Machine Code Programs are Predicates Too

Theodore S. Norvell
norvell@cs.toronto.edu
norvell@comlab.ox.ac.uk

Abstract

I present an interpretation of machine language programs as boolean expressions. Source language programs may also be so interpreted. The correctness of a code generator can then be expressed as a simple relationship between boolean expressions. Code generators can then be calculated from their specification.

1 Introduction

A predicate divides its domain in two. In the specification of computational behaviour we wish to divide the set of all imaginable computations in two parts: acceptable computations and unacceptable computations. Predicates provide a convenient way of expressing this division.

In this paper we are interested in two kinds of computational behaviour: the behaviour specified by high level language programs in terms of source level variables, and the behaviour specified by machine language programs in terms of registers and memory. By using predicate logic as a common framework to describe both forms of behaviour we can relate them and write down the logical relationship that should exist between the input and output programs of a compiler. This relationship serves as a specification from which we can derive a code generator.

Except for the derivation of an example code generator, all proofs and derivations have been omitted. Where proofs are not straight forward, an outline is presented. Proofs of most of the theorems can be found in my thesis [Norvell 1993].

2 Motivation

The problem of compiler correctness is clearly an important one for computer reliability. The point of proving a high-level language program correct is diminished if that program is submitted to a compiler that is not correct. It can be expected that a great many source programs will be submitted to a single compiler. As the correct execution of all of them depends —potentially— on the compiler, proving the compiler can be highly worthwhile. Furthermore testing compilers is notoriously difficult, involving, as it does, either reasoning about the machine code output or testing the object program; the former is quite difficult and the latter is indirect and likely to fail to find all bugs.

6th Refinement Workshop, David Till (Ed.), pp. 188–204, Springer-Verlag, 1994.

Aside from this practical consideration, the problem is of interest to those interested in the formalization of reasoning about programs. It involves the interaction of two formal descriptions of language, one for the source language, and one for the target language.

Quite a lot of previous research has been done on proving compilers and on automatically producing compilers from formal descriptions of languages, for example, [McCarthy and Painter 1967, Morris 1973, Milne and Strachey 1974, Thatcher *et al.* 1980, Mosses 1980, Polak 1981, Manasse and Nelson 1984, Hoare 1990, Sampaio 1993]. This research has employed almost the gamut of approaches to formalizing reasoning about programs: operational, algebraic, denotational.

The research presented below uses yet another approach —the predicative. Reasoning about specification (including programs) represented by predicates has proved to be a very effective way to develop correct programs. The question is whether the predicative approach will yield a promising line of attack on compiler correctness.

In addition the compiler correctness problem serves as an interesting case study in the use of the predicative approach. This is another motivation.

As a first step towards compiler correctness, we will look at using predicates to model machine programs. Quite apart from its application to compiler correctness this aspect of the work has applications all its own. For example, the resulting description of the CPU architecture could be used as a departure point for developing a proved implementation of the CPU in hardware.

3 Notation

Among the numbers, we will be using mostly the integers, but sometimes the naturals and the extended naturals. The extended naturals are the naturals together with a single infinity value ∞ which is larger than all naturals.

The number operators $(+, -, =, -, <, \leq, >, \geq)$ have their usual meanings.

A *string* is a sequence of items. The length of a string s is written $\#s$. The string of length zero is written as nil. A string of length one is equal to its sole element. The catenation of two strings s and t is written $s;t$.

The boolean operators $(\equiv, \not\equiv, \Rightarrow, \Leftarrow, \wedge, \vee, \neg)$ also have the usual definitions. The expression $x \ \langle\!|b|\!\rangle \ y$ for boolean b has value x when b is true and y when b is false.

Function application is written $f.x$. The function $\langle \lambda f \cdot \langle \lambda z \cdot y \ \langle\!|z = x|\!\rangle \ f.z \rangle\rangle$ which mutates its functional argument is written $x \mapsto y$.

Other notations and terms will be introduced and explained as needed; but for future reference, the precedence of all the operators is shown in Table 1.

Variables will generally follow the following typing conventions. When they do, I will feel free to omit mention of their types.

i	instructions
s, t, u	strings of instructions
n	natural numbers nat
j, k, l	integers int
P, Q, R	specifications
C, D	conditions

6th Refinement Workshop, David Till (Ed.), pp. 188–204, Springer-Verlag, 1994.

$.$	left associative
$\#\quad \heartsuit\quad \clubsuit\quad \downarrow\quad \uparrow$	
$+\quad -$	left associative
$:=\quad \mapsto$	
$;$	associative
$@\quad !$	
$\circ$	associative
$\langle\!\vert b \vert\!\rangle$	right associative
$=\quad \neq\quad <\quad \leq\quad >\quad \geq$	
$\neg$	
$\wedge\quad \vee$	lifting
$\equiv\quad \not\equiv\quad \Rightarrow\quad \Leftarrow$	lifting; $\Rightarrow$ is right associative; $\Leftarrow$ is left associative; $\equiv$ and $\not\equiv$ are associative
$\cdot\!:\!\cdot\quad \cdot\!:\quad :\!\cdot$	

Table 1: Precedence of operators

S, T, U	source level specifications
V, W, X	machine level specifications
f, g, h	functions

4 Specifications

State spaces are sets of states. We will use two state spaces Σ and M representing the source and machine level states respectively.

We will be particularly interested in functions of type

$$(\Sigma \times \Sigma \times \mathrm{M} \times \mathrm{M}) \rightarrow \mathit{bool}$$

which will be called *specifications*. For convenience, specifications will be written as boolean expressions with the variables (σ and σ' of type Σ, and μ and μ' of type M) representing their four parameters respectively. Accordingly the substitution notation is used for applying or partially applying specifications. For example, P^{σ}_{x} is the application of P to x at parameter σ. As a special case $P^{\sigma;\mu}_{\sigma';\mu'}$ is notated by P'.

A specification is called *machine level* if its value does not depend on the value of the σ and σ' parameters; and is called *source level* if its value does not depend on the value of the μ and μ' variables.

The boolean operators and constants are lifted pointwise to specification operators and constants. Specifications may be compared with the operator

$$(P \;\cdot\!:\; Q) \quad \equiv \quad \langle \forall \sigma; \mu; \sigma'; \mu' \cdot P \Leftarrow Q \rangle$$

In words, $P \;\cdot\!:\; Q$ is written P *is refined by* Q. We write $P \;:\!\cdot\; Q$ for $Q \;\cdot\!:\; P$ and $P \;\cdot\!:\!\cdot\; Q$ for $(P \;\cdot\!:\; Q) \wedge (P \;:\!\cdot\; Q)$, which is equality of specifications. We define the operator $\circ$ on specifications as

$$P \circ Q \quad \cdot\!:\!\cdot \quad \langle \exists \sigma''; \mu'' \cdot P^{\sigma';\mu'}_{\sigma'';\mu''} \;\wedge\; Q^{\sigma;\mu}_{\sigma'';\mu''} \rangle$$

6th Refinement Workshop, David Till (Ed.), pp. 188–204, Springer-Verlag, 1994.

This has an identity $ok \cdot\!: \sigma'; \mu' = \sigma; \mu$. We will also have occasion to use $ok_{\mathrm{M}} \cdot\!: \mu' = \mu$ and $ok_{\Sigma} \cdot\!: \sigma' = \sigma$.

In this paper we are interested in *batch* computations at either the source or machine level. A batch computation consists of an initial and a final state. For source level specifications, σ represents the initial state and σ' the final state. For machine level specifications, μ represents the initial state and μ' the final state. The predicates in

$$(\Sigma \times \Sigma \times \mathrm{M} \times \mathrm{M}) \rightarrow \mathit{bool}$$

that we are calling 'specifications' represent specifications —in an informal sense— by selecting those computations which are acceptable. The comparison $P \cdot\!: Q$ thus means that Q accepts no computation that P does not accept, and thus clearly deserves reading 'P is refined by Q'. The least useful specification is **true**, and the entirely miraculous specification is **false**.

> **Remark** It should be clear from the above that —unlike the common convention in Z— specifications that allow infinite computations are represented by weak predicates rather than strong predicates.

5 Predicative semantics and source programs

A *predicative semantics* for a programming language (or a specification language) is an interpretation of the members of that language as predicates. This idea is illustrated with a source level language in this section and with a machine level language on the next.

5.1 Abstract syntax

Our example source language is the very simple one shown in Table 2, in which the syntactic variables are used as follows: S, T, and U range over statements; E, F, and G range over expressions; v ranges over source variables; c ranges over constants (**true**, **false**, 0, -1, 1, ...); *bop* ranges over binary operators ($+$, $-$, $=$, **and**, and **or**); *uop* ranges over unary operators (**not** and $-$); and *ty* ranges over type constants (**int**, and **bool**).

The state space Σ for this language is the cross product of the types of the source variables. We denote the projections of σ and σ' with the unprimed and primed names of the source variables. Thus for a source variable x we write x to represent its value in the initial state and x' to represent its value in the final state. Furthermore the substitution notation is extended so that —for example— S^x_E means $S^{\sigma}_{f.\sigma}$ where f is a function that leaves every component of a state alone, except for x which it sets to E.

> **Extra explanation** This sounds rather complex, but it just means that we can write a specification S that specifies the final value of x is to be 1 more than its initial value as $x' = x + 1$, and that the specification that says the final value of x is to be three can be written as either $x' = 3$ or, equivalently, S^x_2.

6th Refinement Workshop, David Till (Ed.), pp. 188–204, Springer-Verlag, 1994.

$$
\begin{array}{lcl}
S & \rightarrow & \mathsf{nil} \\
 & | & T;U \\
 & | & v := E \\
 & | & \textbf{if } E \textbf{ then } T \textbf{ else } U \textbf{ fi} \\
 & | & \textbf{while } E \textbf{ do } T \textbf{ od} \\
E & \rightarrow & v \\
 & | & c \\
 & | & F \ bop \ G \\
 & | & uop \ F
\end{array}
$$

Table 2: The abstract syntax of the source language.

nil	ok_Σ
$T;U$	$T \circ U$
$v := E$	$(ok_\Sigma)^v_E$
if E **then** T **else** U **fi**	$T \ \langle\!\lvert E \rvert\!\rangle \ U$
while E **do** T **od**	see text

Table 3: The semantics of the source language.

One global source variable is always t_σ of type *xnat* representing time; t_σ represents the time at which a computation starts and t'_σ represents the time at which the computation ends —at least in an abstract sense. It is assumed there are no explicit references to t_σ in any program.

5.2 Semantics

We interpret each abstract syntax term as a specification according to table 3. For example if the global source variables are x, y, z, t_σ,

$$(x := 1; y := y + x) \quad \cdot\!:\!\cdot \quad x' = 1 \ \wedge \ y' = y + 1 \ \wedge \ z' = z \ \wedge \ \mathsf{t}'_\sigma = \mathsf{t}_\sigma$$

while E **do** T **od** is defined using a method called the *weakest progressive pre-fixedpoint*. Define the function wh from specifications to specifications by

$$\mathsf{wh}.U \quad \cdot\!:\!\cdot \quad (T \circ (\mathsf{t}_\sigma := \mathsf{t}_\sigma + 1) \circ U) \ \langle\!\lvert E \rvert\!\rangle \ ok_\Sigma$$

Now we postulate that S, such that $S \ \cdot\!:\!\cdot$ **while** E **do** T **od**, has the following three properties

$$
\begin{array}{l}
\mathsf{t}'_\sigma \geq \mathsf{t}_\sigma \quad :\!: \quad S \\
\mathsf{wh}.S \quad :\!: \quad S \\
\langle \forall U \cdot (\mathsf{t}'_\sigma \geq \mathsf{t}_\sigma \quad :\!: \quad U) \wedge (\mathsf{wh}.U \quad :\!: \quad U) \Rightarrow (S \quad :\!: \quad U) \rangle
\end{array} \qquad (1)
$$

For example, it can be shown from this that **while true do nil od** $\cdot\!:\!\cdot \ \mathsf{t}'_\sigma = \infty$.

Further reading on predicative semantics may be found in [Hehner 1984], [Hehner 1993], [Hoare 1992], and [Norvell 1993].

6th Refinement Workshop, David Till (Ed.), pp. 188–204, Springer-Verlag, 1994.

Remark Although predicative semantics is used in this paper to represent specifications of batch computations, it should not be inferred that it is limited to batch computations. One of the strengths of the approach is its ability to handle interactive computations.

6 A machine language and its semantics

In this section I present the semantics of a simple machine language. It should be understood that this is to present a general method of defining machine languages. All the theorems of the paper which do not mention specific instructions hold true for any machine language that meets the conditions spelled out in Section 6.2.

The machine language semantics is presented in two parts: machine dependent and machine independent. The machine dependent part defines the structure of the machine level state space M, the instruction set, and axioms defining the semantics of individual instructions. The machine independent part consists of additional axioms that define how individual instructions act together. Theorems based on the machine independent axioms are reusable for all machine languages.

6.1 Machine dependent aspects

The example machine for this paper has a state space M of four components

- $p : int$ The program counter.
- $\mathbf{t}_\mu : xnat$ The time. The type $xnat$ includes all natural numbers and a special ∞ value larger than all natural numbers.
- $a : int$ The accumulator.
- $m : int \to int$ The memory.

Thus M is the product $(int \times xnat \times int \times (int \to int))$. The four projections of μ are written p, $\mathbf{t}_\mu$, a, and m; and the four projections of μ' as p', $\mathbf{t}'_\mu$, a', and m'.

Each instruction in the instruction set is a string of length one and is interpreted as a specification according to the axioms in Table 4. As a convenience, we write $x :=_{\mathrm{M}} E$ for $(ok_M)^x_E$ with x any string of component names and E an equal length string of expressions. Note that only backward and self jumps are considered to take any time.

6.2 Assumptions about machine dependent aspects

We will make four assumptions about the machine dependent axioms. These serve as the interface between the machine independent axioms and the machine dependent ones. Each should be demonstrated about the machine dependent axioms in order to ensure the applicability of the rest of the theory.

Axioms For any integer j,

$$
\begin{array}{rcl}
\mathsf{enter}\ j & \cdot{:}\cdot & p; a :=_{\mathrm{M}} p+1; j \\
\mathsf{load}\ j & \cdot{:}\cdot & p; a :=_{\mathrm{M}} p+1; m.j \\
\mathsf{add}\ j & \cdot{:}\cdot & p; a :=_{\mathrm{M}} p+1; a+m.j \\
\mathsf{store}\ j & \cdot{:}\cdot & p; m :=_{\mathrm{M}} p+1; (j \mapsto a).m \\
\mathsf{jump}\ j & \cdot{:}\cdot & p; \mathsf{t}_\mu :=_{\mathrm{M}} p+j; (\mathsf{t}_\mu + 1 \ \langle\!|\, j \le 0 \,|\!\rangle\ \mathsf{t}_\mu) \\
\mathsf{zjump}\ j & \cdot{:}\cdot & p; \mathsf{t}_\mu :=_{\mathrm{M}} p + (j \ \langle\!|\, a = 0 \,|\!\rangle\ 1); (\mathsf{t}_\mu + 1 \ \langle\!|\, a = 0 \wedge j \le 0 \,|\!\rangle\ \mathsf{t}_\mu)
\end{array}
$$

Table 4: Axioms for instructions

Assumption M should possess a component p of a subtype of int and a component t_μ of type $xnat$.

Assumption Machine Level. Each instruction i should be a machine level specification.

Assumption Implementability. For all instructions i

$$\langle \forall \mu \cdot \langle \exists \mu' \cdot i \rangle \rangle$$

Assumption Progress. For all instructions i

$$\mathsf{t}'_\mu \ge \mathsf{t}_\mu \quad \cdot{:}\quad i$$

All theorems in the remainder of Section 6 depend only on these assumptions and machine independent axioms. None directly rely on any specifics of the example machine presented.

6.3 Machine independent aspects

The machine independent axioms define the operators '!' and '@', which both take a string of instructions and an integer and produce a specification.

6.3.1 The ! operator.

The specification $s \,!\, l$ specifies the acceptable behaviour of a computer loaded with instruction string s beginning at location l in the program memory over the execution of a single instruction. It is defined by the

6th Refinement Workshop, David Till (Ed.), pp. 188–204, Springer-Verlag, 1994.

Axioms

$$\begin{aligned} \mathsf{nil} \,!\, l &\;\cdot\!:\cdot\; true \\ i \,!\, l &\;\cdot\!:\cdot\; (p = l) \;\Rightarrow\; i \\ s;t \,!\, l &\;\cdot\!:\cdot\; (s \,!\, l) \wedge (t \,!\, l + \#s) \end{aligned}$$

The following theorems follow from these axioms and the assumptions.

Theorem Noninterference.

$$(l \leq p < l + \#s) \;\vee\; (s \,!\, l)$$

Theorem Implementability.

$$\langle \forall \mu \cdot \langle \exists \mu' \cdot s \,!\, l \rangle \rangle$$

Theorem Progress.

$$\mathsf{t}'_\mu \geq \mathsf{t}_\mu \quad \cdot\!: \quad (l \leq p < l + \#s) \;\wedge\; (s \,!\, l)$$

6.3.2 The @ operator.

The specification $s \mathbin{@} l$ describes the acceptable behaviour of a computer loaded with string s beginning at location l over the time the instruction counter points to instructions in s.

Definition For each s and each l we define the following function iter from specifications to specifications

$$\mathsf{iter}.V \quad \cdot\!:\cdot \quad (s \,!\, l \circ V)\ \langle\!| l \leq p < l + \#s |\!\rangle\ ok_{\mathrm{M}}$$

A specification is a *pre-fixedpoint of* iter just if

$$\mathsf{iter}.V \quad \cdot\!: \quad V$$

A specification is a *fixedpoint of* iter just if

$$\mathsf{iter}.V \quad \cdot\!:\cdot \quad V$$

We define $s \mathbin{@} l$ with the following axioms:

Axiom Construction. $s \mathbin{@} l$ is a pre-fixedpoint of iter.

$$\mathsf{iter}.(s \mathbin{@} l) \quad \cdot\!: \quad s \mathbin{@} l$$

6th Refinement Workshop, David Till (Ed.), pp. 188–204, Springer-Verlag, 1994.

Axiom Progression. $s \mathbin{@} l$ is progressive.

$$\mathsf{t}'_\mu \geq \mathsf{t}_\mu \ \cdot\!: \ s \mathbin{@} l$$

Axiom Induction. $s \mathbin{@} l$ is as weak as any progressive pre-fixedpoint of iter.

$$\langle \forall V \cdot (\mathsf{iter}.V \quad \cdot\!: \quad V) \ \wedge \ (\mathsf{t}'_\mu \geq \mathsf{t}_\mu \ \cdot\!: \ V) \ \Rightarrow \ (s \mathbin{@} l \ \cdot\!: \ V) \rangle$$

These axioms are consistent by virtue of the Knaster-Tarski theorem [Tarski 1955].

The weakest progressive pre-fixedpoint of iter is also a fixedpoint. Indeed it is the weakest progressive fixedpoint.

Theorem Fixedpoint. $s \mathbin{@} l$ is a fixedpoint

$$\mathsf{iter}.(s \mathbin{@} l) \quad \cdot\!:\!\cdot \quad s \mathbin{@} l$$

Theorem Weakest progressive fixedpoint.

$$\langle \forall V \cdot (\mathsf{iter}.V \quad \cdot\!: \quad V) \ \wedge \ (\mathsf{t}'_\mu \geq \mathsf{t}_\mu \ \cdot\!: \ V) \ \Rightarrow \ (s \mathbin{@} l \ \cdot\!: \ V) \rangle$$

The following two theorems give a slightly more operational perspective.

Theorem Okness.

$$s \mathbin{@} l \ \equiv \ ok_{\mathrm{M}} \quad \cdot\!: \quad (p < l) \ \vee \ (p \geq l + \#s)$$

Theorem Single stepping.

$$i \circ (s; i; t \mathbin{@} l) \ \equiv \ s; i; t \mathbin{@} l \quad \cdot\!: \quad p = l + \#s$$

The specifications we get are reasonable in the sense of being implementable.

Theorem Implementability. $\langle \forall \mu \cdot \langle \exists \mu' \cdot s \mathbin{@} l \rangle \rangle$.

Proof sketch. If we can show there is one implementable and progressive pre-fixedpoint of iter, then by the induction axiom, $s \mathbin{@} l$ must also be implementable.

From the construction and progression axioms we know there is at least one progressive pre-fixedpoint, implementable or not, namely $s \mathbin{@} l$. Let R be some progressive pre-fixedpoint. Construct Q as

$$Q \quad \cdot\!:\!\cdot \quad R \ \vee \ (\neg \langle \exists \mu' \cdot R \rangle \ \wedge \ \mathsf{t}'_\mu = \infty)$$

It is trivial that Q is progressive and implementable and not hard to show it is a pre-fixedpoint. □

6th Refinement Workshop, David Till (Ed.), pp. 188–204, Springer-Verlag, 1994.

As long as a computer is executing the code in a certain region of its program memory, the contents of the rest of the program memory may be disregarded.

This is a crucial separation of concerns. It will be particularly important when we are deriving code generators; it will mean that the object code can be constructed by the code generator bit by bit rather than all at one go.

We can capture the idea formally with the following theorem.

Theorem Separation.

$$s; t; u \mathbin{@} l \quad \cdot\!:\!\cdot \quad t \mathbin{@} l + \#s \circ s; t; u \mathbin{@} l$$

Proof sketch [Cook 1993]. Consider a graph in which nodes are states in $\Sigma \times \mathrm{M}$ and there is an edge from μ to μ' just when $s \mathbin{!} l$. Then $s \mathbin{@} l$ is satisfied by μ and μ' just when either there is a finite path from μ to μ' on which μ' is the only state in which $\neg(l \leq p < l + \#s)$, or $\mathbf{t}'_\mu = \infty$ and there is an infinite path from μ on which for all states $l \leq p < l + \#s$.

Suppose μ and μ' satisfy $s; t; u \mathbin{@} l$, there is a suitable path in the graph for $s; t; u \mathbin{@} l$, this path can be divided at the first node where $\neg(l + \#s \leq p < l + \#s + \#t)$ to get paths that show the right hand side is satisfied. Likewise if the right hand side is satisfied, that gives two paths that can be catenated to show the left hand side is satisfied. □

7 Coupling source and machine levels

In order to use a machine to simulate source level computations, we will need an example correspondence between source and machine states. This can be represented by a specification R dependent on only σ and μ.

An example of such a specification is given as follows. Let n be the number of components, aside from $\mathbf{t}_\sigma$, comprising Σ, and let v_j, for $0 \leq j < n$, refer to component j. Define two one-one functions from integers

$$\begin{aligned}
&\mathsf{abs}_{int} : int \to int \\
&\mathsf{abs}_{int}.i = i \\
&\mathsf{abs}_{bool} : int \to bool \\
&\mathsf{abs}_{bool}.0 = \mathbf{false} \\
&\mathsf{abs}_{bool}.i = \mathbf{true} \quad \text{for } i \neq 0
\end{aligned}$$

and a one-one function from $0, ..., n-1$ to memory addresses

$$\mathsf{addr} : 0, ..., n-1 \to int$$

The R predicate is then

$$(\mathbf{t}_\sigma = \mathbf{t}_\mu) \wedge \langle \forall j : 0, ..., n-1 \cdot v_j = \mathsf{abs}.(m.(\mathsf{addr}.j)) \rangle$$

8 Code generator specification

Consider the following "thought experiment". We wish to simulate the behaviour of a source program S starting in a state σ. We initialize a machine

to a state μ that is related to σ by R and has $p = l$, and then run program $s \mathbin{@} l$. We check the final state μ' and consider all high-level states σ' that correspond to it. If they all could be reached by running S, we say that $s \mathbin{@} l$ has simulated S. If for all l and all initial states σ, the specification $s \mathbin{@} l$ must simulate the high level program S, then s is a suitable translation of S for the given representation relation R.

We can easily formalize the thought experiment as the following definition

Definition We say that s *simulates* S, in notation $\mathsf{sim}.s.S$, just if

$$\langle \forall l; \sigma; \mu \cdot \mathsf{R} \wedge p = l \Rightarrow \langle \forall \mu' \cdot s \mathbin{@} l \Rightarrow \langle \forall \sigma' \cdot \mathsf{R}' \Rightarrow S \rangle \rangle \rangle$$

In order to express this as refinement of either machine or source level specifications, we define two operators parameterized by R.

$$\begin{array}{lcl} P{\downarrow}S & \cdot{:}\cdot & \langle \forall \sigma; \sigma' \cdot \mathsf{R} \wedge \mathsf{R}' \wedge P \Rightarrow S \rangle \\ P{\uparrow}V & \cdot{:}\cdot & \langle \exists \mu; \mu' \cdot \mathsf{R} \wedge \mathsf{R}' \wedge P \wedge V \rangle \end{array}$$

When P is omitted, it defaults to **true**. Note that, for any P, $(P{\uparrow})$ and $(P{\downarrow})$ are functions related by the Galois property

$$(P{\downarrow}S \;\; \cdot{:} \;\; V) \equiv (S \;\; \cdot{:} \;\; P{\uparrow}V) \tag{2}$$

The definition of $\mathsf{sim}.s.S$ can be rewritten as any of

$$\begin{array}{l} \langle \forall l \cdot (p = l){\downarrow}S \;\; \cdot{:} \;\; s \mathbin{@} l \rangle \\ \langle \forall l \cdot S \;\; \cdot{:} \;\; (p = l){\uparrow}(s \mathbin{@} l) \rangle \\ {\downarrow}S \;\; \cdot{:} \;\; s \mathbin{@} p \end{array} \tag{3}$$

It is reasonable —on the grounds of the thought experiment— to insist on the following restrictions on R.

$$\begin{array}{l} \langle \forall l \cdot \langle \forall \sigma \cdot \langle \exists \mu \cdot \mathsf{R} \wedge p = l \rangle \rangle \rangle \\ \langle \forall \mu \cdot \langle \exists! \sigma \cdot \mathsf{R} \rangle \rangle \\ \mathsf{t}_\sigma = \mathsf{t}_\mu \;\; \cdot{:} \;\; \mathsf{R} \end{array}$$

It can easily be seen that the example R defined in Section 7 meets these three requirements.

9 Prelude to code generator derivation

In the next section I will sketch the derivation of a code generator using the example specifications of the machine language, source language, and representation relation described above. The goal is to derive a function C such that for all abstract syntax terms S

$$\mathsf{sim}.(\mathsf{C}.S).S$$

Before doing so we need a number of results relating some of the concepts defined above.

6th Refinement Workshop, David Till (Ed.), pp. 188–204, Springer-Verlag, 1994.

9.1 Nice strings

A little thought shows that $\mathsf{sim}.t.T \wedge \mathsf{sim}.u.U$ does not —as one might hope— imply $\mathsf{sim}.(t;u).(T;U)$ The difficulty is that t need not leave the program counter pointing to the first instruction of u.

It is not possible to simply add the requirement that

$$p' = p + \#s \quad \cdot\!: \quad s \mathbin{@} p$$

for all strings s generated by our code generator, because it is quite possible that $s \mathbin{@} p$ will loop forever. If a program is executed from a state from which it may take an infinite amount of time, it can not be expected to set the program counter to a particular value. This is a consequence of our choice of the *weakest* progressive prefixed point.

So we make the following definition

Definition A string s is called *nice* just if

$$p' = p + \#s \quad \cdot\!: \quad s \mathbin{@} p \wedge \mathsf{t}'_\mu \neq \infty$$

We notate this as $\heartsuit s$.

Definition A string s will be a *correct compilation* of an abstract syntax term S just if

$$\begin{array}{l} \mathsf{sim}.s.S \\ \heartsuit s \end{array}$$

9.2 Useful facts

Because R defines a function onto the source state space, it has some useful manipulative properties.

Theorem (4) $\downarrow$ over $\circ$. For specifications P and Q

$$\downarrow(P \circ Q) \quad \cdot\!: \quad \downarrow P \circ \downarrow Q$$

Theorem (5) If P is a machine level condition,

$$\downarrow(Q \ \langle\!|P|\!\rangle \ R) \quad \cdot\!\cdot\!: \quad \downarrow Q \ \langle\!|P|\!\rangle \ \downarrow R$$

Definition A specification V is *explosive* just if

$$\langle \exists \mu' \cdot \mathsf{t}'_\mu = \infty \wedge V \rangle \quad \cdot\!: \quad \langle \forall \mu' \cdot \mathsf{t}'_\mu = \infty \Rightarrow V \rangle$$

That is, just if for all initial states for which V does not specify termination, V does not specify anything stronger than progressiveness. The notation $\clubsuit V$ will be used to say that V is explosive.

6th Refinement Workshop, David Till (Ed.), pp. 188–204, Springer-Verlag, 1994.

Theorem Source explosiveness. For all abstract syntax terms S, we have $\clubsuit{\downarrow}S$.

Clearly, if $\heartsuit s$ and $\heartsuit t$, then $\heartsuit(s;t)$. Furthermore, $\heartsuit\mathsf{nil}$ so nice strings form a submonoid of the strings. But we can also see that if $s;t$ is started at its beginning and terminates then it will execute by first executing s and then t.

Theorem (6) Nice catenation. If

- $\heartsuit s$, $\heartsuit t$, $\clubsuit V$, $\clubsuit W$,
- $V \quad \cdot\!: \quad s \mathbin{@} p$
- $W \quad \cdot\!: \quad t \mathbin{@} p$

then $V \circ W \quad \cdot\!: \quad s;t \mathbin{@} p$.

Theorem (7) Storing.

$$\downarrow((ok)^{v_m}_{\mathsf{abs}.a}) \quad \cdot\!: \quad \mathsf{store\ addr}.m \mathbin{@} p$$

Theorem (8) Jumping. If

- $\heartsuit s$, $\heartsuit t$, $\clubsuit V$, $\clubsuit W$,
- $V \quad \cdot\!: \quad s \mathbin{@} p$
- $W \quad \cdot\!: \quad t \mathbin{@} p$

then

$$V \ \langle\!|a \neq 0|\!\rangle \ W \quad \cdot\!: \quad u \mathbin{@} p$$

where $u = \mathsf{zjump}\ (2 + \#s); s; \mathsf{jump}\ (1 + \#t); t$

Notation

$$\mathsf{mtick} \ \cdot\!\cdot\!: \ a' = a \ \wedge \ m' = m \ \wedge \ t' = t + 1$$

Theorem (9) Looping. If

- $\heartsuit s$, $\heartsuit t$, $\clubsuit V$, $\clubsuit W$,
- $V \quad \cdot\!: \quad s \mathbin{@} p$
- $W \quad \cdot\!: \quad t \mathbin{@} p$

then $V \circ (W \circ \mathsf{mtick} \circ u \mathbin{@} l \ \langle\!|a \neq 0|\!\rangle \ ok_{\mathrm{M}}) \quad \cdot\!: \quad u \mathbin{@} l$ where $u = s; \mathsf{zjump}\ (2 + \#t); t; \mathsf{jump}\ (-1 - \#t - \#s)$

6th Refinement Workshop, David Till (Ed.), pp. 188–204, Springer-Verlag, 1994.

Theorem (10) The expression theorem. If f is a function from either the integers or the booleans to source level specifications, and E is an expression of the right type, then

$$f.E \quad \cdot\!: \quad (\mathsf{abs}.a' = E \ \wedge \ ok_\Sigma) \circ f.(\mathsf{abs}.a)$$

9.3 Expression correctness

We will say that a string e is a correct compilation of an expression E just if

$$\langle \forall l \cdot (p = l) \downarrow (\mathsf{abs}.a' = E \ \wedge \ ok_\Sigma) \quad \cdot\!: \quad e \ @ \ l \rangle \\ \heartsuit e$$

10 Code generator derivation

Throughout this section, we will assume as induction hypotheses that t is a correct compilation of T, u is a correct compilation of U, and e is a correct compilation of E.

The goal is to find a correct compilation of program S whatever its form.

10.1 The nil statement

We have $S = \mathsf{nil}$

$$\begin{array}{lll} & \downarrow ok_\Sigma & \{\downarrow ok_\Sigma \quad \cdot\!: \quad ok_\mathrm{M}\} \\ \cdot\!: & ok_\mathrm{M} & \\ \cdot\!:\cdot & \mathsf{nil} \ @ \ p & \end{array}$$

10.2 The sequential composition statement

We have $S = T; U$.

$$\begin{array}{lll} & \downarrow (T \circ U) & \{\downarrow \text{ over } \circ \ (4)\} \\ \cdot\!: & \downarrow T \circ \downarrow U & \{\text{nice catenation (6), induction hypotheses}\} \\ \cdot\!:\cdot & t; u \ @ \ p & \end{array}$$

So $\mathsf{C}.(S; T)$ can be $\mathsf{C}.S; \mathsf{C}.T$.

10.3 The assignment statement

We have $S = (v_m \ := \ E)$.

$$\begin{array}{lll} & \downarrow ((ok_\Sigma)^{v_m}_E) & \{\text{the expression theorem (10)}\} \\ \cdot\!: & \downarrow ((\mathsf{abs}.a' = E \ \wedge \ ok_\Sigma) \circ (ok_\Sigma)^{v_m}_{\mathsf{abs}.a}) & \{\downarrow \text{ over } \circ \ (4)\} \\ \cdot\!: & \downarrow (\mathsf{abs}.a' = E \ \wedge \ ok_\Sigma) \circ \downarrow ((ok_\Sigma)^{v_m}_{\mathsf{abs}.a}) & \\ \{\text{storing (7), nice catenation (6), induction hypotheses}\} & & \\ \cdot\!: & e; \mathsf{store \ addr}.m & \end{array}$$

6th Refinement Workshop, David Till (Ed.), pp. 188–204, Springer-Verlag, 1994.

10.4 The if statement

We have $S = \textbf{if } E \textbf{ then } T \textbf{ else } U \textbf{ fi}$.

$$
\begin{array}{ll}
 & \downarrow(T \ \langle\!|E|\!\rangle \ U) \qquad \{\text{the expression theorem (10)}\} \\
\cdot\!: & \downarrow((\mathsf{abs}.a' = E \ \wedge \ ok_\Sigma) \circ (T \ \langle\!|\mathsf{abs}.a|\!\rangle \ U)) \qquad \{\downarrow \text{ over } \circ \ (4)\} \\
\cdot\!: & \downarrow(\mathsf{abs}.a' = E \ \wedge \ ok_\Sigma) \circ \downarrow(T \ \langle\!|\mathsf{abs}.a|\!\rangle \ U) \qquad \{\downarrow \text{ over } \ \langle\!|\rangle\!| \ \ (5)\} \\
\cdot\!: & \downarrow(\mathsf{abs}.a' = E \ \wedge \ ok_\Sigma) \circ (\downarrow T \ \langle\!|\mathsf{abs}.a|\!\rangle \ \downarrow U) \\
\multicolumn{2}{l}{\{\text{jumping (8); nice catenation (6), induction hypotheses}\}} \\
\cdot\!: & e; \mathsf{zjump}\ 2 + \#t; t; \mathsf{jump}\ 1 + \#u; u \ @ \ p
\end{array}
$$

10.5 The while statement

We have $S = \textbf{while } E \textbf{ do } T \textbf{ od}$.

Now (3) says we want an s such that

$$(p = l)\downarrow S \quad \cdot\!: \quad s \ @ \ l$$

From the laws about while loops, only one has S on the left-hand side of a '$\cdot\!:$': induction. But to use induction requires the S to be isolated.

$$
\begin{array}{ll}
 & \{\text{Galois connection (2)}\} \\
\equiv & S \quad \cdot\!: \quad (p = l)\uparrow(s \ @ \ l)
\end{array}
$$

Let $U = (p = l)\uparrow(s \ @ \ l)$ and apply (1).

$$\Leftarrow \quad (\mathsf{t}'_\sigma \geq \mathsf{t}_\sigma \quad \cdot\!: \quad U) \wedge (\mathsf{wh}.U \quad \cdot\!: \quad U)$$

The first conjunct easily reduces to **true**. The derivation continues with the second conjunct:

$$
\begin{array}{ll}
 & \{\text{definition } \mathsf{wh} \text{ and } U\} \\
\equiv & T \circ (\mathsf{t}_\sigma :=_\Sigma \ \mathsf{t}_\sigma + 1) \circ (p = l)\uparrow(s \ @ \ l) \ \langle\!|E|\!\rangle \ ok_\Sigma \quad \cdot\!: \quad (p = l)\uparrow(s \ @ \ l) \\
 & \{\text{Galois connection (2)}\} \\
\equiv & (p = l)\downarrow T \circ (\mathsf{t}_\sigma :=_\Sigma \ \mathsf{t}_\sigma + 1) \circ (p = l)\uparrow(s \ @ \ l) \ \langle\!|E|\!\rangle \ ok_\Sigma \quad \cdot\!: \quad s \ @ \ l
\end{array}
$$

The remainder of the derivation is fairly straight forward and works towards being able to apply the looping theorem (9) to finally get

$$\Leftarrow \quad s = e; \mathsf{zjump}\ 2 + \#t; t; \mathsf{jump}\ -1 - \#t - \#e$$

10.6 Summary of the compiler

Assuming the existence of a correct code generator CE for expressions, the results of this section constitute a proof that the compiler

$$
\begin{array}{rcl}
\mathsf{C}.\mathsf{nil} & = & \mathsf{nil} \\
\mathsf{C}.(T; U) & = & \mathsf{C}.T; \mathsf{C}.U \\
\mathsf{C}.(v_m := E) & = & \mathsf{CE}.E; \mathsf{store\ addr}.m \\
\mathsf{C}.(\textbf{if } E \textbf{ then } T \textbf{ else } U \textbf{ fi}) & = & \mathsf{CE}.E; \mathsf{zjump}\ 2 + \#\mathsf{C}.T; \mathsf{C}.T; \\
 & & \mathsf{jump}\ 1 + \#\mathsf{C}.U; \mathsf{C}.U \\
\mathsf{C}.(\textbf{while } E \textbf{ do } T \textbf{ od}) & = & \mathsf{CE}.E; \mathsf{zjump}\ 2 + \#\mathsf{C}.T; \mathsf{C}.T; \\
 & & \mathsf{jump}\ -1 - \#\mathsf{C}.T - \#\mathsf{CE}.E
\end{array}
$$

is correct.

6th Refinement Workshop, David Till (Ed.), pp. 188–204, Springer-Verlag, 1994.

11 Conclusion

In the preceding I have outlined an approach to the predicative semantics of machine code and given a small example of its application to the specification and derivation of a simple compiler. Obviously, the compiler arrived at above could be written informally with much less effort. But, the simple compiler of the paper is only an example. The specification of the compiler is generic; it is valid for all source languages, machine languages, and representation relations, provided the semantics of the source language is given predicatively, and the machine language and representation relation meet the conditions mentioned in the paper. Similarly, most of the theorems used in the derivation of the compiler are generic and the techniques used in the derivation are reusable.

Along the way to compiler correctness we have developed a predicative interpretation for machine languages. Although it is illustrated by a simple machine language, it is suitable for machine languages with more complex instruction sets and more complex state spaces. This predicative interpretation may also be employed for other problems such as processor verification and reasoning about hand coded machine code.

Clearly there is work to be done on using the framework presented above for more elaborate and realistic source and target languages, for dealing with optimizing compilers, and for automating some of the theorem proving. These and other topics are further addressed in [Norvell 1993].

Acknowledgements

I would like to thank He Jifeng for helping me understand while loops, and Steve Cook for providing a superior proof of the separation theorem. I especially thank Ric Hehner, not only for many specific contributions to this paper, but also for many years of guidance and support in many ways. I am grateful to the Natural Sciences and Engineering Research Council for financial support, the Computing Laboratory at Oxford for logistical support, and especially to the Department of Computer Science at the University of Toronto for providing both.

References

[Cook 1993] Stephen A. Cook, 1993. Personal Communication.

[Hehner 1984] Eric C.R. Hehner. Predicative programming. *Communications of the ACM*, 27(2):134–151, 1984.

[Hehner 1993] Eric C.R. Hehner. *A Practical Theory of Programming.* Springer-Verlag, 1993.

[Hoare 1990] C.A.R. Hoare. Refinement algebra proves correctness of compiling specifications. Technical Report PRG-TR-6-90, Oxford University Computing Laboratory, Oxford University, 1990.

[Hoare 1992] C.A.R Hoare. Programs are predicates. In *Meeting of the Fifth Generation Project.* ICOT, 1992.

6th Refinement Workshop, David Till (Ed.), pp. 188–204, Springer-Verlag, 1994.

[Jones 1980] Neil D. Jones, editor. *Semantics-Directed Compiler Generation.* Number 94 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[Manasse and Nelson 1984] Mark Manasse and Greg Nelson. Correct compilation of control structures. Technical Report Technical Memorandum 11272-840909-09TM, AT&T Bell Laboratories, 1984.

[McCarthy and Painter 1967] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Proceedings of Symposia in Applied Mathematics, Volume XIX*, 1967.

[Milne and Strachey 1974] R. E. Milne and C. Strachey, editors. *A Theory of Programming Language Semantics.* Chapman & Hall, London, 1974.

[Morris 1973] F. Lockwood Morris. Advice on structuring compilers and proving them correct. In *Proceedings of the ACM Conferfence on Principles of Programming Languages*, 1973.

[Mosses 1980] Peter D. Mosses. A constructive approach to complier correctness. In [Jones 1980]. 1980.

[Norvell 1993] Theodore S. Norvell. *A Predicative Theory of Machine Languages and its Application to Compiler Correctness.* PhD thesis, University of Toronto, 1993.

[Polak 1981] Wolfgang Polak. *Compiler Specification and Verification.* Number 124 in Lecture Notes in Computer Science. Springer-Verlag, 1981. Also a Stanford Ph.D. thesis.

[Sampaio 1993] Augusto Sampaio. *An Algebraic Approach to Compiler Design.* PhD thesis, Oxford University, 1993.

[Tarski 1955] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Thatcher *et al.* 1980] James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. More advice on structuring compilers and proving them correct. In [Jones 1980]. 1980.