Programming Language Semantics with Isabelle/HOL

Alfio Martini
Faculty of Informatics, PUCRS
Porto Alegre, Brazil
e-mail: alfio.martini@pucrs.br

***Abstract*—Isabelle is a generic meta-logical framework for implementing logical formalisms, and Isabelle/HOL is the specialization of Isabelle for HOL, which stands for *Higher Order Logic*. In programming language theory, formal semantics is the field concerned with the rigorous mathematical study of the meaning of programming languages. In this paper we develop a relational denotational semantics for a small imperative programming language and implement it as a theory in Isabelle/HOL. We give special attention to the specification of monotonic fixpoint functionals for loops and provide non-trivial proofs of interesting lemmas and properties with the structured proof language Isar.**

***Index Terms*—proof assistants, higher-order logic, formal semantics, Isabelle/HOL.**

I. Introduction

Proof assistants are computer systems that allow a user to do mathematics on a computer, where the proving and defining of mathematics is emphasized, rather than the computational (numeric or symbolic) aspect of it [3]. Thus a user can set up a mathematical theory, define properties and do logical reasoning with them. In many proof assistants one can also define functions and compute with them, but their main focus is on doing proofs. Important proof assistants include Isabelle [9], Coq [1], Agda [2] and the HOL family of theorem provers [4].

On the other hand, in programming language theory, formal semantics is the field concerned with the rigorous mathematical study of the meaning of programming languages. Formal semantics, for instance, is instrumental in the writing of compilers, helps a person better understand what a program is doing, and is useful for proving important equivalencies between compound statements in the programming language. The three major approaches to formal semantics are denotational semantics, axiomatics semantics and operational semantics [11]. In a denotational definition, each phrase (part of a program) S is given a denotation $[\![S]\!]$, which is a mathematical object representing the contribution of S to the meaning of any complete program in which it occurs. The denotation of a phrase is determined just by the denotations of its subphrases. In this case one says that the semantics is *compositional*.

In this work, we develop a relational denotational semantics of a small imperative language along the lines of [11] and [7] and provide a complete formalization of this semantics as a logical theory in Isabelle/HOL. One of the great advantages of using Isabelle relies on the fact that one can write highly readable and structured proofs with a very sophisticated proof language called Isar. The material we present here is especially recommended for computer scientists and students alike who are interested in the formalization of programming language semantics with proof assistants. It is an essential part of a short course I gave on Isabelle/HOL during the 2^{nd} *Workshop on Theoretical Computer Science* (WEIT2013) that took place in 2013, Rio Grande, Brazil.

This paper is organized as follows: in section II we introduce the basic machinery needed in this work, which includes familiar concepts for functions, relations, lattices and fixed points. In section III we introduce the core of a small imperative language, covering the abstract syntax for expressions and commands and their corresponding denotational semantics. In section IV we introduce Isabelle/HOL, a specialization of the Isabelle meta-logical framework for Higher Order Logic. In section V we describe in detail the implementation in Isabelle/HOL of the syntax and semantics initially developed in section III. In section VI we introduce the proof language of Isabelle, called Isar, and detailed examples of proofs related to the denotational semantics developed in this work. Finally, in section VII, we state some concluding remarks.

II. Basic Concepts

A. Relations and Functions

The set of all subsets of a set X, called *powerset* and written as $\mathcal{P}(X)$, is defined as $\mathcal{P}(X) \triangleq \{A \mid A \subseteq X\}$. A *binary relation* f between sets X and Y is a subset of the cartesian product $X \times Y$. If f is a relation from X to Y we write it as $f : X \leftrightarrow Y$ or $f : \mathcal{P}(X \times Y)$. We call X the domain and Y the codomain of the relation. Given a relation $f : X \leftrightarrow Y$, the domain of definition of f is the set of all $x \in X$ that are related to some $y \in Y$, i.e.,

$$dom(f) \triangleq \{x \in X \mid \exists y \in Y.\ (x, y) \in f\}$$

If $f : X \leftrightarrow Y$ and $g : Y \leftrightarrow Z$ are relations, then the composition of f and g, written $f; g : X \leftrightarrow Z$ is defined as

$$f; g \triangleq \{(x, z) \mid \exists y \in Y.(x, y) \in f \wedge (y, z) \in g\}$$

A *partial function* f from X to Y, written $f : X \rightharpoonup Y$ is a relation from X to Y such that for all $x \in X, y, z \in Y, (x, y) \in f$ and $(x, z) \in f$ then $y = z$, i.e., if every $x \in X$ is related to at most one $y \in Y$. We write $f(x) = y$ for this unique

y related to x. A partial function $f : X \rightharpoonup Y$ is called a *total function* or simply function if $dom(f) = X$, i.e., every $x \in X$ is related to exactly one element $y \in Y$. If f is a total function from X to Y, then this is written as $f : X \to Y$. The composition of total functions $f : X \to Y$ and $g : Y \to Z$ is defined by

$$g \circ f : X \to Z \triangleq g(f(x)) \text{ for all } x \in X$$

We use the letter $\mathbb{N}$ to represent the set of natural numbers $\{0, 1, 2, \ldots\}$, $\mathbb{B}$ to represent the set of boolean values $\{true, false\}$ and the letter $\mathbb{Z}$ to denote the set of integers $\{\ldots, -1, 0, 1, \ldots\}$. Given a set D we have the $\mathbf{if\_then\_else\_} : \mathbb{B} \times D \times D \to D$ total function, defined by

$$\textbf{if } b \textbf{ then } d_1 \textbf{ else } d_2 = \begin{cases} d_1 & \text{if } b = true \\ d_2 & \text{otherwise} \end{cases}$$

For any total function $f : X \to Y$ we have the *update* function $f[\_ \mapsto \_] : X \to Y \to (X \to Y)$, which is a function that takes arguments x and y and returns the function that differs from f only at $x \in X$ by taking on the value $y \in Y$. This updated function is denoted $f[x \mapsto y]$ and is defined as

$$f[x \mapsto y](k) = \begin{cases} y & \text{if } x = k \\ f(k) & otherwise \end{cases} \quad \text{(II.1)}$$

Given a total function $f : X \to X$, the n-fold composition of f, written f^n, is defined by induction as

$$\begin{array}{lcl} f^0(x) & = & id(x) \\ f^{n+1}(x) & = & f(f^n(x)) \end{array}$$

B. Lattices and Fixed Points

A binary relation $\leq$ on a set D is called a partial order if and only if it has the following properties:

- $\forall d \in D.\ d \leq d$ (reflexive)
- $\forall d, d' \in D.\ d \leq d' \leq d \to d = d'$ (antisymmetric)
- $\forall d, d', d'' \in D.\ d \leq d' \leq d'' \to d \leq d''$ (transitive)

The pair $(D, \leq)$ is called a *partially ordered set* or simply a *poset*. If $d_1, d_2 \in D$ and $d_1 \leq d_2$ we say that d_1 *approximates or is equal to the information content* of d_2.

Assume that $(D, \leq)$ is a partial order and $A \subseteq D$. If there is an element $i \in D$ such that

- $\forall x \in A.\ i \leq x$ (i is a lower bound)
- $\forall y \in D.\ \forall x \in A.\ y \leq x \to y \leq i$ (i is greatest),

then i is called the *greatest lower bound* (glb) or *infimum* of A. The infimum of A is denoted $\bigsqcap A$. The dual notion, i.e., of the *least upper bound* or *supremum* of a subset A of D is defined analogously and is denoted $\bigsqcup A$. An element $m \in A$ is called a *minimal* element of A if and only if $\forall x \in A.\ x \leq m \to x = m$.

Given partial orders $(D, \leq)$ and $(E, \leq)$, a total function $f : D \to E$ is said to be *monotonic* if and only if for all $d, d' \in D.\ d \leq d' \to f(d) \leq f(d')$.

A poset $(L, \leq)$ is a complete lattice if every subset A of L has both an infimum and a supremum. In a complete lattice L two elements are particularly interesting: the top element $\top \triangleq \bigsqcap \varnothing$ and the bottom element $\bot \triangleq \bigsqcup \varnothing$.

Example II.1. *The power set $\mathcal{P}(X)$ of a set X is a complete lattice with the inclusion order where the supremum and infimum are given, respectively, by union and intersection of subsets*

Definition II.2. *If L is a set and $f : L \to L$ is a function, then $k \in L$ is called a fixed point of f if $f(k) = k$. If a function $f : L \to L$ has exactly one minimal fixed point then this fixed point is called the least fixed point, which is denoted as $lfp(f)$.*

Lemma II.3 (Knaster-Tarski)**.** *Let $(L, \leq)$ be a lattice and $f : L \to L$ be a monotonic function. Then $f : L \to L$ has a least fixed point $k \in L$ defined by*

$$k \triangleq \bigsqcap \{x \in L \mid f(x) \leq x\}$$

Proof. To show that k is a fixed point of f we show that $f(k) \leq k$ and $k \leq f(k)$. Then the desired property follows directly from antisymmetry. Let $P \triangleq \{x \in L \mid f(x) \leq x\}$, $k \triangleq \bigsqcap P$ and x_0 be an arbitrary element of P. First note that $P \neq \varnothing$ since at least $\top \in P$. By the definition of infimum, $k \leq x_0$. Since f is monotonic $f(k) \leq f(x_0)$. By the definition of P, $f(x_0) \leq x_0$. By transitivity of $\leq$, $f(k) \leq x_0$. Since x_0 is arbitrary, we have that $\forall x \in P.\ f(k) \leq x$. Thus $f(k)$ is a lower bound of P. Since k is the greatest lower bound, $f(k) \leq k$. Now, from this we have that $f(f(k)) \leq f(k)$, since f is monotonic. Thus $f(k) \in P$ by definition. Since k is the infimum of P we have $k \leq f(k)$. Thus $f(k) = k$ and thus k is indeed a fixed point of $f : L \to L$. Now note that, since $\leq$ is reflexive, all the fixpoints of f are in P. But k is the infimum of P and thus the least fixed point of $f : L \to L$. Hence, $lfp(f) = k$. □

III. A Small Imperative Language

In this section we introduce a small imperative language that will be used as an example in this paper. The language has the familiar syntactic constructions for arithmetic and boolean expressions as well as for imperative statements like assignments, sequential, conditional and iterative commands. We will occasionally call this language ***IMP*** throughout our discussion. The presentation is based in [11].

A. Syntax of Expressions and Commands

The syntactic sets associated with ***IMP*** are countably infinite sets for

- numerals ***Num***, which stands for integers $\mathbb{Z}$,
- memory locations ***Loc***,
- arithmetic expressions ***AExp***,
- boolean expressions ***BExp***, and
- commands or statements ***Prg***.

The concrete syntactic structure of numerals and locations is assumed to be given (elsewhere). The structure of expressions and commands are given below by defining the abstract syntax

of each syntactic category. Note that this means that they denote parse (term) trees and not strings of symbols.

An arithmetic expression can be an integer number, a memory location, or a binary arithmetic expression that is meant to denote sum, subtraction, and multiplication of arithmetic expressions.

$$\begin{array}{rcll} a : \textbf{\textit{AExp}} & ::= & n & n \in \textbf{\textit{Num}} \\ & | & X & X \in \textbf{\textit{Loc}} \\ & | & a_0 \oplus a_1 & a_0, a_1 \in \textbf{\textit{AExp}} \\ & | & a_0 \ominus a_1 & a_0, a_1 \in \textbf{\textit{AExp}} \\ & | & a_0 \otimes a_1 & a_0, a_1 \in \textbf{\textit{AExp}} \end{array} \tag{III.1}$$

A boolean expression can be a constant for true and false, an equality or inequality between arithmetic expressions, a negation of a boolean expression or conjunction of boolean expressions.

$$\begin{array}{rcll} b :: \textbf{\textit{BExp}} & ::= & \textbf{\textit{true}} & \\ & | & \textbf{\textit{false}} & \\ & | & a_0 \; eq \; a_1 & a_0, a_1 \in \textbf{\textit{AExp}} \\ & | & a_0 \preceq a_1 & a_0, a_1 \in \textbf{\textit{AExp}} \\ & | & !b & \\ & | & b_0 \;\&\&\; b_1 & \end{array} \tag{III.2}$$

A statement or program can be the identity command, an assignment of an arithmetic expression to a location, a conditional command, an iterative loop and finally a sequence of two commands.

$$\begin{array}{rcll} S : \textbf{\textit{Prg}} & ::= & \underline{\textbf{skip}} & \\ & | & X := a & X \in \textbf{\textit{Loc}}, a \in \textbf{\textit{AExp}} \\ & | & \underline{\textbf{if}}\; B \; \{S_1\} \; \underline{\textbf{else}} \; \{S_2\} & B \in \textbf{\textit{BExp}} \\ & | & \underline{\textbf{while}}\; B \; \{S\} & B \in \textbf{\textit{BExp}} \\ & | & \underline{\textbf{repeat}}\; \{S\} \; \underline{\textbf{until}}\; B & B \in \textbf{\textit{BExp}} \\ & | & S_1 ; ; S_2 & S_1, S_2 \in \textbf{\textit{Prg}} \end{array} \tag{III.3}$$

B. Semantics of Expressions and Commands

In a denotational definition, each phrase (part of a program) S is given a denotation $[\![S]\!]$, a mathematical object representing the contribution of S to the meaning of any complete program in which it occurs. The denotation of a phrase is determined just by the denotations of its subphrases. In this case one says that the semantics is *compositional*.

The denotational semantics of arithmetic and boolean expressions is straightforward and can be given by a primitive recursive function by structural induction on the corresponding syntactic structure.

In the following, let a *state* be a function $\sigma : \textbf{\textit{Loc}} \to \mathbb{Z}$ and by $\Sigma = (\textbf{\textit{Loc}} \to \mathbb{Z})$ we denote the set of all possible such states. Thus $\sigma(X)$ is the value, or contents, of location X in state σ.

The semantics of an arithmetic expression is a total function $\mathsf{E}[\![\_]\!] : \textbf{\textit{AExp}} \to (\Sigma \to \mathbb{Z})$. We assume an evaluation function $eval_{\mathbb{Z}} : \textbf{\textit{Num}} \to \mathbb{Z}$ that compiles numerals into integers. Now, the semantics of arithmetic expressions is given by the following primitive recursive function:

$$\begin{array}{l} \mathsf{E}[\![n]\!]\sigma = eval_{\mathbb{Z}}(n) \\ \mathsf{E}[\![X]\!]\sigma = \sigma(X) \\ \mathsf{E}[\![a_0 \oplus a_1]\!]\sigma = \mathsf{E}[\![a_0]\!]\sigma + \mathsf{E}[\![a_1]\!]\sigma \\ \mathsf{E}[\![a_0 \otimes a_1]\!]\sigma = \mathsf{E}[\![a_0]\!]\sigma \times \mathsf{E}[\![a_1]\!]\sigma \\ \mathsf{E}[\![a_0 \ominus a_1]\!]\sigma = \mathsf{E}[\![a_0]\!]\sigma - \mathsf{E}[\![a_1]\!]\sigma \end{array} \tag{III.4}$$

Note that the arithmetic signs on the left of the equations are just operation symbols while on the right they denote the actual functions.

The semantics of a boolean expression is a total function $\mathsf{B}[\![\_]\!] : \textbf{\textit{BExp}} \to (\Sigma \to \mathbb{B})$ defined as

$$\begin{array}{l} \mathsf{B}[\![\textbf{\textit{true}}]\!]\sigma = true \quad \mathsf{B}[\![\textbf{\textit{false}}]\!]\sigma = false \\ \mathsf{B}[\![a_0 \; eq \; a_1]\!]\sigma = (\mathsf{E}[\![a_0]\!]\sigma = \mathsf{E}[\![a_1]\!]\sigma) \\ \mathsf{B}[\![a_0 \preceq a_1]\!]\sigma = (\mathsf{E}[\![a_0]\!]\sigma \leq \mathsf{E}[\![a_1]\!]\sigma) \\ \mathsf{B}[\![!b]\!]\sigma = \neg(\mathsf{B}[\![b]\!]\sigma) \\ \mathsf{B}[\![b_0 \;\&\&\; b_1]\!]\sigma = \mathsf{B}[\![b_0]\!]\sigma \wedge \mathsf{B}[\![b_1]\!]\sigma \end{array} \tag{III.5}$$

In denotational semantics of imperative languages, commands are normally interpreted as partial functions from states to states [11]. Since partial functions are special cases of relations, we give here a relational semantics of commands along the lines of [11], [7]. Thus we have that

$$[\![S]\!] : \mathcal{P}(\Sigma \times \Sigma), \text{ i.e., } [\![S]\!] \subseteq \Sigma \times \Sigma$$

From now on we stop using B, E in the corresponding semantic functionals whenever it does not lead to confusion. Now the relational semantics of our imperative language is given as follows:

$$\begin{array}{l} [\![\underline{\textbf{skip}}]\!] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\} \\ [\![X := E]\!] = \{(\sigma, \sigma[X \mapsto [\![E]\!]]) \mid \sigma \in \Sigma\} \\ [\![S_1 ; ; S_2]\!] = [\![S_1]\!]; [\![S_2]\!] \\ [\![\underline{\textbf{if}}\; B \; \{S_1\} \; \underline{\textbf{else}} \; \{S_2\}]\!] = \\ \quad \{(\sigma, \sigma') \mid [\![B]\!]\sigma = true \wedge (\sigma, \sigma') \in [\![S_1]\!]\} \cup \\ \quad \{(\sigma, \sigma') \mid [\![B]\!]\sigma = false \wedge (\sigma, \sigma') \in [\![S_2]\!]\} \\ [\![\underline{\textbf{while}}\; B \; \{S\}]\!] = lfp(\Phi \; [\![B]\!] \; [\![S]\!]) \\ [\![\underline{\textbf{repeat}}\; \{S\} \; \underline{\textbf{until}}\; B]\!] = lfp(\Psi \; [\![B]\!] \; [\![S]\!]) \end{array} \tag{III.6}$$

The discussion that follows explain in detail the meaning of the **while** and **repeat** loops as least fixed point of functionals $\Phi \; [\![B]\!] \; [\![S]\!]$ and $\Psi \; [\![B]\!] \; [\![S]\!]$ which are both of type $\mathcal{P}(\Sigma \times \Sigma) \to \mathcal{P}(\Sigma \times \Sigma)$.

The intuitive understanding of the **while** construct is that its meaning should satisfy for all states σ the following equation:

$$[\![\underline{\textbf{while}}\; B \; \{S\}]\!] = [\![\underline{\textbf{if}}\; B \; \{S; \underline{\textbf{while}}\; B \; \{S\}\} \; \underline{\textbf{else}} \; \{\underline{\textbf{skip}}\}]\!]$$

However, taking $W \triangleq \underline{\textbf{while}}\; B \; \{S\}$ and manipulating the right expression in the above equation we have

$$\begin{array}{rcl} [\![W]\!] & = & \{(\sigma, \sigma') \mid [\![B]\!]\sigma = true \wedge (\sigma, \sigma') \in [\![S; W]\!]\} \\ & & \cup \{(\sigma, \sigma') \mid [\![B]\!]\sigma = false \wedge (\sigma, \sigma') \in [\![\underline{\textbf{skip}}]\!]\} \end{array}$$

and thus

$$[\![W]\!] \quad = \quad \{(\sigma,\sigma') \mid [\![B]\!]\sigma = true \wedge (\sigma,\sigma') \in [\![S]\!]; [\![W]\!]\} \cup \{(\sigma,\sigma) \mid [\![B]\!]\sigma = false\} \tag{III.7}$$

This recursive equation is not compositional, since the meaning of the **while** construct does not depend solely on the meaning of its sub-phrases S and B. That is to say, the right side contains as a sub-phrase the very phrase whose denotation we are trying to define. In order to formalize precisely what does it mean to solve recursive equations as the one in (III.7), we need to consider *generating functions* or *functionals*. Let be given $x : D$ and a recursive equation of the form $x = (\ldots x \ldots)$. Then, such an equation can always be coded as a total function $\tau : D \to D$ as follows

$$\tau = \lambda x : D.\ (\ldots x \ldots)$$

Solutions of such recursive equations are then always fix-points of the correspoding generating functions.

Now, abstracting $[\![B]\!]$ and $[\![S]\!]$ with the parameters $b : \Sigma \to \mathbb{B}$ and $c : \mathcal{P}(\Sigma \times \Sigma)$, we have the functional $\Phi\ b\ c : \mathcal{P}(\Sigma \times \Sigma) \to \mathcal{P}(\Sigma \times \Sigma)$, where

$$\Phi\ b\ c \quad \triangleq \quad \lambda\varphi.\ \{(\sigma,\sigma') \mid b(\sigma) = true \wedge (\sigma,\sigma') \in c\,;\varphi\} \cup \{(\sigma,\sigma) \mid b(\sigma) = false\} \tag{III.8}$$

Thus, $lpf(\Phi\ [\![B]\!]\ [\![C]\!])$ stands for the least fixed point of $\Phi\ b\ c$.

Using analogous reasoning one has that the functional associated with the **repeat** constructor is

$$\Psi\ b\ c \quad \triangleq \quad \lambda\varphi.\ c\,;(\{(\sigma,\sigma') \mid b(\sigma) = false \wedge (\sigma,\sigma') \in \varphi\} \cup \{(\sigma,\sigma) \mid b(\sigma) = false\}) \tag{III.9}$$

It remains to show that both $\Phi\ b; c$ and $\Psi\ b\ c$ are monotonic functions from $\mathcal{P}(\Sigma \times \Sigma)$ to $\mathcal{P}(\Sigma \times \Sigma)$. This will be shown directly as formally verified proof in Isar in the context of section VI-B.

IV. Basic Concepts of Isabelle

Isabelle is a generic meta-logical framework for implementing logical formalisms, and Isabelle/HOL is the specialization of Isabelle for HOL, which stands for *Higher Order Logic*. In what follows we give the basic concepts of Isabele/HOL which are needed to follow this paper. We strongly recommend [9], [6] for an in-depth coverage of these and many other essential concepts.

A. Types, Terms and Formulas

HOL is a typed logic whose type system resembles that of functional programming languages. Thus there are

- **base types**, in particular *bool*, the type of truth values, *nat*, the type of natural numbers ($\mathbb{N}$), and *int*, the type of mathematical integers ($\mathbb{Z}$);
- **type constructors**, in particular *list*, the type of lists, and *set*. Type constructors are written postfix, e.g. *nat set* is the type of sets whose elements are natural numbers;
- **function types**, denoted by $\Rightarrow$. For instance, *nat list* $\Rightarrow$ *nat* represents the type of all total functions from lists of natural numbers to natural numbers;
- **type variables**, denoted by *'a*, *'b* etc., just like in ML [8].

Terms are formed as in functional programming by applying functions to arguments. If f is a function of type $\tau_1 \Rightarrow \tau_2$ and t is a term of type τ_1 then $f\ t$ is a term of type τ_2. The expression $t :: \tau$ means that the term t has type τ. Natural numbers and integers are equipped with the usual arithmetic expressions. For instance, the overloaded constants *op +*, *op −*, *op ∗* denote adition, subtraction and multiplication of arithmetic expressions. Thus the expression $(x::'a) + (y::'a)$ has type *'a*, while $(x::int) + (y::int)$ has type *int*. HOL also supports some basic constructs from functional programming, especially conditional, let and case expressions. Formulas are terms of type *bool*. There are the basic constants *True* and *False* and the usual logical connectives (in decreasing order of precedence): $\neg$, $\wedge$, $\vee$, $\longrightarrow$.

Equality is available in the form of the infix function $=$ of type *'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool*. It also works for formulas, where it means *if and only if*. Quantifiers are written $\forall x.\ P$ and $\exists x.\ P$.

Isabelle automatically computes the type of each variable in a term. This is called **type inference**. Despite type inference, it is sometimes necessary to attach explicit **type constraints** to a variable or term. The syntax is $t :: \tau$ as in *n*::*nat*.

Finally there are the universal quantifier $\bigwedge$ and the implication $\Longrightarrow$. They are part of the Isabelle framework, not the logic HOL. They are used essentially for generality (the notion of an arbitrary value) and judgments or inference rules, respectively. Right-arrows of all kinds always associate to the right. In particular, the formula $A_1 \Longrightarrow A_2 \Longrightarrow A_3$ means $A_1 \Longrightarrow (A_2 \Longrightarrow A_3)$. The (Isabelle specific) notation $[\![\ A_1;\ \ldots;\ A_n\ ]\!] \Longrightarrow A$ is a shorthand for the iterated implication $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A$. An iterated implication like this will often indicate a judgment $A_1, \ldots, A_n \vdash A$ with assumptions $A_1, \ldots, A_n$ and conclusion A. Finally, applying a theorem $A \Longrightarrow B$ (named *T*) to a theorem *A* (named *U*) is written *T*[*OF U*] and yields theorem *B*.

V. Denotational Semantics in Isabelle/HOL

A. Semantics of Expressions

A definition is simply an abbreviation, i.e., a new name for an existing construction. In particular, definitions cannot be recursive. Isabelle offers definitions on the level of types and terms. Those on the type level are called **type synonyms**. Type synonyms are created by the **type_synonym** command.

type_synonym *vname* = *string*
type_synonym *val* = *int*
type_synonym *state* = *vname* $\Rightarrow$ *val*

The above declarations say that *vname*, *val* and *state* are synonyms, respectively, for the types of strings, integers and total functions from strings to integers. The main purpose is to improve readability. They can be used like any other type.

Context free grammars like the ones defined in section III-A are essentially inductive sets. They can be represented in a straightforward way in Isabelle as inductive types through the datatype definition.

datatype *aexp* = *N int* | *V vname* |
Plus aexp aexp (**infixl** $\oplus$ 65)|
Times aexp aexp (**infixl** $\otimes$ 70)|
Minus aexp aexp (**infixl** $\ominus$ 60)

The type *aexp* of arithmetic expressions has five constructors, which here are operators which receive one or more arguments and return always an element of type *aexp*: the constructor *N* adds all integers to the *aexp*; *V* that takes a string and returns an arithmetic expression. The remaining constructors *op* $\oplus$, *op* $\otimes$ and *op* $\ominus$ all take two arithmetic expressions as arguments and return also an arithmetic expressions as a result. Also the datatype declaration is annotated with an alternative syntax. For instance, the term *Plus a b* can be written as $a \oplus b$. The declaration **infixl** says that is an infix operator that associates to the left. The number 65 is its priority.

Hence all arithmetic expressions are of the form *N k*, *V s*, $a \oplus b$, $a \otimes b$, $a \ominus b$, for arbitrary integers k, strings s and arithmetic expressions a, b.

Total functions on inductive datatypes can easily be defined by primitive recursion.

primrec *aval* :: *aexp* $\Rightarrow$ *state* $\Rightarrow$ *val* **where**
aval (*N n*) *s* = *n* |
aval (*V x*) *s* = *s x* |
aval (*a*1 $\oplus$ *a*2) *s* = *aval a*1 *s* + *aval a*2 *s*|
aval (*a*1 $\otimes$ *a*2) *s* = *aval a*1 *s* $*$ *aval a*2 *s*|
aval (*a*1 $\ominus$ *a*2) *s* = *aval a*1 *s* − *aval a*2 *s*

Each function definition is of the form

primrec *name* :: *type (optional syntax)* **where** *equations*.

Primitive recursion over a datatype t means that the recursion equations must be of the form

$$f\, x_1 \ldots (C\, y_1 \ldots y_k) \ldots x_n = r$$

such that C is a constructor of t and all recursive calls of f in r are of the form $f \ldots y_i \ldots$ for some i. Thus Isabelle immediately sees that f terminates because one argument becomes smaller with every recursive call. There must be at most one equation for each constructor. Their order is immaterial.

The following inductive type definition as well as the evaluation of boolean expressions are straightfoward translations from the respective definitions in (III.2) and (III.5).

datatype *bexp* = *Bc bool* | *Not bexp* (!)|
And bexp bexp (**infixr** && 35) |
Eq aexp aexp (**infix** *eq* 45) |
Leq aexp aexp (**infix** $\preceq$ 50)

primrec *bval* :: *bexp* $\Rightarrow$ *state* $\Rightarrow$ *bool* **where**
bval (*Bc v*) *s* = *v* |
bval (!*b*) *s* = (¬ *bval b s*) |
bval (*b*1 && *b*2) *s* = (*bval b*1 *s* ∧ *bval b*2 *s*) |
bval (*a*1 *eq a*2) *s* = (*aval a*1 *s* = *aval a*2 *s*) |
bval (*a*1 $\preceq$ *a*2) *s* = (*aval a*1 *s* ≤ *aval a*2 *s*)

B. Semantics of Commands

Below we specify the abstract grammar from (III.3) as an inductive datatype in Isabelle. There is one constructor for each command in the language.

datatype
com = *SKIP*
| *Assign vname aexp* (_ ::= _ [62, 62] 65)
| *Seq com com* (_;; _ [61, 61] 60)
| *If bexp com com* (*if*_ *then* _*else*_ *fi*
[60, 60, 60] 62)
| *While bexp com* (*while_do*_ [60, 60] 63)
| *Repeat com bexp* (*repeat_until*_ [60, 60] 63)

Note that each constructor is annotated with an optional concrete syntax. For instance, in the *While* constructor, the string *while_do_* is a template, where *while* and *do* are a sequence of characters that acts as delimiters surrounding the argument positions indicated by the underscores. Thus, for arbitrary boolean expression b and command c the user is allowed to write a while command either as *While b c* or as *while b do c*. The numbers represent the priorities of the arguments and result of the constructor operator.

Here we introduce a type synonym for the denotation of commands. Thus, in the declaration below, $CDen$ stands for the formalization of the type of $\mathcal{P}(\Sigma \times \Sigma)$.

type_synonym *CDen* = (*state* × *state*) *set*

Below we encode in a straightforward way the functionals associated to the *repeat* and *while* loops presented and discussed, respectively, in (III.8) and (III.9). Sequential relation composition in Isabelle is defined as *r O s* = {(*x*, *z*) | ∃*y*. (*x*, *y*) ∈ *r* ∧ (*y*, *z*) ∈ *s*}.

definition
Φ :: (*state* ⇒ *bool*) ⇒ *CDen* ⇒ (*CDen* ⇒ *CDen*)
where Φ *b cd* = (λφ. {(*s*,*t*). (*s*,*t*) ∈ (*cd O* φ) ∧ *b s*}
∪ {(*s*,*t*). *s*=*t* ∧ ¬ *b s*})

Note that set comprehension in Isabelle, written in code as {*x*. *P x*}, denotes the set of all elements that satisfy the predicade P.

definition
Ψ ::(*state* ⇒ *bool*) ⇒ *CDen* ⇒ (*CDen* ⇒ *CDen*)
where Ψ *b cd* = (λφ. *cd O* ({(*s*,*t*). *s*=*t* ∧ *b s*}
∪ {(*s*,*t*). (*s*,*t*) ∈ φ ∧ ¬ *b s*}))

Note that for an arbitrary boolean function b and command denotation c, both Φ *b c* and Ψ *b c* have type *CDen* ⇒ *CDen*. Now we can give the recursive definition of the denotational semantics of commands as defined in (III.6).

fun $C :: com \Rightarrow CDen$ ($[\![\_]\!]$) **where**
skip: $[\![SKIP]\!] \quad = \{(s,t).\ s{=}t\} \ \mid$
assign: $[\![x ::= a]\!] = \{(s,t).\ t = s(x := aval\ a\ s)\} \mid$
seq: $[\![c0;;c1]\!] = C(c0)\ O\ C(c1) \mid$
cond: $[\![(if\ b\ then\ c1\ else\ c2\ fi)]\!] =$
$\{(s,t).\ (s,t) \in C\ c1 \wedge bval\ b\ s\}$
$\cup \{(s,t).\ (s,t) \in C\ c2 \wedge \neg bval\ b\ s\} \mid$
while:$[\![while\ b\ do\ c]\!] = lfp\ (\Phi\ (bval\ b)\ [\![c]\!])\mid$
repeat:$[\![repeat\ c\ until\ b]\!] = lfp\ (\Psi\ (bval\ b)\ [\![c]\!])$

In Isabelle, the expression $f(x := y)$ represents a function update, i.e., in f, the argument x is updated with the value expression y. The update operator has an important simplication rule which encondes the analogous equation defined in (II.1):

$$(f(x := y))\ z = (\textit{if}\ z = x\ \textit{then}\ y\ \textit{else}\ f\ z)$$

Thus, the expression $s(x := aval\ a\ s)$ denotes the state s where x is updated with the denotation of the arithmetic expression a in state s. The constant *lfp* in Isabelle has type $('a \Rightarrow 'a) \Rightarrow 'a$ and definition $\bigcap \{u \mid f\ u \leq u\}$ where $\bigcap$ has type $'a\ set \Rightarrow 'a$.

VI. The Isar Proof Language

In this section, we introduce the structured proof language Isar. Isar stands for *Inteligible semi-automated reasoning* and was developed by Markus Wenzel in his PhD thesis [10]. The Isar proof language provides a general framework for human-readable natural deduction proofs. We assume the reader is familiar with proofs in the natural deduction style.

A. Basic Elements of Isar

The two key features of Isar are: it is structured, not linear; it is readable without running it because you need to state what you are proving at any given point. Isar proofs are like structured programs with comments. We introduce Isar by example and using a standard natural deduction proof as a guide to fix the basic elements of the language. For this, we consider the proof of the conjecture (or sequent): $\forall x.F(x) \vee \forall x.G(x) \vdash \forall x.(F(x) \vee G(x))$. The standard natural deduction proof goes as follows:

1	$\forall x.F(x) \vee \forall x.G(x)$	Pr
2	x_0	hyp, $\forall\mathcal{I}$
3	$\forall x.F(x)$	hyp, $\vee\mathcal{E}$
4	$F(x_0)$	$\forall\mathcal{E}(3)$
5	$F(x_0) \vee G(x_0)$	$\vee\mathcal{I}(4)$
6	$\forall x.G(x)$	hyp, $\vee\mathcal{E}$
7	$G(x_0)$	$\forall\mathcal{E}(6)$
8	$F(x_0) \vee G(x_0)$	$\vee\mathcal{I}(7)$
9	$F(x_0) \vee G(x_0)$	$\vee\mathcal{E}(1, 3-5, 6-8)$
10	$\forall x.(F(x) \vee G(x))$	$\forall\mathcal{I}(2-9)$

In the above proof, we first reason backwards, using $\forall\mathcal{I}$ and then forwards using $\vee\mathcal{E}$. We now rewrite the proof above in the Isar proof language. A proof in Isar can be either compound (**proof**-**qed**) or atomic (**by**). A typical proof skeleton looks like this:

proof
  assume *assms*
  have ... **by** $(method)$
  $\vdots$
  have ... **by** $(method)$
  show *concl* **by** $(method)$
qed

This proves the conjecture $assms \Longrightarrow concl$. The intermediates **haves** are only there to bridge the gabp between the assumption and the conclusion and do not contribute to the theorem being proved. On the other hand, **show** establishes the conclusion of the theorem. A *method* is a proof method and it is an argument to the command **by**. Proof methods includes *rule* (which performs a backwards step with a given rule, unifying the conclusion of the rule with the current subgoal and replacing the subgoal by the premises of the rule), *simp* (for simplification), *auto* (for automation) and *blast* (for predicate calculus reasoning).

First, we can state our conjecture as a theorem by means of the **assumes**-**shows** elements of the language which allow direct naming of premises. Several premises could be introduced using the keyword **and**. Note that in Isabelle, quantifiers have low priority so we must use brackets in the premise the symbolize the corresponding premise of our original conjecture.

theorem
assumes Pr:$(\forall\ x.\ F\ x) \vee (\forall\ x.\ G\ x)$
shows $\forall\ x.\ F\ x \vee G\ x$

Note that the pair **assumes**-**shows** corresponds to the implication $\Longrightarrow$ at the meta-level. Thus, we could have written the above conjecture as $(\forall x.\ F\ x) \vee (\forall x.\ G\ x) \Longrightarrow \forall x.\ F\ x \vee G\ x$, but then we would not be able to name the premise. We now apply the introduction rule for the universal quantifier, which is an argument for the command **proof**. At this point, we are reasoning backwards.

proof (*rule allI*)

The proof state now reads

1. $\bigwedge x.\ F\ x \vee G\ x$

which means that for an arbitrary x we have to show $F\ x \vee G\ x$. So we fix our local variable $x0$ and state that the new goal now is $F\ x0 \vee G\ x0$.

fix $x0$
show $F\ x0 \vee G\ x0$

The pair **fix**-**show** corresponds to $\bigwedge$, the universal quantifier at the meta-level. We now have to use disjunction elimination

on the premise $(\forall x.\ F\ x) \vee (\forall x.\ G\ x)$. The rule DISJE in Isabelle is written as $[\![P \vee Q;\ P \Longrightarrow R;\ Q \Longrightarrow R]\!] \Longrightarrow R$ and its application to the premise $(\forall x.\ F\ x) \vee (\forall x.\ G\ x)$, expressed as *disjE*[*OF Pr*], results in $[\![\forall x.\ F\ x \Longrightarrow R;\ \forall x.\ G\ x \Longrightarrow R]\!] \Longrightarrow R$. The two cases are then considered below.

proof (*rule disjE*[*OF Pr*])
  assume $\forall x.\ F\ x$
  from *this* **have** $F\ x0$ **by** (*rule spec*)

The special fact called "**this**" always refers to the last result proved or assumed. The argument for the command **by** must be a proof method. The *spec* rule is the rule for universal elimination and is expressed as $\forall x.\ P\ x \Longrightarrow P\ x$. The goal is solved by an application of disjunction introduction $P \Longrightarrow P \vee Q$.

from *this* **show** $F\ x0 \vee G\ x0$ **by** (*rule disjI*1)

The two subgoals corresponding to disjuction elimination are separated by the keyword **next** and the proof of the next case is analogous to the previous one. The only difference is that we use the abbreviation **then** which stands for the expression "**from this**".

next
  assume $\forall x.\ G\ x$
  then have $G\ x0$ **by** (*rule spec*)
  then show $F\ x0 \vee G\ x0$ **by** (*rule disjI*2)
qed
qed

B. Semantical Properties in Isar

In the following we show some examples of proofs in Isar that are relevant to the semantics of the language discussed in this paper.

The following lemma establishes that the functional Ψ associated to the loop **repeat** is indeed monotonic. The proof for the functional Φ is analogous.

lemma *Psi_mono*: *mono* $(\Psi\ b\ c)$
  unfolding *mono_def*
  proof (*intro allI impI*)

The **unfolding** command operates directly on the current facts and goal by applying equalities. So in this case, it substitutes the expression *mono* $(\Psi\ b\ c)$ for its definition $\forall x\ y.\ x \subseteq y \longrightarrow \Psi\ b\ c\ x \subseteq \Psi\ b\ c\ y$. After the application of introduction rules for the universal quantifier $(\bigwedge x.\ P\ x) \Longrightarrow \forall x.\ P\ x$ and implication $(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$, we are let with the following goal

1. $\bigwedge x\ y.\ x \subseteq y \Longrightarrow \Psi\ b\ c\ x \subseteq \Psi\ b\ c\ y$

The command **let** introduces an abbreviation for a term. Thus *?S* is a shortened form for the corresponding expression. The remaining of the proof of the lemma is easy to follow.

let *?S* $= \{(s, t).\ s = t \wedge\ b\ s\}$
fix *s*0::(*state* $\times$ *state*)*set* **and** *s*1::(*state* $\times$ *state*)*set*
assume *hyp*:$s0 \subseteq s1$
show $\Psi\ b\ c\ s0 \subseteq \Psi\ b\ c\ s1$
  proof (*rule subsetI*)
    fix *x*0::*state*$\times$*state*
    assume $x0 \in \Psi\ b\ c\ s0$
    then have
     $x0 \in\ c\ O\ (?S \cup \{(s, t).\ (s, t) \in s0 \wedge \neg\ b\ s\})$
       by (*simp add*:Ψ*_def*)
    then have
     $x0 \in\ c\ O\ (?S \cup \{(s, t).\ (s, t) \in s1 \wedge \neg\ b\ s\})$
       using *hyp* **by** *auto*
    then show $x0 \in \Psi\ b\ c\ s1$ **using** Ψ*_def* **by** *simp*
  qed
qed

The following proof formalizes the informal reasoning we gave in the proof of Knaster-Tarski lemma in II.3. It is a good example of the power and expressiveness of Isar. We attach several comments in the proof in order to improve readability.

lemma *Knaster_Tarski*:
fixes f::$'a$::*complete_lattice* $\Rightarrow$ $'a$
assumes *mono*:$\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y$
shows $f(\bigsqcap\ \{x.\ f(x) \leq x\}) = \bigsqcap\ \{x.\ f(x) \leq x\}$
 (**is** $f\ ?k = ?k$)

The general strategy is to prove the equality by first showing the inequalities $f\ (\bigsqcap\{x \mid f\ x \leq x\}) \leq \bigsqcap\{x \mid f\ x \leq x\}$ and $\bigsqcap\{x \mid f\ x \leq x\} \leq f\ (\bigsqcap\{x \mid f\ x \leq x\})$ and then invoking antysimmetry. The expression

shows *formula* (**is** *pattern*)

matches the pattern agains the formula, thus instantiating here the unknown variable $?k$ and binding it to the expression $\bigsqcap\{x \mid f\ x \leq x\}$ for latter use in the proof. The annotation $'a$ states, roughly speaking, that any type $'a$ must obey the interface specification of the type class *complete_lattice* (see [5] for further details).

proof $-$

The *hyphen* "$-$" method as an argument for **proof** has the purpose of leaving the goal unchanged. Now, the first part of the proof in Isar goes as follows:

have *first*:$f\ ?k \leq ?k$ (**is** $\_ \leq \bigsqcap\ ?P$)
proof (*rule Inf_greatest*)
  fix $x0$ **assume** $x0 \in ?P$
  then have $?k \leq x0$ **by** (*rule Inf_lower*)
  from *this*
    have $f(?k) \leq f(x0)$ **by** (*rule mono*)
  from ‹$x0 \in ?P$› **have** $f(x0) \leq x0$
    by (*simp only*:*mem_Collect_eq*)
  from *this* **and** ‹$f(?k) \leq f(x0)$›
    show $f(?k) \leq x0$ **by** (*rule dual_order.trans*)
qed

We now look at this compound proof above in more detail. Firstly, we start the proof of $f\ (\bigsqcap\{x \mid f\ x \leq x\}) \leq \bigsqcap\{x \mid f\ x$

$\leq x\}$ using the introduction rule INF_GREATEST ($\bigwedge x0.\ x0 \in A \Longrightarrow z \leq x0) \Longrightarrow z \leq \bigsqcap A$ where $A = \{x \mid f\,x \leq x\}$ and $z = f\ (\bigsqcap \{x \mid f\,x \leq x\})$. This rule says that to prove that $z \leq \bigsqcap A$, it suffices to assume that $x0 \in A$ and to show that $z \leq x0$, for an arbitrary value $x0$. This follows directly by monotonicity $x \leq y \Longrightarrow f\,x \leq f\,y$, the rule

$$\frac{x0 \in \{x \mid P(x)\}}{P(x0)}\ \text{MEM\_COLLECT\_EQ}$$

where $P(x)$ is $\{x \mid f\,x \leq x\}$ and transitivity $[\![b \leq a;\ c \leq b]\!] \Longrightarrow c \leq a$. The second part of the proof is similar to the previous one:

have *second*: *?k* $\leq f(?k)$
proof (*rule Inf_lower*)
from ‹$f(?k) \leq ?k$›
have $f(f\ ?k) \leq f(?k)$ **by** (*rule mono*)
then show $f\ ?k \in ?P$ **by** (*simp only*:*mem_Collect_eq*)
qed

Note that now we start the proof of $\bigsqcap \{x \mid f\,x \leq x\} \leq f\ (\bigsqcap \{x \mid f\,x \leq x\})$ using the introduction rule INF_LOWER $x \in A \Longrightarrow \bigsqcap A \leq x$ where as before, $A = \{x \mid f\,x \leq x\}$. Moreover, we also use the previous result, i.e., that $f\ (\bigsqcap \{x \mid f\,x \leq x\}) \leq \bigsqcap \{x \mid f\,x \leq x\}$ by quoting it in the beginning of the proof. Using '**from** first' would also suffice. The desired result follows now from the two inequalities and antisymmetry $[\![x \leq y;\ y \leq x]\!] \Longrightarrow x = y$.

from *first* **and** *second* **show** *f ?k = ?k*
by (*rule antisym*)
qed

In this last proof we show that a **repeat** loop can be expressed with sequential composition and the conditional command.

lemma *C_Repeat_If*:
$[\![$*repeat c until b*$]\!]$
$=$ $[\![$*c;;if b then SKIP else repeat c until b fi*$]\!]$
proof −
let *?r* = *repeat c until b*
let *?f* = Ψ (*bval b*) $[\![c]\!]$
have *eq*1:$[\![?r]\!]$ = *lfp ?f* **by** (*simp only*:*repeat*)
from *Psi_mono*
have *eq*2:*lfp ?f* = *?f* (*lfp ?f*) **by** (*rule lfp_unfold*)
from *eq*1 **and** *eq*2
have *eq*3:$[\![?r]\!]$= *?f* (*lfp ?f*) **by** (*rule trans*)
from Ψ*_def* **have**
?f (*lfp ?f*) = $[\![$*c;;if b then SKIP else ?r fi*$]\!]$ **by** *simp*
from *eq*3 **and** *this* **show** *?thesis* **by** (*rule trans*)
qed

Some brief remarks about the above proof should suffice. The command **let** introduces an abbreviation for a term. Thus both *?r* and *?f* are a shortened form for the corresponding expressions. In Isabelle, the unfolding of the least fixed point rule is expressed as *mono f* $\Longrightarrow$ *lfp f* = *f* (*lfp f*) while the rule TRANS is simply $[\![r = s;\ s = t]\!] \Longrightarrow r = t$. The unknown variable *?thesis* is matched against any goal stated by **lemma**, **theorem** or **show**. Thus the expression *?thesis* actually denotes the goal stated in the lemma.

VII. Concluding Remarks

Formal semantics is one of the most important areas of theoretical computer science. It provides a solid foundation for the design of programming languages as well as a underlying mathematical theory with which we can prove properties of programs written in these very programming languages. Also, we believe that proof assistants and theorem provers will become the future of mathematics, where definitions, statements, computations and proofs are all available in a computerized form, where they can be formally verified in a mechanical way. Hence, in this paper we have developed a relational denotational semantics of a small imperative language and formalized its syntax and semantics with the Isabelle/HOL proof assistant. On top of this formalization and on top of the expressiveness of the high-level Isar proof language, we provided the reader with a number of non-trivial, highly readable and structured proofs of important properties in the framework of natural deduction.

Acknowledgments

I am especially grateful to Uwe Wolter and Edward H. Häusler for reading carefully early drafts of this paper. Their suggestions were most helpful.

References

[1] Y. Bertot and P. Castran, *Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[2] A. Bove, P. Dybjer, and U. Norell, "A Brief Overview of Agda - A Functional Language with Dependent Types," in *TPHOLs*, 2009, pp. 73–78.

[3] H. Geuvers, "Proof Assistants: History, Ideas and Future," *Sadhana Journal*, vol. 34, pp. 3–25, 2009.

[4] M. Gordon, "Twenty Years of Theorem Proving for HOLs Past, Present and Future," in *TPHOLs*, 2008, pp. 1–5.

[5] F. Haftmann, "Haskell-style type classes with Isabelle/Isar," http://isabelle.in.tum.de/dist/Isabelle2013-2/doc/classes.pdf, November 2013.

[6] T. Nipkow, "Programming and Proving with Isabelle/HOL," http://isabelle.in.tum.de/dist/Isabelle2013-1/doc/prog-prove.pdf, November 2013.

[7] T. Nipkow and G. Klein, "Concrete Semantics - A Proof Assistant Approach," http://www21.in.tum.de/~nipkow/Concrete-Semantics/concrete_semantics.pdf, November 2014.

[8] L. C. Paulson, *ML for the Working Programmer (2. ed.)*. Cambridge University Press, 1996.

[9] L. P. Tobias Nipkow and M. Wenzel, "Isabelle/HOL – A Proof Assistant for Higher-Order Logic," in *Lectute Notes in Computer Science, vol. 2283*. Springer-Verlag, 2002.

[10] M. Wenzel, "Isabelle/Isar – a versatile environment for human-readable formal proof documents," Ph.D. dissertation, Institüt für Informatik, Technische Universität München, 2002.

[11] G. Winskel, *The Formal Semantics of Programming Languages - An Introduction*, ser. Foundation of computing series. MIT Press, 1993.