequence of this is that in showing the ‘compilation equivalence’ of a high-level program and an assembler program, we can assume that none of the purely low-level data elements used by the assembler (such as registers, or call-stacks) appear in the postcondition. We shall call such variables *framework* variables, and so we may assume that the predicate R does not refer to any framework variables.

We begin with a simple high-level language program, which we shall call P . P assumes that the variable x is a positive number, and sets the variable y to the smallest power of 2 not less than x . P consists of the following composition of an assignment and a **while** loop.

$y := 2$; **while** $y < x$ **begin** $y := y + y$ **end**

P might be compiled to the assembler code shown in Figure 1 — program A .

```

S : loadi 2
    store y
L : load x
    sub y
    blez E
    load y
    add y
    store y
    jmp L
E : ...

```

Figure 1. Assembler Program A

The list of basic blocks for program A is as follows.

1. S : loadi 2, store y
2. L : load x, sub y, blez E
3. load y, add y, store y, jmp L

The control flow of program A is described by converting this list to a list of path segments. We annotate each block explicitly with its start and next tags, assigning block 3 the fresh tag !A. As described in Section 3, we express the conditional branch at the end of block 2 by duplicating this block and terminating the two blocks with complementary coercions, $[a > 0]$ and $[a \leq 0]$, which describe the state of accumulator a . Setting the next tag of block 3 to L implements the unconditional jump to label L at the end of that block. We also express the load and store sequences for each block more succinctly as a single multiple assignment statement which encapsulates the update of the program state performed by the block.

This gives the following collection of four path segments.

1. (S, $a, y := 2, 2$, L)
- 2a. (L, $a := x - y; [a > 0]$, !A)
- 2b. (L, $a := x - y; [a \leq 0]$, E)
3. (!A, $a, y := y + y, y + y$, L)

The control flow graph for program A, derived from these path segments, is shown in Figure 2. The set of paths for this graph includes an arbitrary number of traversals of the loop (the actual number depending on the state from which the program is executed).

The wlp of program A is calculated as a choice between *all* possible paths. We do this by determining a closed path expression for A by inspection from the graph. The flow-graph contains a cycle around tag L, representing a loop whose possi-

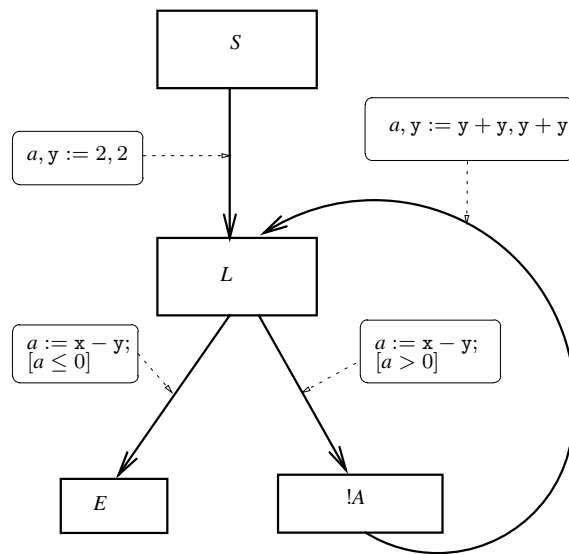


Figure 2. Flow Graph for Program A

ble iterations are:

- 0 : skip
- 1 : $a := x - y; [a > 0]; a, y := y + y, y + y$
- 2 : $a := x - y; [a > 0]; a, y := y + y, y + y;$
 $a := x - y; [a > 0]; a, y := y + y, y + y$
and so on ...

This pattern can be represented by the closed path expression:

$$\prod_{n \in \mathbb{N}} (a := x - y; [a > 0]; a, y := y + y, y + y)^n$$

The whole collection of paths from the start tag S to the exit tag E is thus:

$$a, y := 2, 2;$$

$$\left(\prod_{n \in \mathbb{N}} (a := x - y; [a > 0]; a, y := y + y, y + y)^n \right);$$

$$a := x - y; [a \leq 0]$$

We show that A is a correct compilation of P.

Proof

To do this we evaluate the wlp of the path expression for A and show that it equals the wlp for P. At steps 1 and 3 we use the following facts. Firstly, that $x := E; [P]$ is equivalent to $x := E; [P[E/x]]$. Secondly, that where we have a sequence of statements in which some assembler level framework item only occurs on the left-hand side of assignments, and never in the coercions, then we can eliminate all references to that item in a wlp expression. This is allowable because in compilation

proofs we can assume that such framework items never occur in the predicate R .

wlp.A.R

$$\begin{aligned}
&= \text{wlp.}(a, y := 2, 2; \\
&\quad (\prod_{n \in \mathbb{N}} (a := x - y; [a > 0]; \\
&\quad \quad a, y := y + y, y + y)^n); \\
&\quad \quad a := x - y; [a \leq 0]).R \\
&\equiv \text{push assignments into coercions} \\
&\text{wlp.}(a, y := 2, 2; \\
&\quad (\prod_{n \in \mathbb{N}} (a := x - y; [x - y > 0]; \\
&\quad \quad a, y := y + y, y + y)^n); \\
&\quad \quad a := x - y; [x - y \leq 0]).R \\
&\equiv \text{simplify coercions} \\
&\text{wlp.}(a, y := 2, 2; \\
&\quad (\prod_{n \in \mathbb{N}} (a := x - y; [y < x]; \\
&\quad \quad a, y := y + y, y + y)^n); \\
&\quad \quad a := x - y; [\neg (y < x)]).R \\
&\equiv \text{remove the assignments to } a \text{ since } a \text{ only} \\
&\quad \text{appears on LHS of assignments} \\
&\text{wlp.}(y := 2; (\prod_{n \in \mathbb{N}} ([y < x]; y := y + y)^n); \\
&\quad \quad [\neg (y < x)]).R \\
&\equiv \text{wlp definitions and Definition 2} \\
&\text{wlp.}(y := 2). \\
&\quad (\bigwedge_{n \in \mathbb{N}} \text{wlp.}(((y < x]; y := y + y)^n); \\
&\quad \quad [\neg (y < x)]).R \\
&\equiv \text{Definition 8 and wlp definitions} \\
&\quad \text{Convert to high-level variables} \\
&\text{wlp.}(y := 2; \\
&\quad \quad \text{while } x < y \text{ do } y := y + y \text{ end}).R \\
&= \text{wlp.P.R}
\end{aligned}$$

5 Conclusions

In this paper we have addressed the problem of giving a semantics to assembler programs in the predicate transformer style. Previous approaches involved indirect models of control flow, which made reasoning difficult. We have shown how familiar ideas from the theory of program compilation can be combined with those of partial correctness semantics to define a path semantics for assembler programs, suitable for reasoning about compilation strategies.

In our example we have demonstrated the correctness of the complete compilation of a tiny program. But, as with any approach of this kind, the issue of scalability from small examples to practical cases must be addressed. Indeed, the overall goal of this work is not to formally prove the correctness of every compilation performed, but

to prove the correctness of generic compilation laws. This can be done, using the formalism described above, simply by treating the program-specific variable names, types and operators symbolically [9].

Acknowledgements This research was funded by Australian Research Council Large Grant A00104650, *Verified Compilation Strategies for Critical Computer Programs*.

References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.
- [2] R.-J. R. Back. Proving total correctness of nondeterministic programs in infinitary logic. *Acta Informatica*, 15:233–249, 1981.
- [3] J.-F. Bergeretti and B. A. Carré. Information-flow and data-flow analysis of while-programs. *ACM TOPLAS*, 7(1):37–61, Jan. 1985.
- [4] D. Bjørner. Experiments in block-structured goto-modelling: Exits vs. continuations. In D. Bjørner, editor, *Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*, pages 216–247. Springer-Verlag, 1980.
- [5] E. Börger and I. Durdanović. Correctness of compiling Occam to transputer code. *The Computer Journal*, 39(1):52–92, 1996.
- [6] A. de Bruin. Goto statements. In J. de Bakker, editor, *Mathematical theory of program correctness*, Series in Computer Science, chapter 10, pages 401–443. Prentice-Hall International, 1980.
- [7] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [8] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
- [9] C. Fidge, K. Lermer, and G. Watson. Verifying program transformations using partial correctness semantics. Technical report 02-45, Software Verification Research Centre, Dec. 2002.
- [10] C. J. Fidge. Timing analysis of assembler code control-flow paths. In L.-H. Eriksson and P. Lindsay, editors, *FME 2002*, volume 2391 of *Lecture Notes in Computer Science*, pages 370–389. Springer-Verlag, 2002.
- [11] M. J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction*. Springer-Verlag, 1979.
- [12] I. J. Hayes, C. J. Fidge, and K. Lermer. Semantic characterisation of dead control-flow paths. *IEE Proceedings—Software*, 148(6):175–186, Dec. 2001.
- [13] M. S. Hecht. *Flow Analysis of Computer Programs*. North Holland, New York, 1977.

- [14] C. A. R. Hoare. Refinement algebra proves correctness of compiling specifications. In C. Morgan and J. Woodcock, editors, *3rd Refinement Workshop*, pages 33–48. Springer-Verlag, 1990.
- [15] K. Lerner, C. J. Fidge, and I. J. Hayes. Formal semantics for program paths. In J. Harland, editor, *Computing: The Australasian Theory Symposium (CATS 2003)*, volume 78 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.
- [16] M. S. Mannasse and G. Nelson. Correct compilation of control structures. Technical report, AT&T Bell Laboratories, Sept. 1984.
- [17] C. Morgan and T. Vickers. Types and invariants in the refinement calculus. *Science of Computer Programming*, 14:281–304, 1990.
- [18] M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 1997.
- [19] M. Müller-Olm and A. Wolf. On the translation of procedures to finite machines: Abstraction allows a clean proof. In G. Smolka, editor, *Programming Languages and Systems: ESOP'2000*, volume 1782 of *Lecture Notes in Computer Science*, pages 290–304. Springer-Verlag, 2000.
- [20] T. S. Norvell. Machine code programs are predicates too. In D. Till, editor, *Sixth Refinement Workshop*, pages 188–204. Springer-Verlag, 1994.
- [21] A. Sampaio. *An Algebraic Approach to Compiler Design*, volume 4 of *AMAST Series in Computing*. World Scientific, 1997.
- [22] S. Stepney. *High Integrity Compilation: A Case Study*. Prentice-Hall, 1993.
- [23] M. Wegman. Summarizing graphs by regular expressions. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 203–216. ACM Press, 1983.