CS704: WLP for a Language with Pointers

Thomas Reps

04-19-2010

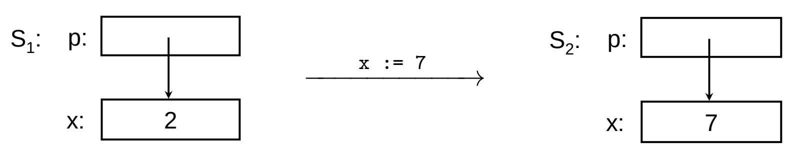
Abstract

This lecture concerns weakest liberal precondition for a language with pointers. It discusses two approaches to the issue: (i) one based on an enhanced rule of substitution (of programming-language elements into formulas), and (ii) one based on an encoding of the programming language semantics into logic.

1 Introduction

For a language with only integer variables, we have $\mathcal{WLP}(\texttt{x := e}, \varphi) = \varphi[e/x]$. However, for a language with pointers (i.e., variables that can hold addresses as values) and dereferencing, this rule no longer holds. For instance, suppose that the language is like C and allows taking the address of a variable (`&x`) and dereferencing (`*p`); then $\mathcal{WLP}(\texttt{x := e}, \varphi)$ is not necessarily $\varphi[e/x]$.

Example 1.1 $\mathcal{WLP}(\texttt{x := 7}, *p \geq 5) \neq (*p \geq 5)[7/x] = *p \geq 5$. Pictorially, we have



$[\![*p \geq 5]\!]^{S_1} = \mathit{false}$ $\qquad$ $[\![*p \geq 5]\!]^{S_2} = \mathit{true}$

In S_2 (and S_1), `p` and `&x` are aliases. That is, they are names for the same memory location: their rvalues equal the address of memory location `x`. □

2 An Enhanced Rule of Substitution

We first consider an enhanced rule of substitution [1] (which is similar to one originally devised by Morris [4]).

Definition 2.1 *A* location expression *is either*

- *a variable (e.g.,* `p`*)*
- *a structure field access from a location expression (e.g.,* `e.f`*)*
- *a dereference of a location expression (e.g.,* `*e`*)*

□

Note that the location expression for a C-style access of the form `p->f` would be represented by the location expression `(*p).f`.

The meaning of a location expression is the address of some memory location, which is defined by the *lvalue* interpretation of the location expression. That is, the lvalue interpretation of a location expression `e` yields the address of the location denoted by `e`. For instance, let `x` be a location expression; the lvalue interpretation of `x` denotes the address of `x`. (Equivalently, the *rvalue* interpretation of `&x` denotes the address of `x`.) Let `*p` be a location expression; the lvalue interpretation of `*p` denotes the contents of `p` (i.e, the address of the memory location that `p` points to.) (Equivalently, the *rvalue* interpretation of `p`—i.e., the contents of `p`—denotes the address of the memory location that `p` points to.)

Now consider $\mathcal{WLP}(\texttt{x := e}, \varphi)$. Let `y` be a location expression mentioned in φ. There are two cases to consider:

1. `x` and `y` are aliases. That is, as location expressions they denote the address of the same memory location.
2. `x` and `y` are not aliases, and thus the assignment `x := e` leaves the value of `y` unchanged.

Definition 2.2

- $\varphi[x,e,y] \stackrel{\text{def}}{=} \begin{array}{ll} & ((lval(x) = lval(y)) \wedge \varphi[e/y]) \\ \vee & ((lval(x) \neq lval(y)) \wedge \varphi). \end{array}$
- $\mathcal{WLP}(\texttt{x := e}, \varphi) \stackrel{\text{def}}{=} \varphi[x,e,y_1][x,e,y_2]\ldots[x,e,y_n]$, *where* y_1*,* y_2*,* $\ldots$*,* y_n *are the location expressions in* φ*.*

□

Note: if `y` is a variable different from `x`, and we assume that `x` and `y` are separate variables that occupy separate, non-overlapping memory locations, then $\&\texttt{x} \neq \&\texttt{y}$, and $\varphi[x,e,y]$ simplifies to φ. In other words, in such circumstances we can omit the case for $[x,e,y]$ in Defn. 2.2.

Example 2.3 Assume that `x`, `p`, and `q`, are separate variables that occupy separate, non-overlapping memory locations.

(a)

$$\begin{array}{rcl} \mathcal{WLP}(\texttt{x := 7}, *p \geq 5) & = & \begin{array}{ll} & (\&x = p \wedge 7 \geq 5) \\ \vee & (\&x \neq p \wedge *p \geq 5) \end{array} \\ & = & \&x = p \vee (\&x \neq p \wedge *p \geq 5) \end{array}$$

(b)

$$\begin{array}{rcl} \mathcal{WLP}(\texttt{x := 3}, *p \geq 5) & = & \begin{array}{ll} & (\&x = p \wedge 3 \geq 5) \\ \vee & (\&x \neq p \wedge *p \geq 5) \end{array} \\ & = & (\&x \neq p \wedge *p \geq 5) \end{array}$$

(c)

$$\begin{array}{rcl} \mathcal{WLP}(\texttt{x := 7}, *p \geq *q) & = & (*p \geq 5)[x,7,*p][x,7,*q] \\ & = & \left(\begin{array}{ll} & (\&x = p \wedge 7 \geq *q) \\ \vee & (\&x \neq p \wedge *p \geq *q) \end{array} \right)[x,7,*q] \\ & = & \begin{array}{ll} & \left(\&x = q \wedge \left(\begin{array}{ll} & (\&x = p \wedge 7 \geq 7) \\ \vee & (\&x \neq p \wedge *p \geq 7) \end{array} \right)\right) \\ \vee & \left(\&x \neq q \wedge \left(\begin{array}{ll} & (\&x = p \wedge 7 \geq *q) \\ \vee & (\&x \neq p \wedge *p \geq *q) \end{array} \right)\right) \end{array} \\ & = & \begin{array}{ll} & (\&x = q \wedge \&x = p) \\ \vee & (\&x = q \wedge \&x \neq p \wedge *p \geq 7) \\ \vee & \left(\&x \neq q \wedge \left(\begin{array}{ll} & (\&x = p \wedge 7 \geq *q) \\ \vee & (\&x \neq p \wedge *p \geq *q) \end{array} \right)\right) \end{array} \end{array}$$

$\square$

Note that we have not developed the machinery to handle more complicated assignments, such as $\mathcal{WLP}(\texttt{*p := *q}, x \geq 5)$.

3 First-Order Logic

In this section, we summarize the syntax and semantics of first-order logic. First-order logic will be used in §4 to encode a programming language's semantics.

3.1 Syntax

A logic is defined in terms of a *vocabulary* of (i) constant symbols, (ii) function symbols, and (iii) relation symbols. A constant symbol will be denoted by a subscripted c. We will use F^j and R^k for function symbols and relation symbols, respectively. The superscripts j and k indicate the *arity* of the symbol (and will be omitted when there is no chance of confusion).

We also assume that there is set of variables; a variable will be denoted by a subscripted v.

Terms. Let $t \in \mathit{Term}$ denote a term. The set Term of terms is defined as follows:

$$t \quad ::= \quad v_m \mid c_i \mid F^j(t_1, \ldots, t_j)$$

Formulas. Let $\varphi \in \mathit{Formula}$ denote a formula. The set $\mathit{Formula}$ of formulas is defined as follows:

$$\begin{array}{rcl} \varphi & ::= & t_1 = t_2 \\ & | & R^k(t_1, \ldots, t_k) \\ & | & \neg\varphi_1 \\ & | & \varphi_1 \wedge \varphi_2 \\ & | & \varphi_1 \vee \varphi_2 \\ & | & \forall v : \varphi_1 \\ & | & \exists v : \varphi_1 \end{array}$$

3.2 Semantics

The semantics of a first-order logic is given in terms of a *model* (over the given vocabulary used in the logic). A model S, also known as a *(logical) structure*, is a pair $S = (U, \iota)$, where U is a set of *individuals* and ι is an *interpretation*. (U and ι from structure S are often written as U^S and ι^S, respectively.) The interpretation ι provides a meaning for each constant symbol, function symbol, and relation symbol:

- ι maps each constant symbol to an individual
- ι maps each function symbol F^j to an arity-j function
- ι maps each relation symbol R^k to an arity-k relation

In other words, ι is overloaded to accept constant symbols function symbols, and relation symbols as arguments.

We also overload $[\![\cdot]\!]$ to define the meaning functions of both terms and formulas:

$$\begin{array}{l} [\![\cdot]\!] : \mathit{Term} \times \mathit{Struct} \times \mathit{Assignment} \rightarrow \mathit{Individual} \\ [\![\cdot]\!] : \mathit{Formula} \times \mathit{Struct} \times \mathit{Assignment} \rightarrow \mathit{Bool} \end{array}$$

An *assignment* Z is a function that maps free variables to individuals (i.e., an assignment has the functionality $Z : \{v_1, v_2, \ldots\} \rightarrow U$). $Z[v \mapsto u]$ denotes an update to Z in which variable v is mapped to u. We write the arguments to $[\![\cdot]\!]$ as follows: $[\![t]\!]^S Z$ and $[\![\varphi]\!]^S Z$.

Semantics of Terms. The semantics of terms is defined as follows:

$$\begin{array}{rcl} [\![c]\!]^S Z & = & \iota^S(c) \\ [\![v]\!]^S Z & = & Z(v) \\ [\![F^j(t_1,\ldots,t_j)]\!]^S Z & = & \iota(F^j)([\![t_1]\!]^S Z,\ldots,[\![t_j]\!]^S Z) \end{array}$$

Semantics of Formulas. The semantics of formulas is defined as follows, where 1 is used as the semantic value for *true*, 0 is used as the semantic value for *false*, and $\iota(eq)$ is predefined as the identity relation on individuals:

$$\begin{array}{rcl} [\![t_1 = t_2]\!]^S Z & = & \iota^S(eq)([\![t_1]\!]^S Z, [\![t_2]\!]^S Z) \\ [\![R^k(t_1,\ldots,t_k)]\!]^S Z & = & \iota(R^k)([\![t_1]\!]^S Z,\ldots,[\![t_k]\!]^S Z) \\ [\![\neg\varphi_1]\!]^S Z & = & 1 - [\![\varphi_1]\!]^S Z \\ [\![\varphi_1 \wedge \varphi_2]\!]^S Z & = & \min([\![\varphi_1]\!]^S Z, [\![\varphi_2]\!]^S Z) \\ [\![\varphi_1 \vee \varphi_2]\!]^S Z & = & \max([\![\varphi_1]\!]^S Z, [\![\varphi_2]\!]^S Z) \\ [\![\forall v : \varphi_1]\!]^S Z & = & \min_{u \in U^S} [\![\varphi_1]\!]^S Z[v \mapsto u] \\ [\![\exists v : \varphi_1]\!]^S Z & = & \max_{u \in U^S} [\![\varphi_1]\!]^S Z[v \mapsto u] \end{array}$$

4 Encoding of the Programming Language Semantics into Logic

In this section, we introduce an encoding of a programming language's semantics into logic. We will find that after we do so, $\mathcal{WLP}$ can be handled by pure substitution.

We will use the overloaded operator $\langle\cdot\rangle$ to denote the encodings of program variables, programming-language functions, programming-language relations, and assignment statements into elements of logical structures.

4.1 Encoding a Language with only Integer Variables

For a language with only integer-valued variables, the semantics only needs to use a one-level store. We encode the one-level store with a collection of constant symbols (one per program variable):

item	$\langle$*item*$\rangle$
program variable `x`	c_x
function symbol in language $(+, *, \ldots)$	same function symbol $(+, *, \ldots)$
equality symbol `==`	$=$
relation symbol in language $(<, \leq, \ldots)$	same relation symbol $(<, \leq, \ldots)$

To encode statements, we need *structure transformers*. That is, we want to specify how the input state (encoded as a structure) is transformed to an output state (encoded as a structure). For this, we use formula (of a special kind) with primed symbols denoting the symbol in the transformed (output) structure:

item	$\langle$*item*$\rangle$
`x := y + z`	$c'_x = c_y + c_z \wedge c'_y = c_y \wedge c'_z = c_z$

Note that variables whose value is unchanged by `x := y + z` have the identity transformation ($c'_y = c_y \wedge c'_z = c_z$). Strictly speaking, the identity transformation is also applied to all the function symbols and relation symbols of the programming language $(+, *, \ldots, <, \leq, \ldots)$, which have unchanging "intended" interpretations; e.g., $+' = \lambda w_1, w_2 . w_1 + w_2$. For brevity, these will be omitted.

Note that our two-vocabulary formulas have a special form: there is a single primed symbol on the left-hand side of each equality, and the right-hand-side term uses unprimed symbols exclusively.

4.2 Encoding a Language with Pointer Variables

To handle a language with pointer variables, a programming language's semantics needs to use a two-level store:

$$\sigma = (\eta, \rho),$$

where $\eta : name \rightarrow loc$ is the *environment* and $\rho : loc \rightarrow value$ is the *store*.

In contrast to §4.1, where constant symbols (and constants) encoded the store, here constant symbols (and constants) are used to encode the environment η. For each variable `x` we introduce a constant symbol c_x; however, as we become apparent below, c_x will model the *location* of `x`, not the *value* of `x`. To model the store ρ, we introduce a function symbol F_ρ.

In general, the statements of the programming language are now modeled as follows:

$$\langle \texttt{x := e} \rangle \longrightarrow c'_x = c_x \wedge c'_y = c_y \wedge \ldots \wedge F'_\rho \hookleftarrow F_\rho[c_x \mapsto \langle e \rangle].$$

Above, the notation $F' \hookleftarrow F[t_1 \mapsto t_2]$ denotes a function-update expression, which expresses how the post-state value of function F is expressed in terms of pre-state quantities. Henceforth, we will also use $\hookleftarrow$ for the updates to constants (to emphase that the special nature of our update formulas).

In general, statements of a language with pointers and dereferencing can be normalized into one of the following four forms, for which we give semantic encodings.

item	$\langle$*item*$\rangle$
`x := &y`	$c'_x \hookleftarrow c_x \wedge c'_y \hookleftarrow c_y \wedge F'_\rho \hookleftarrow F_\rho[c_x \mapsto c_y]$
`x := y`	$c'_x \hookleftarrow c_x \wedge c'_y \hookleftarrow c_y \wedge F'_\rho \hookleftarrow F_\rho[c_x \mapsto F_\rho(c_y)]$
`x := *y`	$c'_x \hookleftarrow c_x \wedge c'_y \hookleftarrow c_y \wedge F'_\rho \hookleftarrow F_\rho[c_x \mapsto F_\rho(F_\rho(c_y))]$
`*x := y`	$c'_x \hookleftarrow c_x \wedge c'_y \hookleftarrow c_y \wedge F'_\rho \hookleftarrow F_\rho[F_\rho(c_x) \mapsto F_\rho(c_y)]$

5 WLP for a Language with Pointers

We now consider $\mathcal{WLP}$ for the language discussed in §4.2. Because first-order logic is "referentially transparent" (i.e., supports substitution of equals for equals), we regain the simplicity of the rule for $\mathcal{WLP}$ that we had for a language with only integer variables. That is, with the encoding given in §4.2, $\mathcal{WLP}$ for a language with pointers and dereferencing can be performed by substitution.

Example 5.1 Consider again the example used in Ex. 1.1 and Ex. 2.3(a): $\mathcal{WLP}(\texttt{x := 7}, *p \geq 5)$. First, both `x := 7` and $*p \geq 5$ must be encoded:

$$\begin{array}{rcl} \langle \texttt{x := 7} \rangle & = & c'_x \hookleftarrow c_x \wedge c'_p \hookleftarrow c_p \wedge F'_\rho \hookleftarrow F_\rho[c_x \mapsto 7] \\ \langle *p \geq 5 \rangle & = & F'_\rho(F'_\rho(c'_p)) \geq 5 \end{array}$$

We now have

$$\begin{array}{rcl} \mathcal{WLP}(\texttt{x := 7}, *p \geq 5) & = & (F'_\rho(F'_\rho(c'_p)) \geq 5)[c'_x \hookleftarrow c_x \wedge c'_p \hookleftarrow c_p \wedge F'_\rho \hookleftarrow F_\rho[c_x \mapsto 7]] \\ & = & ((F_\rho[c_x \mapsto 7])((F_\rho[c_x \mapsto 7])(c_p))) \geq 5. \end{array}$$

□

The result given in Ex. 5.1 can be taken a step further, which will show how the result derived in Ex. 5.1 is related to the one given in Ex. 2.3(a) (and more generally how our substitution-based method relates to the method discussed in §2).

To do so, we equip our logic with an additional kind of *Term*, $ite(\varphi, t_1, t_2)$, to represent conditional expressions. ($ite(\varphi, t_1, t_2)$ is similar in spirit to the conditional-expression construct of C: φ `?` *exp* `:` *exp*.) We also use a standard axiom that relates function updates to *ite* expressions:

$$(F[k_1 \mapsto t])(k_2) = ite(k_1 = k_2, t, F(k_2)).$$

We now have (assuming that the variables `x` and `p` that we are modeling are separate variables that occupy separate, non-overlapping memory locations, and hence we can assume that $c_x = c_p$ is always *false*)

$$\begin{array}{rcl} \mathcal{WLP}(\texttt{x := 7}, *p \geq 5) & = & ((F_\rho[c_x \mapsto 7])((F_\rho[c_x \mapsto 7])(c_p))) \geq 5 \\ & = & ((F_\rho[c_x \mapsto 7])(ite(c_x = c_p, 7, F_\rho(c_p)))) \geq 5 \\ & = & ((F_\rho[c_x \mapsto 7])(F_\rho(c_p))) \geq 5 \\ & = & ite(c_x = F_\rho(c_p), 7, F_\rho(F_\rho(c_p))) \geq 5 \\ & = & ite(c_x = F_\rho(c_p), 7 \geq 5, F_\rho(F_\rho(c_p)) \geq 5) \\ & = & c_x = F_\rho(c_p) \vee (c_x \neq F_\rho(c_p) \wedge F_\rho(F_\rho(c_p)) \geq 5) \end{array}$$

Note that the last line is exactly the encoding of the result we obtained in Ex. 2.3(a), namely, $\&x = p \vee (\&x \neq p \wedge *p \geq 5)$.

For Further Information. More details about the ideas discussed in these notes can be found in the papers of Cartwright and Oppen [2], Scherpelz et al. [5], and Lim et al. [3].

References

[1] T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *Prog. Lang. Design and Impl.*, 2001.

[2] R. Cartwright and D.C. Oppen. The logic of aliasing. *Acta Inf.*, 15:365–384, 1981.

[3] J. Lim, A. Lal, and T. Reps. Symbolic analysis via semantic reinterpretation. *Softw. Tools for Tech. Transfer.* (To appear.).

[4] J.M. Morris. A general axiom of assignment. In M. Broy and G. Schmidt, editors, *Theor. Found. of Program. Methodology, Proc. of the 1981 Marktoberdorf Summer School*, volume 91 of *NATO Adv. Study Insts. Ser. C, Math. and Phys. Sci.*, pages 25–34. Reidel, 1982.

[5] E.R. Scherpelz, S. Lerner, and C. Chambers. Automatic inference of optimizer flow functions from semantics meanings. In *PLDI*, 2007.