ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**D. J. Duke , University of York, UK and A.S. Evans, University of Bradford,
UK (Eds)**

2nd BCS-FACS Northern Formal Methods Workshop

Proceedings of the 2nd BCS-FACS Northern Formal Methods
Workshop, Ilkley, 14-15 July 1997

Modelling Program Compilation in the Refinement Calculus

C.J. Fidge

Springer

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Modelling Program Compilation in the Refinement Calculus

C. J. Fidge

Software Verification Research Centre

School of Information Technology

The University of Queensland

Queensland 4072, Australia

cjf@it.uq.edu.au

Abstract

We show how compilation of high-level language programs to assembler code can be formally represented in the refinement calculus. New operators are introduced to widen the modelling language to encompass assembler code. A compilation strategy is then embodied as a set of derived refinement rules.

1 Introduction

The idea of modelling program compilation as a formal development procedure has surfaced many times in the literature, but has presented a significant challenge. This has resulted in complex models, often using new, unfamiliar formalisms.

Our goal is to develop a model of program compilation within the already-familiar *refinement calculus*. Normally the refinement calculus translates an abstract requirements *specification* into a programming language *implementation*, using guarded command language augmented with specification statements as the underlying *modelling* notation. In the context of compilation, however, our ‘specification’ is a high-level language (HLL) program, and our desired ‘implementation’ is assembler code.

In this paper we focus on refinement of a program’s *control flow*. In the past this has proven to be a major stumbling block because assembler code does not necessarily obey structured programming principles, and is therefore awkward to model in the guarded command language. Here we present a simple way of modelling assembler language constructs in the guarded command language, thus widening the modelling language to encompass the new application domain. Several simple ‘compilation laws’ and a small example are then given to show how this model can be effectively exploited.

2 Previous work

Recently, Norvell [15, 16] showed how a simple compilation strategy for a small programming language could be derived from formal models of the HLL and assembler languages. His assembler code model was complicated, however, by the need for assembler instructions to be ‘interpreted’ by an imaginary abstract machine. Earlier, Hale [5, §7.2.2], Fränzle and Müller-Olm [4, p. 303], Hoare [9, §4] [10, §4] and He Jifeng [8, §6.2] used similar models, with the semantics of assembler instruction sequences given by their interpretation.

This need for an interpreter stems from the potentially unstructured control flow of assembler programs—such code cannot be easily represented in the structured guarded command language used by the refinement calculus. Unfortunately the presence of this interpreter introduces a significant paradigm shift during attempts to model compilation as refinement—the source and target languages are represented in very different ways. Indeed, Börger and Durdanović, in defining a compilation-as-refinement strategy for occam programs, go as far as stating, “we could not make reasonable use of any of the many refinement notions in the literature” [2].

Nevertheless, Back [1] demonstrated through a case study that a guarded command language subset *is* capable of representing assembler-like programs, albeit clumsily. Lermer and Fidge [11] then exploited this model to show how compilation laws could be expressed in the standard refinement calculus.

$S ::=$	$[[\text{var } v \bullet S]]$	– variable block
	skip	– null statement
	$\vec{v} := \vec{E}$	– (multiple) assignment
	$S_1 ; S_2$	– sequential composition
	$\text{if } (\ i \bullet B_i \rightarrow S_i) \text{ fi}$	– conditional composition
	$\text{do } (\ i \bullet B_i \rightarrow S_i) \text{ od}$	– iterative composition
	$v : [P, Q]$	– specification
	$\{P\}$	– assumption
	$[Q]$	– coercion

Figure 1: Wide-spectrum modelling language for refinement [12]. Let S be a statement in the language; v a variable name; P and Q predicates; E an expression; B a boolean expression. Scoping brackets $[[\cdot \cdot]]$ may be omitted when programs are displayed vertically [13, pp. 55–6].

$v := E$	$=$	$v := E$
$S_1 ; S_2$	$=$	$S_1 ; S_2$
if B then S_1 else S_2 end if	$=$	$\text{if } B \rightarrow S_1 \parallel \neg B \rightarrow S_2 \text{ fi}$
while B loop S end loop	$=$	$\text{do } B \rightarrow S \text{ od}$

Figure 2: High-level language statement definitions. The HLL statements on the left are equivalent to the underlying model on the right.

The approach presented herein is inspired by various aspects of these previous attempts, especially the assembler-language models of Norvell [15, 16] and Back [1], and our own earlier work in the area [11], but is considerably more elegant than any previous method.

3 Languages

Our ‘compilation laws’ are expressed in the well-known refinement calculus developed by Morgan, et al [12, 14]. As usual, we use an extended guarded command language as the ‘wide-spectrum’ modelling language. Both high-level language statements and individual assembler instructions are then merely subset notations.

3.1 Modelling language

As shown in Figure 1, our modelling language consists of Dijkstra’s guarded command language, augmented with specification statements [12, §23][13]. We freely use the usual refinement calculus laws and definitions [12, App. C] for manipulating this language.

3.2 High-level language statements

Our source language is an Ada-like sequential programming language, featuring the usual structured programming statements for assignment, sequence, choice and iteration. However, as shown in Figure 2, these constructs merely denote a distinguished subset of our modelling language.

$$\begin{aligned}
 \text{load } reg_i, loc_j &= pc, reg_i := pc + 1, mem(loc_j) \\
 \text{store } loc_j, reg_i &= pc, mem(loc_j) := pc + 1, reg_i \\
 \text{jump } loc_j &= pc := loc_j \\
 \text{brtrue } reg_i, loc_j &= \text{if } reg_i = \text{true} \rightarrow pc := loc_j \\
 &\quad \parallel reg_i \neq \text{true} \rightarrow pc := pc + 1 \\
 &\quad \text{fi} \\
 \text{brfalse } reg_i, loc_j &= \text{if } reg_i = \text{false} \rightarrow pc := loc_j \\
 &\quad \parallel reg_i \neq \text{false} \rightarrow pc := pc + 1 \\
 &\quad \text{fi} \\
 \text{noop} &= pc := pc + 1 \\
 \text{eval } reg_i, E &= pc, reg_i: [\text{true}, reg_i = E]
 \end{aligned}$$

Figure 3: Assembler instruction definitions. Let $loc_1, \dots, loc_m$ denote memory locations; pc the program counter, ranging over a subset of these locations; $reg_1, \dots, reg_n$ general-purpose registers; mem a location-indexed array of values representing the machine’s memory.

3.3 Assembler instructions

Our target assembler language instructions act on a number of new variables introduced into the program, representing observable aspects of the target processor [15, p. 193]. Since our concern in this paper is with control flow, we are mainly interested in the program counter pc herein, introducing the other variables informally as needed.

As shown in Figure 3, individual assembler instructions are modelled as multiple assignments acting on these variables [15, §6.1][5, p. 133][8, §6.1.2]. Here we need only a few instructions, to load and store data values, perform unconditional jumps and conditional branches, do nothing, and evaluate expressions.

Pseudo-instruction ‘eval’ denotes evaluation of a *HLL* expression, with the result left in a register. Expression evaluation has been well explored elsewhere, and has no impact on control flow, so we refer the interested reader to previous work on formalising compilation of expressions for further detail [3, §7.3.2][8, §8.2]. In effect, eval is a temporary, intermediate statement in our model, requiring further refinement to primitive assembler instructions, but we do not consider this herein. (The definition of eval in Figure 3 leaves the program counter value unspecified—in use below we are careful to always augment this statement with an explicit final program counter value.)

4 Creating assembler programs from instructions

So far our language models differ little from their predecessors. The challenge now is to devise suitable operators for composing the assembler instructions in Figure 3 to form complete assembler programs. Furthermore, to support refinement, the operators must work equally well on *any* statement in our modelling language, not just assembler instructions!

There are two aspects to the problem, labelling instructions with the locations at which they will reside in memory, and composing sequences of instructions together. We satisfy these requirements through two new definitions.

4.1 Labelling statements

An assembler instruction performs its required function only when the program counter points to the particular location at which the instruction resides, otherwise the instruction “does” nothing [15, §6.3.1]. For instance, the behaviour of instruction s at location l could be modelled by the statement $do\ pc = l \rightarrow s\ od$. Instruction s executes only if the program counter currently points to it. The iteration operator allows for the possibility that s is a branch or jump instruction that returns control to location l !

Here we generalise this concept by allowing *any* statement in our modelling language to be associated with a *set* of memory locations.

Definition 1 (Labelling) A statement S can be labelled with a set of locations L using the labelling operator ‘:’.

$$L : S \stackrel{\text{def}}{=} \text{do } pc \in L \rightarrow S \text{ od}$$

□

The statement “executes” only while the program counter points to one of the locations. Otherwise the construct has no effect, i.e., behaves like skip. As illustrated below, allowing a *set* of labels is helpful during refinement because S may prove to be a compound statement that refines to several distinct instructions, residing in several memory locations.

As a syntactic convenience we allow singleton label sets to be written without braces:

$$l : S = \{l\} : S.$$

Also, since label sets are usually contiguous in practice, we often write $l_1 \dots l_2$ to denote all locations between l_1 and l_2 , inclusive:

$$l_1 \dots l_2 = \{l \mid l_1 \leq l \wedge l \leq l_2\}.$$

4.2 Non-sequential statement composition

The next requirement is the ability to compose sequences of instructions together. The challenge is that a ‘sequence’ of assembler instructions is not necessarily performed sequentially! Branches and jumps may cause instructions to be executed in an order totally different than their textual one.

We therefore introduce a “non-sequential” composition operator on labelled statements. To support subsequent refinement, the operator constructs a labelled statement.

Definition 2 (Non-sequential composition) Two labelled statements $L_1 : S_1$ and $L_2 : S_2$ can be composed using the non-sequential composition operator ‘;’.

$$(L_1 : S_1) ; (L_2 : S_2) \stackrel{\text{def}}{=} (L_1 \cup L_2) : (L_1 : S_1 ; L_2 : S_2)$$

□

It may seem curious that our ‘non-sequential’ operator is defined in terms of the *sequential* composition operator ‘;’. Using Definition 1, we can express the definition of ‘;’ in full as

$$\begin{aligned} (L_1 : S_1) ; (L_2 : S_2) &= \text{do } pc \in L_1 \cup L_2 \rightarrow \text{do } pc \in L_1 \rightarrow S_1 \text{ od} ; \\ &\quad \text{do } pc \in L_2 \rightarrow S_2 \text{ od} \\ &\text{od} . \end{aligned}$$

It thus gives preference to statement S_1 when the program counter points into *both* L_1 and L_2 .

However, in practice, we always expect L_1 and L_2 to be disjoint, i.e., $L_1 \cap L_2 = \emptyset$. In this case standard program transformation rules allow us to simplify this definition considerably, because choice between the two statements is always mutually exclusive. Thus, for *disjoint* L_1 and L_2 ,

$$\begin{aligned} (L_1 : S_1) ; (L_2 : S_2) &= \text{do } pc \in L_1 \rightarrow S_1 \\ &\quad \parallel pc \in L_2 \rightarrow S_2 \\ &\text{od} . \end{aligned}$$

In this context, therefore, ‘;’ is ‘non-sequential’ since it allows its operands to execute in *either* order, depending on the value of the program counter, and exhibits no bias towards either statement. It also allows repetition as long as the program counter points to *either* labelled statement.

Finally, we note that it is possible to combine the labelling and non-sequential composition operators in ways that are not useful. Given two distinct locations l_1 and l_2 , then

$$\{l_1\} : (l_1 : S_1 ; l_2 : S_2) \neq \{l_1, l_2\} : (l_1 : S_1 ; l_2 : S_2) .$$

If the program counter initially equals l_2 , the specification on the right will perform statement S_2 , but the one on the left will behave like skip. We consider a statement such as that on the left to be ill-formed.

Definition 3 (Well-formedness) When labels are nested, the construct is well-formed only when all ‘inner’ labels are contained within the ‘outer’ label set. That is,

$$L : (L_1 : S_1 ; \dots ; L_n : S_n)$$

is well-formed only if $L \supseteq (L_1 \cup \dots \cup L_n)$.

□

4.3 Properties

The following properties of the labelling and non-sequential composition operators are frequently used below. Each can be readily proven by expanding the definitions into their guarded command language equivalents.

Property 1 (Associativity) *Non-sequential composition is associative for disjoint sets of labels. If L_1 , L_2 and L_3 are all mutually disjoint, then*

$$(L_1 : S_1 \circ L_2 : S_2) \circ L_3 : S_3 = L_1 : S_1 \circ (L_2 : S_2 \circ L_3 : S_3) .$$

□

We can therefore omit bracketting around the non-sequential composition operator. (We have also assumed that ‘ $\circ$ ’ binds more tightly than ‘ $\circ$ ’, but more loosely than other modelling language operators.)

Property 2 (Commutativity) *The order of labelled instructions composed using non-sequential composition is irrelevant for disjoint sets of labels. If L_1 and L_2 are disjoint, then*

$$L_1 : S_1 \circ L_2 : S_2 = L_2 : S_2 \circ L_1 : S_1 .$$

□

Redundant ‘outer’ labels can be removed, once all statements are explicitly labelled.

Property 3 (Flattening) *If $L_1, \dots, L_n$ are all mutually disjoint, then*

$$(L_1 \cup \dots \cup L_n) : (L_1 : S_1 \circ \dots \circ L_n : S_n) = L_1 : S_1 \circ \dots \circ L_n : S_n .$$

□

A statement with an empty label set does nothing.

Property 4 (Null label)

$$\emptyset : S = \text{skip}$$

□

If there is only one label via which a statement can be reached, then we can insert an assumption [12, p. 11] to explicitly state the initial program counter value before the statement.

Property 5 (Unique entry point) *If L_1 and L_2 are disjoint, and $l_1 \in L_1$ and $\bar{L}_1 = L_1 \setminus \{l_1\}$, then*

$$\begin{aligned} & \{pc \notin \bar{L}_1\} ; (L_1 : (S_1 ; [pc \notin \bar{L}_1]) \circ L_2 : (S_2 ; [pc \notin \bar{L}_1])) \\ = & \{pc \notin \bar{L}_1\} ; (L_1 : (\{pc = l_1\} ; S_1 ; [pc \notin \bar{L}_1]) \circ L_2 : (S_2 ; [pc \notin \bar{L}_1])) . \end{aligned}$$

□

In other words, the *only* way to begin executing code within S_1 is via label l_1 . This is enforced by the initial assumption and the coercion following each statement. We are therefore free to assume that the program counter *equals* l_1 immediately before S_1 starts executing. (S_1 may make use of labels in L_1 other than l_1 while it is executing, however.)

5 Refinement rules for compilation

In this section we define refinement rules for translating a ‘specification’, expressed in the high-level language statements from Section 3.2, into an ‘implementation’, expressed using the assembler instructions from Section 3.3 composed with the operators from Section 4. In effect, these refinement rules define a program compilation strategy.

The refinement goal is to construct an assembler program of the form

$$\begin{array}{l} l_1 : i_1 \% \\ l_2 : i_2 \% \\ \vdots \\ l_n : i_n \end{array}$$

where i_1 to i_n are individual assembler instructions, and $l_1, \dots, l_n$ is a contiguous sequence of locations at which the instructions are placed. Since we usually do not know in advance how many instructions will be generated for each high-level language statement, we make frequent use of symbolic constants for labels, on the understanding that these will be instantiated with consecutive numbers once the refinement is complete.

5.1 Introduce machine-dependent constructs

Given a high-level language program S , the first step is to introduce new variables modelling machine-specific features. Since we are concerned only with control flow in this paper the only variable of interest here is the program counter. In general, however, the register and memory variables should also be declared, along with a symbol-table relation for associating HLL variables with assembler-level memory locations.

Law 1 (Introduce assembler variables) *If variable pc is fresh, and i and f are location-valued constants such that $i < f$, then*

$$S \sqsubseteq \text{var } pc \bullet \\ pc := i ; \\ i \dots f - 1 : S ; [pc = f] .$$

□

Here the program counter is set to some initial value i , and f is its final one [11]. Thus the assembler code generated for statement S will reside in locations i up to $f - 1$, inclusive. Since S did not previously refer to the program counter, we have used a coercion [12, §17.3] to introduce the new requirement that S leaves pc equal to f . All subsequent refinement laws follow this template of explicitly stating the initial and final program counter values [11].

5.2 Compiling assignment

An assignment statement needs to firstly evaluate the expression E , and then store the result at the address associated with the target variable v . Let register x be one that is currently ‘unallocated’; in a full compilation strategy this variable would need to be formally declared and its status maintained. Similarly, let $addr$ be a symbol-table lookup function that returns the memory location associated with each HLL variable; in a full compilation strategy this relation would be extended on entering a scope and retracted on exit.

Law 2 (Compile assignment) *If $i < f - 1$, then*

$$\begin{array}{l} \{pc = i\} ; v := E ; [pc = f] \\ \sqsubseteq \quad i \dots f - 2 : \text{eval } reg_x, E ; [pc = f - 1] ; \\ \quad f - 1 : \text{store } addr(v), reg_x . \end{array}$$

□

By definition (Figure 3) the final **store** instruction increments the program counter, so placing it at location $f - 1$ ensures that the coercion to leave pc equal to f will be satisfied. Similarly, the coercion that constrains the **eval** instruction to leave the program counter equal to $f - 1$ ensures that the **store** instruction will be executed immediately after E has been evaluated.

As mentioned in Section 3.3, we do not know how many actual assembler instructions will be required to implement our temporary **eval** instruction, so we reserve a range of locations (from i to $f - 2$) for it. In a complete compilation strategy this pseudo-instruction would typically be further refined to a number of **load** instructions to fetch the operands into registers, followed by arithmetic and logical operations that leave the final result in register x . (If the range of locations is too small for the number of instructions required, then refinement of **eval** to primitive assembler instructions will be impossible. This is why it is best to avoid allocating *absolute* labels until refinement is complete.)

5.3 Compiling sequential composition

Compiling sequential composition of two HLL statements merely involves allocating consecutive memory blocks for their respective assembler instruction implementations.

Law 3 (Compile sequence) *If $i < m$ and $m < f$, then*

$$\begin{aligned} & \{pc = i\} ; (S_1 ; S_2) ; [pc = f] \\ \sqsubseteq & \quad i \dots m - 1 : S_1 ; [pc = m] \circ \\ & \quad m \dots f - 1 : S_2 ; [pc = f] . \end{aligned}$$

□

The two statements are essentially unchanged, but are augmented with coercions requiring them to update the program counter in such a way that they will be executed in the correct sequence.

5.4 Compiling choice

Implementing a conditional statement involves first evaluating the boolean expression and then branching to the appropriate alternative. Care must be taken to ensure that when the chosen alternative finishes the construct exits correctly.

Law 4 (Compile choice) *If $i < j - 1$ and $j < k - 1$ and $k < f$, then*

$$\begin{aligned} & \{pc = i\} ; \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end if} ; [pc = f] \\ \sqsubseteq & \quad i \dots j - 2 : \text{eval } reg_x, B ; [pc = j - 1] \circ \\ & \quad \quad j - 1 : \text{brfalse } reg_x, k \circ \\ & \quad j \dots k - 2 : S_1 ; [pc = k - 1] \circ \\ & \quad \quad k - 1 : \text{jump } f \circ \\ & \quad k \dots f - 1 : S_2 ; [pc = f] . \end{aligned}$$

□

Here the code to evaluate B has been placed at location i , that to implement S_1 at location j , and that for S_2 at location k . The **brfalse** instruction changes control to S_2 if B evaluates to false, otherwise S_1 executes. The **jump** instruction after S_1 exits the whole construct after S_1 terminates.

5.5 Compiling iteration

Compiling iterative statements involves generating instructions to repeatedly evaluate the boolean expression, and execute the statement if the expression is true, otherwise exit the construct. The particular compilation strategy used below places the code to evaluate the expression *after* the statement. This strategy generally produces more efficient code, saving one **jump** instruction on all iterations after the first.

Law 5 (Compile iteration) *If $i < j - 1$ and $j < f - 1$, then*

$$\begin{aligned} & \{pc = i\} ; \text{while } B \text{ loop } S \text{ end loop} ; [pc = f] \\ \sqsubseteq & \quad i : \text{jump } j \circ \\ & \quad i + 1 \dots j - 1 : S ; [pc = j] \circ \\ & \quad \quad j \dots f - 2 : \text{eval } reg_x, B ; [pc = f - 1] \circ \\ & \quad \quad f - 1 : \text{brtrue } reg_x, i + 1 \end{aligned}$$

□

Here the code to evaluate B is placed at location j , and that for statement S at location $i + 1$. Initially the **jump** changes control to evaluate the expression for the first time. If B is true the **brtrue** instruction then changes control to execute S . Once S finishes the expression is re-evaluated.

6 Example

As a simple example, we consider a code fragment that calculates the remainder of integer division by repeated subtraction. Let p be the dividend, q the divisor and r the required remainder. The high-level language program is as follows.

```

{p ≥ 0 ∧ q > 0};
r := p;
while r ≥ q loop
    r := r - q
end loop
    
```

The assumption on the first line documents knowledge about the initial values of p and q . It is not used in the following ‘compilation’ [12, p. 15].

Firstly we introduce assembler-level constructs to the executable statements.

```

⊆ ‘by Law 1’
var pc •
pc := a;
a .. d - 1 : (r := p;
               while r ≥ q loop
                   r := r - q
               end loop); [pc = d]
    
```

Symbolic constants a and d represent the initial and final program counter values. Thus this block of code has been allocated to instruction memory locations $a \dots d - 1$. The proviso on Law 1 introduces the constraint that $a < d$.

Next the sequential composition operator can be compiled, introducing a new location b , with provisos $a < b$ and $b < d$.

```

statement labelled a .. d - 1
⊆ ‘by introducing {pc = a} [12, p. 183] and Law 3’
a .. b - 1 : r := p; [pc = b];
b .. d - 1 : (while r ≥ q loop
               r := r - q
           end loop); [pc = d]
    
```

The first assignment is then readily compiled to instructions in our target assembler language, introducing proviso $a < b - 1$.

```

statement labelled a .. b - 1
⊆ ‘by Property 5 and Law 2’
a .. b - 2 : eval reg1, p; [pc = b - 1];
b - 1 : store addr(r), reg1
    
```

(Clearly the **eval** instruction could be trivially compiled to ‘**load** reg₁, addr(p)’ in this case.)

Next the loop can be compiled, with the boolean test placed at location c , as long as $b < c - 1$ and $c < d - 1$.

```

statement labelled b .. d - 1
⊆ ‘by Property 5 and Law 5’
b : jump c;
b + 1 .. c - 1 : r := r - q; [pc = c];
c .. d - 2 : eval reg2, r ≥ q; [pc = d - 1];
d - 1 : brtrue reg2, b + 1
    
```

The assignment statement within the loop is then easily compiled, provided $b < c - 2$.

```

statement labelled b + 1 .. c - 1
⊆ ‘by Property 5 and Law 2’
b + 1 .. c - 2 : eval reg3, r - q; [pc = c - 1];
c - 1 : store addr(r), reg3
    
```

(In this case the ultimate implementation of `eval` needs to ensure that the values of both r and q are in registers, and perform the subtraction, leaving the result in register 3.)

Putting these steps together yields the final assembler program, and we can instantiate the symbolic instruction memory location constants with particular values. For the purposes of illustration, assume each `eval` ‘instruction’ occupies exactly one location—let ‘`eval`’ denote an `eval` statement that adds one to the program counter. Combining the provisos accumulated above requires that $a < b - 1$, $b < c - 2$ and $c < d - 1$. Then letting $a = 1$, $b = 3$, $c = 6$ and $d = 8$, and flattening the nested labels, yields the final ‘compiled’ code.

```

⊢ ‘by Property 3’
  1 : eval  $reg_1, p$  ;
  2 : store  $addr(r), reg_1$  ;
  3 : jump 6 ;
  4 : eval  $reg_3, r - q$  ;
  5 : store  $addr(r), reg_3$  ;
  6 : eval  $reg_2, r \geq q$  ;
  7 : brtrue  $reg_2, 4$ 
    
```

7 Future work

An obvious extension to this work is to consider other high-level language statements and constructs. For instance, it is trivial to devise a compilation law for an `if` statement with no `else` part, using Law 4 as a guide. More challenging, though, are constructs such as subroutine calls, and concurrency and communication statements.

Given this paper’s focus on control flow only, we noted above several informal aspects of our model that need fleshing out. The assembler-level register and memory variables, and the compiler-level symbol table relation, should be formally declared and their status maintained as part of the refinement rules. Keeping track of these variables would support many useful optimisations. For instance, we can easily envisage a variant of Law 2 that uses the knowledge that the required expression value is already available in some register x .

Law 6 (Compile assignment of known value)

```

    { $pc = i \wedge reg_x = E$ } ;  $v := E$  ; [ $pc = i + 1$ ]
⊢  $i$  : store  $addr(v), reg_x$ 
    
```

□

(This also overcomes another weakness of the laws above in that they require every statement to occupy at least one memory location—statements logically equivalent to skip currently compile to unnecessary `noop` instructions.) Similarly, Law 2 is rather naive in that it *always* stores the result in memory. Instructions 2 and 5 in the example in Section 6 could potentially be eliminated, and replaced with a single `store` instruction at the end of the code fragment, if we knew *when* HLL variables must be observably updated in memory.

Our overall goal is to develop trustworthy compilation techniques for *real-time* programs. Elsewhere we have proposed a real-time refinement formalism which makes ‘time’ an integral part of the refinement laws [6, 17, 7]. Using this real-time calculus instead of the standard refinement calculus will allow us to consider compilation of time-sensitive HLL statements such as ‘clock’ functions and `delay` statements.

8 Conclusion

We have shown how compilation of high-level language programs to assembler code can be modelled in the standard refinement calculus. To achieve this we introduced two new operators for composing assembler programs from individual instructions. The innovation of allowing *any* statement to be labelled with a *set* of locations enabled an elegant compilation-as-refinement methodology.

Acknowledgements I wish to thank Karl Lerner and Ian Hayes for identifying errors in this work, and Peter Kearney and the anonymous referees for reviewing drafts of this paper. This project is funded by Australian Research Council grant A49600176, *Verified compilation rules for real-time programs via program refinement*.

References

- [1] R.-J. R. Back. Refinement of parallel and reactive programs. Technical Report Caltech-CS-TR-92-23, California Institute of Technology, 1992.
- [2] E. Börger and I. Durdanović. Correctness of compiling occam to transputer code. *The Computer Journal*, 39(1):52–92, 1996.
- [3] J. Bowen, editor. *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems*. Elsevier, 1994.
- [4] M. Fränzle and M. Müller-Olm. Towards provably correct code generation for a hard real-time programming language. In P. Fritzson, editor, *Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 294–308. Springer-Verlag, 1994.
- [5] R. W. S. Hale. Program compilation. In J. Bowen, editor, *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems*, chapter 7, pages 131–146. Elsevier, 1994.
- [6] I. J. Hayes and M. Utting. Coercing real-time refinement: A transmitter. In D. J. Duke and A. S. Evans, editors, *BCS-FACS Northern Formal Methods Workshop, 1996*, Electronic Workshops in Computing. Springer-Verlag, 1997. <http://www.springer.co.uk/ewic/workshops/NFM96/>.
- [7] I. J. Hayes and M. Utting. Sequential real-time refinement. School of Information Technology, University of Queensland, January 1997.
- [8] He Jifeng. *Provably Correct Systems*. McGraw-Hill, 1995.
- [9] 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.
- [10] C. A. R. Hoare and He Jifeng. Refinement algebra proves correctness of a compiler. Draft, June 1990.
- [11] K. Lerner and C. J. Fidge. Compilation as refinement. In L. Groves and S. Reeves, editors, *Formal Methods Pacific '97*, pages 142–164. Springer, 1997.
- [12] C. Morgan. *Programming from Specifications*. Prentice-Hall, second edition, 1994.
- [13] C. Morgan. The specification statement. In C. Morgan and T. Vickers, editors, *On the Refinement Calculus*, pages 1–21. Springer-Verlag, 1994.
- [14] C. Morgan and T. Vickers. *On the Refinement Calculus*. Springer-Verlag, 1994.
- [15] T. S. Norvell. Machine code programs are predicates too. In D. Till, editor, *Sixth Refinement Workshop*, pages 188–204. Springer-Verlag, 1994.
- [16] T. S. Norvell. *A Predicative Theory of Machine Languages and its Application to Compiler Correctness*. PhD thesis, Graduate Department of Computer Science, University of Toronto, 1994.
- [17] M. Utting and C. J. Fidge. Refinement of infeasible real-time programs. In L. Groves and S. Reeves, editors, *Formal Methods Pacific '97*, pages 243–262. Springer, 1997.