Programming language specification and implementation

Peter Sestoft[0000−0002−5843−6021]

Computer Science Department, IT University of Copenhagen, Denmark
sestoft@itu.dk

Abstract. The specification of a programming language is a special case of the specification of software in general. This paper discusses the relation between semantics and implementation, or specification and program, using two very different languages for illustration. First, we consider small fragments of a specification of preliminary Ada, and show that what was considered a specification in VDM in 1980 now looks much like an implementation in a functional language. Also, we discuss how a formal specification may be valuable even though seen from a purely formal point of view it is flawed. Second, we consider the simple language of spreadsheet formulas and give a complete specification. We show that nondeterminism in the specification may reflect run-time nondeterminism, but also underspecification, that is, implementation-time design choices. Although specification nondeterminism may appear at different binding-times there is no conventional way to distinguish these. We also consider a cost semantics and find that the specification may need to contain some "artificial" nondeterminism for underspecification.

Keywords: Specification · Implementation · Programming languages · Nondeterminism · Underspecification.

1 Introduction

This paper investigates the relation between specification (or modeling) and programming in the special case where the specification describes a programming language and its implementations, rather than some other software artefact. We study two examples of programming language specification to make several observations.

We first use a large but incomplete specification (of an Ada language subset) to make two observations. One observation is that the 1980 VDM formal specification looks very much like a functional program today. This indicates that the distinction between specification and programming may be one of degree, contingent on (specification and programming) language sophistication and notational support for the use of sets, maps, sequences and other useful structures. The other observation is that although the VDM specification is incomplete and has some formal deficiencies, it enabled its authors to subsequently develop a

validated compiler for full Ada. Hence the value of an attempted formal specification may lie as much in the knowledge that its authors acquire through developing the specification as in the resulting formal text itself.

We next use a small but complete example specification (of simple spreadsheet formulas) to discuss the role of nondeterminism in specifications. We observe that specification nondeterminism may reflect desired run-time nondeterminism in the implementation; or it may reflect language aspects left unspecified and hence open to implementation-time choices; or both. The first case is illustrated by the spreadsheet `RAND` function which is expected to produce a new pseudorandom number at each evaluation. The second case is illustrated by unspecified argument evaluation order in function calls, where a sequential implementation may choose left-right or right-left or any other fixed order. The third case is illustrated by the run-time nondeterminism exhibited by truly parallel implementations of argument evaluation order. We note that there is no standard way to describe the intended binding-time (run-time or implementation-time) of specification nondeterminism. We further observe that a richer instrumented semantics eg. accounting also for the cost of expression evaluation may need to have additional nondeterminism so as not to overconstrain the possible implementations.

2 Example: A Formal Description of Ada

In this section we study the book "Towards a Formal Description of Ada" [3] from 1980 and present some reflections on it. That book uses the VDM specification language [2] to give a formal description of a preliminary version of the Ada programming language, at the time a novel, advanced and complex but rather well-designed language. The bulk of the 630-page book is based on MSc theses written by five software students (Bundgaard, Schultz, Pedersen, Løvengreen, Dommergaard) at the Technical University of Denmark, under the supervision of Dines Bjørner and Hans Bruun. It represents an impressive amount of work.

We make the following observations:

- The formal description of Ada from 1980 can today be considered a (very large) functional program, an implementation of the language. We discuss this in Sec. 2.1 and show some concrete examples of a transliteration of the VDM specification into the F# programming language [23].
- From a purely formal viewpoint, the Ada description in [3] is incomplete and inconsistent, and most likely wrong in several details. Some functions are not defined, names are misspelt, types are wrong, and so on. In Sec. 2.2 we show and discuss some examples.
- However, to consider the specification a failure would be to completely misunderstand its significance and utility. Some of the authors of the specification went on to develop a full-fledged Ada compiler, the first European one to be validated, and founded the company DDC International, now DDC-I, to sell it [4]. We discuss this in Sec. 2.3.

2.1 The Ada Formal Description as a Functional Program

The 1980 formal description of Ada consists of VDM *domain definitions*, which are type definitions in modern functional programming terms, and VDM *formulae*, which are function definitions. The specification consists of hundreds of such formulae, most of which are admirably short and comprehensible, representing a careful factorization of the many interacting aspects of the Ada language.

In this section we consider a few fragments (around two percent) of the VDM description of the Ada static semantics and show how some of the advanced constructs used in the VDM formulae can be expressed in a modern functional programming language. Specifically, we express them in F#, developed at Microsoft Research Cambridge UK [23], but we might have used the Haskell or Scala languages instead. Historically, F# descends from the ML programming language which was created in the 1970es and whose first description was published in 1979 [9]. It is possible that the designs of VDM and of ML may have inspired each other at the time, and hence not necessarily a surprise that VDM can be translated into F#.

Nevertheless, the translation supports our point of view that what was considered a specification in 1980 can be considered an implementation in 2018, thanks to advances in programming language design and implementation.

For space reasons we present only fragments of the VDM formulae, and no domain definitions, so we do not expect the reader to understand all details of the examples.

Figs. 1 and 2 show how VDM record update, record field selection, pattern matching and lambda expressions can be expressed in F#.

Figs. 3 and 4 show how VDM list comprehension and iteration over indices can be expressed elegantly using F# sequence expressions, and how F#'s nested pattern matching can be used to good effect.

Figs. 5 and 6 show how the VDM combination of choice of an element, recursion, and production (or not) of a value can be expressed using F# sequence expressions.

Figs. 7 and 8 show how the VDM nondeterministic construction "such that" (`s.t.`), which here simply produces a list without duplicates, and be expressed in F# using its `Set` module in combination with sequence expressions.

2.2 Some Flaws in the 1980 Formal Description of Ada

This section shows that from a purely formal point of view, the Ada description in [3] has some flaws. The purpose is not to blame the authors of that specification, who did an admirable job given the tools of the time; there is little value in finding hairs in the soup four decades later. The VDM specification was developed without tool support, so even misspelt variable names and type names, wrong argument order, wrong declared return type of a function, and similar fairly trivial mistakes would have to be discovered by human proofreading. Given the magnitude of the work, the number of such errors is modest.

One drawback of the Ada formal description is that it is incomplete:

```
.0 lookup-base-type(btype)(sur) =
.1 let ds = sel-ds(btype)(sur) in
.2 btype + (s-ds:-> cases ds:
.3          (mk-Access(fct,compl)->
.4             mk-Access(Ld.lookup-base-type(fct(d))
.5                                (sur+(s-dict:->d)),compl) ,
.6           T -> ds) ,
.7          s-sub:-> cases ds:
.8          (mk-Array()  -> nil,
.9           mk-Record() -> nil,
.10          T           -> s-sub(btype))  )

.11 type: Type -> Surroundings -> Type
```

Fig. 1. Fragment of the Ada static semantics [3, page 148] expressed in VDM. The VDM notation "`btype + ...`" represents record field update, and "`cases ds:`" represents pattern matching. Due to typesetting limitations at the time, a lambda expression $\lambda d.e$ was written as `Ld.e` in line 4. The formula can be expressed in a functional language such as F#, see Fig. 2.

```
let rec lookup_base_type (btype : Type) (sur : Surroundings) : Type =
    let ds = sel_ds(btype)(sur)
    { btype
      with
        s_ds =
          match ds with
            | Access { s_map = fct; s_com = compl } ->
              Some (Access { s_map =
                function d -> lookup_base_type (fct(d)) { sur with s_dict = d };
                              s_com = compl });
        s_sub =
          match ds with
            | Array _ -> Nil
            | Record _ -> Nil
            | _ -> btype.s_sub
    }
```

Fig. 2. An F# version of the VDM formula in Fig. 1. Record update is expressed as "`{ btype with ...}`", and pattern matching as "`match ds with`". A constructor such as `Array` is written without the `mk-` prefix that is customary in VDM.

```
.0 lookup-all-name(name)(sur) =
.1 let mk-All-name(name') = name              in
.2 let descr-list = lookup-name(name')(sur) in
.3 conc <cases descr-list[i]:
.4   (mk-Obj-descr(typ, ) ->
.5       cases sel-ds(typ)(sur) :
.6       ( mk-Access(fct) -> <mk-Obj-descr(fct(s-dict(sur)),nil)>,
.7         T -> <>),
.8    T -> <>) | i ∈ ind descrl>

.9 type: All-name -> Surroundings -> Descr*
```

Fig. 3. Fragment of the Ada static semantics [3, page 194] in VDM. The notation "`< e | i ∈ ind descrl>`" spanning lines 3–8 evaluates VDM expression `e` for each index `i` into sequence `descrl`. (The latter is a misspelling of `descr-list` in line 2). This creates a sequence of zero- or one-element sequences, flattened into one sequence using the `conc` VDM function. This can be expressed in F# as shown in Fig. 4.

```
let lookup_all_name (All_name(name') : All_name) (sur : Surroundings)
                  : seq<Descr> =
    seq { for di in lookup_name(name')(sur) do
          match di with
          | Obj_descr { s_tp = { s_ds = Some (Access {s_map = fct }) } }
              -> yield Obj_descr { s_tp = fct(sur.s_dict); s_con = None }
          | _ -> ()
        }
```

Fig. 4. An F# version of the VDM formula in Fig. 3. An F# sequence expression "`seq { ...}`", similar to list comprehensions in Haskell and to mathematical set comprehensions, neatly expresses the more complicated construct in the original's lines 3-8. Also, the original's nested `cases` expressions can be expressed concisely by nested patterns in the argument to the `Obj_descr` constructor match.

```
.0 get-available-obj-list(parms)(pset)(sur)=
.1 if pset = { } then <> else
.2 let d E pset in
.3 let mk-Subprgr-descr( ,,entrance) = d   in
.4 ((if s-return(entrance) ≠ nil and
         parameter-checker(parms)(entrance)(sur)
.5       then <mk-Obj-descr(s-return(entrance),CONSTANT)>
.6       else <> )) ^
.7  get-available-obj-list(parms)(pset\d)(sur) ) )

.8 type: Act-parm* -> ( Subprgr-descr)-set ->Sur -> Obj-descr*
```

Fig. 5. Fragment of the Ada static semantics [3, page 195] expressed in VDM. This VDM formula uses recursion to process the elements of the set `pset` one at a time, producing a zero- or one-element sequence for each, and concatenating the results into a single sequence using sequence concatenation ($\frown$). The formula can be expressed concisely in F# as shown in Fig. 6.

```
let get_available_obj_list (parms : seq<Act_parm>) (pset : Set<Subprgr_descr>)
                           (sur : Surroundings) : seq<Descr> =
    seq { for Subprgr_descr(_, _, entrance) in pset do
            match entrance.s_return with
              | Some typ when parameter_checker(parms)(entrance)(sur) ->
                  yield Obj_descr { s_tp = typ; s_con = Some CONSTANT }
              | None -> ()
        }
```

Fig. 6. An F# version of the VDM formula in Fig. 5. In the sequence expression, "`for ... in pset`" combines the nondeterministic choice of `d` (original line 2) with decomposition (original line 3). The `match` expression with side condition (`when`) neatly expresses the conditional and selection in the original lines 4–5.

```
.0  extract-name-types( name ) (sur) =
.1  let descrl = lookup-name(name) (sur) in
.2  conc < cases descrl[i]:
.3          (  mk-Obj-descr(otype, )  -> <otype> ,
.4             mk-Overload-descr(ds)  ->
.5                ( let list Ɛ Type* be.s.t
.6                     card elems list = ln list and
.7                     elems list = {ltype |
                                    mk-Literal-descr(ltype)Ɛ ds }) in
.8                  list)                          ,
.9             mk-Number(val)          -> <mk-Pseudotype(val)> ,
.10            T                       -> <> )
.11     | i Ɛ ind descrl>

.12 type: Name -> Surroundings -> ( Type | Pseudotype )*
```

Fig. 7. Fragment of the Ada static semantics [3, page 197] expressed in VDM. The "`conc < ...| i` *in* `ind descrl>` in lines 2-11 works as in Fig. 3 to create a sequence of Types or Pseudotypes from a sequence of descriptions. Lines 5-8 nondeterministically choose a `list` of Types such that (`s.t.`) it has no duplicates and it has one element for each type appearing in a Literal-descr in the sequence `ds` of descriptions. This can be expressed equally concisely in F# as shown in Fig. 8.

```
let extract_name_types (name : Name) (sur : Surroundings)
                    : seq<TypeOrPseudoType> =
    let descrl = lookup_name(name)(sur)
    seq { for di in descrl do
            match di with
            | Obj_descr { s_tp = otype } -> yield (TOPT_Type otype)
            | Overload_descr(ds) ->
                yield! Set.ofSeq(Seq.choose
                  (function | Literal_descr(ltype) -> Some (TOPT_Type(ltype))
                            | _ -> None) ds)
            | Number(v) -> yield TOPT_Pseudotype(v)
            | _ -> ()
        }
```

Fig. 8. An F# version of the VDM formula in Fig. 7. The outer sequence expression corresponds to lines 2–11 in the original. The construct `Set.ofSeq(Seq.choose(...))` corresponds to the nondeterministic choice of `list` in the original's lines 5–8. The type `TypeOrPseudoType` and constructors such as `TOPT_Type` are artefacts of F#'s type system being more picky about subtypes than VDM is.

- First of all, the formal description covers a subset of preliminary Ada called A6, and does not claim to be complete. The static semantics chapter has a list of "aspects of Ada not covered" [3, page 131], and several sections called "Missing functions" [3, pages 176, 185, 190, 202].
- The dynamic semantics chapter has a list of "functions used in this model but not defined" [3, page 305].

Also, when investigating the formulae shown in Sec. 2.1 we came across a few mistakes:

- Formula `lookup-unitname` [3, page 148], not shown here, is declared to have return type `Dict`, but the correct type is `Descr`.
- Formula `parameter-checker` [3, page 163], not shown here, is declared to have parameter order `entrance actual-parm-list sur` but in formula `get-available-obj-list`, shown in Fig. 5, it is called with the opposite argument order `parms entrance sur`.
- Some constructors and types are spelled inconsistently, for instance `Simp_name` versus `Simple_name` [3, pages 139 and 191], as well as `Sub_prgr_descr` versus `Subprgr_descr` [3, pages 141 and 195].
- Some local variables are spelled inconsistently, for instance `descr_list` versus `descrl`, as shown in Fig. 3.

Flaws such as those listed above are of course easily fixed, but from a purely formal point of view they show that the description is not a consistent formal object. Moreover, given what we know about human fallibility, it is reasonable to assume that in addition to these superficial inconsistencies there are more substantial, semantically important, mistakes in the description.

The next section argues that the specification is valuable and useful anyway.

2.3 Formal Specification, Valuable Despite Formal Flaws

This section argues that a "formal" specification may be valuable and useful even though, from a purely formal point of view, it is incomplete and even inconsistent. One indication is that, thirty-eight years after the publication of [3], the DDC-I Ada compiler and the DDC-I company still exist [25], in contrast to most competitors from that time. It is clear that to develop the specification in [3] the authors had to scrutinize the informal description of Ada, develop and discuss illustrative examples, and so on, and thereby gained a deep understanding, invaluable when subsequently developing the Ada compiler.

One of the authors, Hans Henrik Løvengreen, said: "It has been shown how the Ada Compiler [...] can be systematically derived from the formal definition. The idea is that problems should be revealed and solved at the abstract level, such that the implementation will be straightforward." [3, page 318].

Also, a 1988 US Institute for Defense Analyses assessment says "[. . .] the formal definition was not mechanically transformed into a compiler. Rather, the formal definition was used by the compiler writers as the reference instead of a

natural language requirements document. DDC personnel with whom we have spoken claim that the extra up-front effort taken to first formalize the definition of Ada led to efficiences in the long run." [19, page 8].

Hence the chief value of a formal specification may be that the very work of developing it forces and motivates the authors to immerse themselves in the domain (whether a programming language or an application) and its intricacies, thereby building a comprehensive mental model of it. This viewpoint was suggested by Naur in 1985 concerning programming: "programming in this sense must be the programmers' building up knowledge of a certain kind, knowledge taken to be basically the programmers' immediate possession, any documentation being an auxiliary, secondary product" [17, page 253]. Replacing "programming" with "writing a formal specification", the view would be that much of the value of writing a formal specification lies in "building up knowledge" in the minds of the specification's authors, and that the resulting formal text or model may actually not be the most significant outcome of this activity. The story of the 1980 Ada description [3] seems to corroborate this view: The value of (attempted) formalization may to a large extent lie in forcing the specification's authors to pay attention to details that would be more easily glossed over if writing (only) in English or another natural language.

Nevertheless, despite its utility in subsequently implementing a validated Ada compiler, it must have been recognized at the time that the Ada description [3] had some shortcomings. Several of its authors became involved in a subsequent 1984–1987 European project to develop a more complete formal specification of Ada [4, Sec. 3.2] [19, Sec. 1.2]. The more complete specification of (revised) Ada resulting from this effort is difficult to locate today, having been published mostly as technical reports. The above-mentioned US assessment laments the complexity of the latter more complete formal specification, the high level of computer science theory background required to understand it, and the poor English of the exposition [19, Sec. 4].

The early, less complete and less formal 1980 Ada specification [3] appears in the end to have been the more useful one.

3 Example: Spreadsheet Semantics

Here we consider a simple "programming language", namely spreadsheet formulas. We use a combination of operational semantics and axiomatic semantics. The former specifies evaluation of the formula in a single spreadsheet cell. The latter specifies the expected consistency of all cells.

3.1 Formulas, Cells and Sheets

In our simplified spreadsheet model, a spreadsheet consists of a grid of cells. Each cell is either blank or contains a formula `=`e where e is an expression as shown in Fig. 9. Each non-blank cell in addition contains its formula's computed value, which is shown to the spreadsheet user.

$$
\begin{array}{lll}
e ::= & \texttt{n} & \text{number constant} \\
| & \texttt{ca} & \text{cell reference} \\
| & \texttt{IF}(e_1,e_2,e_3) & \text{conditional expression} \\
| & \texttt{RAND}() & \text{volatile function} \\
| & \texttt{F}(e_1,\ldots,e_n) & \text{built} - \text{in function call}
\end{array}
$$

Fig. 9. Syntax of the simplified spreadsheet formula language.

3.2 Characteristics of Spreadsheets

Spreadsheets and spreadsheet formulas have some peculiar characteristics:

(a) Consistency after recalculation: A cell's computed value after a recalculation must be the result (or rather, a possible result) of evaluating the cell's formula, given the computed values of all other cells. A reference to a cell such as A2 must have the same value wherever it appears, so `A2=A2` must be true.
(b) Error values: An expression always has a value; there is no notion of exception or failed evaluation. Thus some expressions, such as `1/0` and `ASIN(2)`, must evaluate to error values such as `#DIV/0!` and `#NUM!`.
(c) Error strictness: If an argument to a built-in function evaluates to an error value, then the function call evaluates to that error value.
(d) Volatile functions and cells: Some functions (`RAND()`, `NOW()`) are nondeterministic: each evaluation may produce a different result, so `RAND()=RAND()` may be false. Any cell that involves a nondeterministic function must be recomputed in a recalculation.
(e) Non-strictness of `IF(e1,e2,e3)`: The "then" branch `e2` should not be evaluated unless `e1` evaluates to true; and similarly for the "else" branch `e3`.
(f) Cyclic cell reference dependencies: The evaluation of formula in a cell may refer to the value of that cell itself, directly or indirectly. In that case, no ordinary number value may be found for the cell, but an error value such as `#CYCLE!` may be found instead.
(g) Side effect freedom: The evaluation of a formula has no side effects.

These characteristics are not accidents of design, but essential for the practical utility of spreadsheets. They do have some non-trivial consequences for implementations.

It follows from (a) that the value held in a cell may need to be recalculated after an update to any cell on which it depends, directly or indirectly.

By (d), (e) and (f), if cell A2 contains the formula `=IF(RAND()<0.2,A2+1,42)` then there may or may not be a cyclic dependency of A2 on itself, depending on the value of `RAND()` in this particular recalculation. Hence cycles must be detected during evaluation, not by a preceding topological sorting of cells.

It follows from side effect freedom (g) that it is not observable whether a formula is evaluated once or twice or not at all, or whether it is evaluated before, at the same time as, or after, another formula. This allows an implementation

to choose evaluation order (sequential or parallel), evaluate a cell zero times (reuse cached cell value), once (when precedents are up to date), or multiple times (evaluate it speculatively, maybe in parallel on multiple processors). It also allows cell areas with copy-equivalent formulas to be replaced by map-reduce style bulk array operations [1].

3.3 Formal Evaluation Semantics

The formal semantics of spreadsheet evaluation given here is from our book [22, Sec. 1.8]. It uses operational semantics [14, 18] to specify the local evaluation of each cell's formula (Sec. 3.3) and an axiomatic semantics to specify the global consistency of a spreadsheet after a recalculation (Sec. 3.4).

This gives fine control over formula evaluation, accounting for spreadsheet characteristics (b) through (e) in Sec. 3.2, while leaving completely unconstrained the recalculation mechanism required to obtain spreadsheet characteristics (a) consistency of the recalculated spreadsheet and (f) detection of reference cycles.

Operational Semantics of Expressions We describe a spreadsheet's formulas using a map $\phi : Addr \to Expr$ so that when $ca \in Addr$ is a cell address, $\phi(ca)$ is the formula in cell ca. If cell ca is blank, then $\phi(ca)$ is undefined. The domain $dom(\phi)$ of ϕ is the set of non-blank cells. The ϕ function is not affected by recalculation, only by editing the formulas in the spreadsheet.

We describe the evaluation of expressions (Fig. 9) using the semantic sets and functions in Fig. 10. For instance, $Value = Number + Error$ is the set of values, and $Addr$ contains cell addresses ca such as B2.

We describe the result of a recalculation by a function $\sigma : Addr \to Value$ so that $\sigma(ca)$ is the computed value in cell ca. The σ function gets updated by each recalculation and must satisfy consistency requirements described in Sec. 3.3.

$$
\begin{array}{lll}
n \in Number & = \{ \text{ proper numbers } \} \\
\quad Error & = \{ \texttt{\#NUM!}, \texttt{\#DIV/0!}, \texttt{\#CYCLE!} \} \\
ca \in Addr & = \{ \text{ cell addresses } \} \\
v \in Value & = Number + Error \\
e \in Expr & = \{ \text{ formulas, see Fig. 9 } \} \\
\phi & \in Addr \to Expr \\
\sigma & \in Addr \to Value
\end{array}
$$

Fig. 10. Sets and maps used in the spreadsheet semantics: $Number$ is the set of proper floating-point numbers, excluding NaNs and infinities; $Error$ is the set of error values; $Addr$ the set of cell addresses; $Value$ the set of values (either number or error); and $Expr$ the set of formulas.

We describe the evaluation of an expression e by an evaluation judgment of the form $\sigma \vdash e \Downarrow v$, which says: When σ describes the calculated values of all

cells, then formula e may evaluate to value v. The "may" is important because, in general, an expression may evaluate to multiple different values. For instance, `RAND()` may evaluate to any number between 0.0 (included) and 1.0 (excluded). Hence, `7+1/RAND()` may evaluate to some number greater than 8 or to the error value `#DIV/0!` in case `RAND()` produces 0.0.

The complete set of inference rules that describe when a formula evaluation judgment $\sigma \vdash e \Downarrow v$ holds are given in Fig. 11.

$$\frac{}{\sigma \vdash \mathtt{n} \Downarrow n}\ (e1)$$

$$\frac{ca \notin dom(\sigma)}{\sigma \vdash \mathtt{ca} \Downarrow 0.0}\ (e2b)$$

$$\frac{ca \in dom(\sigma) \qquad \sigma(ca) = v}{\sigma \vdash \mathtt{ca} \Downarrow v}\ (e2v)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \in Error}{\sigma \vdash \mathtt{IF}(e_1,e_2,e_3) \Downarrow v_1}\ (e3e)$$

$$\frac{\sigma \vdash e_1 \Downarrow 0.0 \qquad \sigma \vdash e_3 \Downarrow v}{\sigma \vdash \mathtt{IF}(e_1,e_2,e_3) \Downarrow v}\ (e3f)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \qquad v_1 \neq 0.0 \qquad \sigma \vdash e_2 \Downarrow v}{\sigma \vdash \mathtt{IF}(e_1,e_2,e_3) \Downarrow v}\ (e3t)$$

$$\frac{0.0 \leq v < 1.0}{\sigma \vdash \mathtt{RAND}() \Downarrow v}\ (e4)$$

$$\frac{\sigma \vdash e_i \Downarrow v_i \in Error}{\sigma \vdash \mathtt{F}(e_1,\ldots,e_n) \Downarrow v_i}\ (e5e)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \notin Error \qquad \ldots \qquad \sigma \vdash e_n \Downarrow v_n \notin Error}{\sigma \vdash \mathtt{F}(e_1,\ldots,e_n) \Downarrow f(v_1,\ldots,v_n)}\ (e5v)$$

Fig. 11. Evaluation rules for simplified spreadsheet formulas. From [22].

The formula evaluation rules in Fig. 11 may be explained as follows:

- Rule (e1) says that a number constant `n` evaluates to that constant's value.
- Rule (e2b) says that a reference ca to a blank cell evaluates to 0.0.
- Rule (e2v) says that a reference ca to a non-blank cell evaluates to the value $\sigma(ca)$ calculated for that cell. This value may be a number or an error.
- Rule (e3e) says that the expression $\mathtt{IF}(e_1,e_2,e_3)$ may evaluate to error v_1 if the condition e_1 may evaluate to error v_1.

- Rule (e3f) says that `IF`(e_1,e_2,e_3) may evaluate to v provided the condition e_1 may evaluate to zero and the "false branch" e_3 may evaluate to v.
- Rule (e3t) says that `IF`(e_1,e_2,e_3) may evaluate to v provided e_1 may evaluate to some non-zero number v_1 and the "true branch" e_2 may evaluate to v.
- Rule (e4) says that function call `RAND`() may evaluate to any number v greater than or equal to zero and less than one. Hence, this rule models nondeterministic choice.
- Rule (e5e) says that a call `F`($e_1, \ldots, e_n$) to a built-in function `F` may evaluate to error v_i if one of its arguments e_i may evaluate to error v_i. If more than one argument may evaluate to an error, then the function call may evaluate to any of these. Hence, the semantics does not prescribe an evaluation order for arguments, such as a left to right, or right to left, or all in parallel.
- Rule (e5v) says that a call `F`($e_1, \ldots, e_n$) to a function `F` may evaluate to value v if each argument e_i may evaluate to non-error value v_i, and applying the actual function f to arguments $(v_1, \ldots, v_n)$ produces value v. The final result v may be a number such as 5, for instance, if the call is +(`2`, `3`); or it may be an error such as `#DIV/0!`, for instance, if the call is `/(1.0, 0.0)`.

There are five groups of rules (e1), (e2x), (e3x), (4), (e5x), in Fig. 11, each corresponding to one of the five kinds of formulas in Fig. 9. One can easily write a program whose five cases of the `match` correspond exactly to the five groups of rules; see the F# program in Fig. 12: the distance from specification (operational semantics in Fig. 11) to program (implementation) is short.

However, the various appearences of nondeterminism in the specification have been treated differently in the implementation. Whereas the (e4) `RAND` rule's nondeterminism has been explicitly retained in the interpreter, the (e5e) rule's nondeterminism has been quietly eliminated through the use of the F# `tryFind` function, which searches a list sequentially from the head for a value that satisfies the predicate `isError`.

It is not at all clear from the specification (Fig. 11) whether nondeterminism in a given rule is essential and must be retained in an implementation (as in e4), or whether it is merely underspecification intended to provide some implementation freedom (as in e5e).

3.4 Axiomatic Semantics of Recalculation

The previous subsection describes how to evaluate a formula, given values (via σ) of all cells in the worksheet. Now we can describe the requirements on a recalculation: It must find a value for every non-blank cell ca in the sheet, and that value $\sigma(ca)$ must be a possible result of evaluating the formula $\phi(ca)$ in that cell. These consistency requirements on a recalculation are stated in Fig. 13.

These requirements leave it completely unspecified how a spreadsheet recalculation works: whether it recalculates all cells or only some cells; whether it calculates a cell only once or multiple times; whether it does so sequentially, and if so in what order, or in parallel; whether it guesses the values or computes

```
let rec eval (sigma : env) (e : expr) =      // Rule:
    match e with
    | Const d -> Num d                       // e1
    | CellRef (c,r) ->
        match Map.tryFind (c,r) sigma with
        | None   -> Num 0.0                  // e2b
        | Some v -> v                        // e2v
    | If (e1, e2, e3) ->
        let v1 = eval sigma e1
        match v1 with
        | Error _ -> v1                      // e3e
        | Num 0.0 -> eval sigma e3           // e3f
        | Num _   -> eval sigma e2           // e3t
    | Rand -> Num (random.NextDouble())      // e4
    | Func (f, es) ->
        let vs = List.map (eval sigma) es
        match List.tryFind isError vs with
        | Some vi -> vi                      // e5e
        | None    -> evalBuiltin f vs        // e5v
```

Fig. 12. An interpreter (in F#) for simple spreadsheet expressions, closely following the operational semantics in Fig. 11. The left-hand side of (`->`) in a `match` case corresponds to the conclusion of a rule group, and the right-hand side's conditions and recursive calls correspond to rule premises.

$$
\begin{array}{l}
(1)\ dom(\sigma) = dom(\phi) \\
(2)\ \forall ca \in dom(\phi).\ \sigma \vdash \phi(ca) \Downarrow \sigma(ca)
\end{array}
$$

Fig. 13. The consistency requirements, or axioms, for spreadsheet recalculation. Requirement (1) says that a recalculation must find a value $\sigma(ca)$ for every non-blank cell ca. Requirement (2) says that the computed value $\sigma(ca)$ must agree with the cell's formula $\phi(ca)$. Considered as a "definition" of σ it is circular in that σ appears both on the left of the ($\vdash$) and on the right. This is necessary, since the evaluation of a formula $\phi(ca)$ may depend on the value $\sigma(ca')$ of any cell ca'.

them; and so on. This underspecification is intentional: it is essential to permit a range of implementation strategies and optimizations.

While it is entirely obvious that formula evaluation can be implemented as specified in Sec. 3.3, it is much less clear how to implement recalculation, and whether it can be implemented as specified. A simple sequential (single-threaded) approach is to equip each non-blank cell with a state that is either Dirty (the initial state), Computing or Uptodate. Then while there is at least one Dirty cell, pick one, change its state to Computing, evaluate its formula, and if successful, set the cell's computed value to the formula's value and its state to Uptodate. A cell reference encountered during evaluation may be handled like this: If the referred-to cell is Uptodate, use its computed value; if it is Dirty, recursively compute its values; and if it is Computing, there is dependency cycle in the spreadsheet. This procedure performs a form of depth-first traversal of the depends-on (or precedents) graph, with cycle detection. However, this describes a mechanism, not a specification, and is heavily biased towards single-threaded evaluation. How to make a parallel multi-threaded version of this mechanism so that it correctly discovers (dynamic) dependency cycles is far from clear, and certainly not something one would want to put into a specification.

Hence in this case, the distance from the specification (axiomatic semantics in Fig. 13) to a program (implementation) is considerable and not easily overcome.

One could nevertheless imagine an specification language with a general fix-point construct that would find a σ satisfying Fig. 13 with reasonable efficiency. When in the next section we extend the semantics to further specify the cost of spreadsheet evaluation, this seems less plausible.

4 Example: Spreadsheet Cost Semantics

In this section we extend the evaluation semantics from Sec. 3.3 to a cost semantics, which in addition to a possible computed value of the expression describes the possible cost of computing it. More precisely, the semantics describes the *work*, that is, uni-processor cost [5], of the computation. In a parallel implementation, some of that work may be performed in parallel.

The cost semantics presented here was developed to enable a (static) cost analysis of spreadsheets, for the purpose of partitioning and scheduling parallel evaluation of spreadsheets; see [6].

The cost of a computation is described by a non-negative integer representing a number of computation steps, for instance the number of evaluation rule applications, plus some measure of the cost of calling a built-in function. This notion of work can reasonably be assumed to be within a constant factor of the actual number of nanoseconds required to evaluate an expression.

4.1 Cost Semantics for Expressions

The evaluation judgment $\sigma \vdash e \Downarrow v$ gets extended to $\sigma \vdash e \Downarrow v, c$, which states that when σ describes the calculated values of all cells, then formula e may

evaluate to value v at computational cost c. As in Section 3.3, the semantics is nondeterministic ("may") in the sense that the evaluation of an expression e could produce many different values v at many different costs c.

The inference rules defining the cost judgment $\sigma \vdash e \Downarrow v, c$ are given in Fig. 14.

$$\frac{}{\sigma \vdash \texttt{n} \Downarrow n, 1}\ (c1)$$

$$\frac{ca \notin dom(\sigma)}{\sigma \vdash \texttt{ca} \Downarrow 0.0, 1}\ (c2b)$$

$$\frac{ca \in dom(\sigma) \qquad \sigma(ca) = v}{\sigma \vdash \texttt{ca} \Downarrow v, 1}\ (c2v)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1, c_1 \qquad v_1 \in Error}{\sigma \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v_1, 1 + c_1}\ (c3e)$$

$$\frac{\sigma \vdash e_1 \Downarrow 0.0, c_1 \qquad \sigma \vdash e_3 \Downarrow v, c_3}{\sigma \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v, 1 + c_1 + c_3}\ (c3f)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1, c_1 \qquad v_1 \neq 0.0 \qquad \sigma \vdash e_2 \Downarrow v, c_2}{\sigma \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v, 1 + c_1 + c_2}\ (c3t)$$

$$\frac{0.0 \leq v < 1.0}{\sigma \vdash \texttt{RAND}() \Downarrow v, 1}\ (c4)$$

$$\frac{\begin{array}{c} J \subseteq \{1, \ldots, n\} \\ \forall j \in J.\ \sigma \vdash e_j \Downarrow v_j, c_j \qquad v_i \in Error \text{ for some } i \in J \end{array}}{\sigma \vdash \texttt{F}(e_1, \ldots, e_n) \Downarrow v_i, 1 + \sum_{j \in J} c_j}\ (c5e)$$

$$\frac{\begin{array}{c} \sigma \vdash e_1 \Downarrow v_1, c_1 \qquad \ldots \qquad \sigma \vdash e_n \Downarrow v_n, c_n \\ \forall i.\ v_i \notin Error \end{array}}{\sigma \vdash \texttt{F}(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n), 1 + \sum_{j=1,n} c_j + work(f, v_1, \ldots, v_n)}\ (c5v)$$

Fig. 14. Cost semantics rules for simplified spreadsheet formulas. From [6].

These rules are mostly straightforward extensions of the formula evaluation rules in Fig. 14:

- Rule (c1) says that evaluating a number constant `n` requires 1 computation step, and similarly for cell references by rules (c2b) and (c2v).
- Rule (c3e) says that if e_1 may evaluate to error v_1 in c_1 computation steps, then `IF`(e_1,e_2,e_3) may evaluate to error v_1 in $1 + c_1$ computation steps.

- Rule (c3f) says that if e_1 may evaluate to zero in c_1 computation steps and the "false branch" e_3 may evaluate to v in c_3 computation steps, then $\texttt{IF}(e_1,e_2,e_3)$ may evaluate to value v in $1 + c_1 + c_3$ computation steps.
- Rule (c3t) is similar, for when e_1 may evaluate to some non-error non-zero number v_1 in c_1 computation steps.
- Rule (c4) says that function call $\texttt{RAND}()$ may evaluate to any (non-error) number v between zero and one, in one computation step.
- Rule (c5e) is quite different from the corresponding evaluation rule (e5e) in Fig. 11. It says that an implementation may choose to evaluate just a subset $\{e_j \mid j \in J\}$ of the arguments when some e_i with $i \in J$ evaluates to an error v_i, and then let v_i be the result of the function call. Also, it says that the total cost of this is the cost $\sum_{j \in J} c_j$ of evaluating that subset of arguments, plus one. The rationale for this is discussed in Section 4.2.
- Rule (c5v) says that if each argument e_i may evaluate to non-error value v_i in c_i computation steps and applying the actual function f to argument values $(v_1, \ldots, v_n)$ produces value v at a cost of $work(f, v_1, \ldots, v_n)$ computation steps, then the call $\texttt{F}(e_1, \ldots,e_n)$ may evaluate to value v using a total of $1 + \sum_{j=1,n} c_j + work(f, v_1, \ldots, v_n)$ computation steps.
  Here $work(f, v_1, \ldots, v_n)$ describes the cost of applying function f to argument values $(v_1, \ldots, v_n)$. For instance, one would expect $work(+, v_1, v_2) = 1$ since the cost of addition is independent of the numbers added.

Since each cost rule adds 1 to the cost incurred by subexpression evaluations, the cost semantics essentially counts the number of rule applications.

4.2 Rationale for Cost of an Error Argument

While most of the cost semantics rules in Fig. 14 are obvious extensions of the evaluation rules in Fig. 11, this is not the case for rule (c5e) which is quite different from rule (e5e). Here we discuss why.

It is possible to imagine a cost rule (c5bad) as a trivial extension of rule (e5e), like this:

$$\frac{\sigma \vdash e_i \Downarrow v_i, c_i \qquad v_i \in Error}{\sigma \vdash \texttt{F}(e_1, \ldots,e_n) \Downarrow v_i, 1 + c_i} \; (c5bad)$$

This rule says that if one of the arguments e_i may evaluate to an error v_i using c_i computation steps, then the call $\texttt{F}(e_1, \ldots,e_n)$ to a function $\texttt{F}$ may evaluate to error v_i in $1 + c_i$ computation steps. However, this cost is unrealistically low: a conforming implementation would have to correctly guess which (if any) argument expression e_i can evaluate to an error, and then evaluate only that expression. Such an implementation would seem implausibly clever.

A more realistic rule might stipulate instead that the cost is the sum of the costs of evaluating all argument expressions. However, this is needlessly pessimistic since an implementation may stop evaluating arguments once one of them evaluates to an error.

Another realistic cost rule might correspond to implementations that evaluate argument expressions $e_1, e_2, \ldots$ from left to right until one of them (if any) evaluates to an error. However, this restricts the possible implementations and would preclude or complicate parallel evaluation of arguments.

Instead we propose rule (c5e) in Fig. 14 which corresponds to implementations that may evaluate the argument expressions in any order (or in parallel) but may avoid evaluating all of them in case one evaluates to an error. This corresponds to choosing a subset $J \subseteq \{1, \ldots, n\}$ of the argument indexes and evaluating only those e_j for which $j \in J$, to values v_j at costs c_j, where one of the v_j is an error, and then stating that the total cost of the call is the sum $\sum_{j \in J} c_j$ of the costs of the arguments actually evaluated, plus one. Through different choices of J, rule (c5e) subsumes all three alternative rules discussed above.

Since the set J may be chosen in many ways, this introduces nondeterminism in the evaluation cost, in addition to nondeterminism in the computed value.

Clearly the choice of the set J is a specification artefact of little interest to a spreadsheet implementer, not to speak of a spreadsheet user. Yet apparently the J set is necessary for the specification to permit realistic implementations without favoring any particular ones. See also the discussion in Sec. 5.1.

4.3 Cost Semantics for Recalculation

Sections 4.1 and 4.2 above gave evaluation-and-cost rules for evaluation of spreadsheet formulas. How do we describe the cost of recalculation in terms of these?

First, we introduce a cost environment $\gamma : Addr \to Nat_0$ such that $\gamma(ca)$ is the cost of evaluating the formula at cell address ca. Then we slightly change the recalculation consistency requirements from Fig. 13 to also record the cost of evaluation for each cell, as shown in Fig. 15.

$$
\begin{array}{l}
(1)\ dom(\sigma) = dom(\gamma) = dom(\phi) \\
(2)\ \forall ca \in dom(\phi).\ \sigma \vdash \phi(ca) \Downarrow \sigma(ca), \gamma(ca)
\end{array}
$$

Fig. 15. Recalculation consistency requirements recording also evaluation cost, for simple formulas. The judgment $\sigma \vdash e \Downarrow v, c$ is defined in Fig. 14. Compared to Fig. 13, requirement (2) has been extended to record the evaluation cost of cell ca in $\gamma(ca)$.

Using the cost environment γ we can now express the cost of a full recalculation of a spreadsheet described by ϕ. This is simply the cost of evaluating the formula of every non-blank cell once:

$$fullcost = \textstyle\sum_{ca \in dom(\phi)} \gamma(ca)$$

In general, it is wasteful to perform full recalculation after only a single cell has been edited by the spreadsheet user. This does not matter here.

5 Specification and Implementation

5.1 Nondeterminism versus Underspecification

It should be clear from the discussion in Sec. 4.2 of the Fig. 14 rule (c5e) that rule nondeterminism in the specification may reflect either run-time nondeterminism in an implementation (rules e4 and c4) or underspecification, that is, implementation-time design choices (rules e5e and c5e), or a mixture of those.

The difference is one of binding-time (as in language implementations and in partial evaluation): when is the nondeterministic choice made, and the chosen value henceforth fixed? Should there be a (formal) way to describe stages (implementation design stage, program linking stage, load-time stage, run-time stage, ...) and to describe when a given choice should be made? For instance, a `RAND()` function whose value gets fixed at 0.500 at the implementation design stage would disappoint many spreadsheet users.

Yet it is not so easy to separate the implementation design stage and the run-time stage even in the simple case of rule (c5e) discussed Sec. 4.2. At what stage does it make sense to choose the set J?

If one has decided on a sequential (singlethreaded, uniprocessor) implementation, one would probably decide at implementation design time on an evaluation order (maybe left to right) for actual arguments. Also, one may or may not stop evaluation once an argument evaluates to an error value. In any case this leaves no nondeterminism at run-time, but corresponds to J always having the form $\{1, 2, \ldots, i\}$ or $\{1, 2, \ldots, n\}$. In this case, the choice of the set J in rule (c5e) represents underspecification, or an implementation-time design choice.

By contrast, if one has decided on a parallel (multithreaded, multiprocessor) implementation, one may evaluate function argument expressions in parallel on multiple threads. Any thread that evaluates an argument to an error value v_i may cancel other argument evaluation threads and make the function call return v_i, discarding the remaining argument evaluations. In this case, the choice of the set J in rule (c5e) represents an implicit nondeterministic run-time choice in the implementation. However, the apparently nondeterministic choice is not made by explicitly choosing a set J, but by the run-time system's scheduler, which appears random due to other loads on the machine, outside interrupts, and truly unpredictable races between multiple cores accessing the same memory.

5.2 Does a Cost Semantics Make Sense?

The cost semantics may seem to be overly specific in prescribing the cost of a computation in addition to its result. Is it sensible for a formal specification to do that?

We believe that this is useful and meaningful for two reasons: First, a cost semantics may be abstracted to a (static) cost analysis, which can then be used in programmer feedback, scheduling decisions and the like [6]. Second, a semantics with additional detail may provide better understanding of the specified language. For instance, a call-by-name semantics for lazy evaluation may correctly

specify the values that an implementation of lazy evaluation must compute, but give the wrong impression of time and memory consumption. A slightly more complicated semantics that properly models arbitrary data graphs in memory will prescribe the same computed values but additionally provide insight into time and memory consumption, and into possible implementations. Indeed, we have previously shown that a proper such semantics for lazy evaluation [15] may be rich enough that an abstract machine and an implementation can be derived from it [21].

In the late 1980es several researchers proposed "instrumented" versions of denotational semantics for programming languages. An abstract interpretation based on the instrumented semantics would then be used to provide static reference count analysis, variable escape analysis, and the like. Thus the instrumentation built some implementation-related properties, or expectations, into the denotational semantics [20, Sec. 2.4]. Of course it is possible for an "instrumented" specification to go overboard and specify something that cannot be implemented, witness the unrealistic cost rule (c5bad) discussed in Sec. 4.2. This problem is not specific to instrumented semantics; an ordinary semantics may well be unimplementable, for instance by specifying a fair nondeterministic choice between infinitely many possibilities.

However, an "instrumented" semantics is likely to contain some internal redundancy, and therefore more likely to be contradictory. For instance, rule (c5v) specifies both that all of a function's arguments must be evaluated once before calling the function *and* that the cost of a function call includes the sum of the costs of the argument evaluations. By mistake or design the rule might leave out one of these aspects and thereby specify a language that has only strange implementations.

5.3 The Distance from Specification to Implementation

The plain spreadsheet evaluation semantics in Sec. 3 remained reasonably close to an implementation. It is imaginable that even the axiomatic specification of recalculation in Fig. 13 could be handled by a specification language with a general fixpoint computation mechanism.

For the spreadsheet cost semantics in Sec. 4, this is much harder to imagine, since the semantics specifies a property of the implementation besides the computed value, namely the time (or number of computation steps) that the implementation consumes to compute the value. Even when the cost semantics is consistent and admits reasonable implementations (as we believe the one in Sec. 4 does), it is hard to imagine a general and usable specification/programming language mechanism that can that guarantee both correct result and correct time consumption.

6 Related Work

The view that programming languages are becoming so expressive that they can conveniently be used as specification languages is certainly not new [7, 10].

Already the liberation from manual memory management in Lisp (1960) and the liberation from explicit evaluation order in Prolog (1972) must have given these languages the flavor of "describing what, not how" relative to other programming languages at the time, making programs in those languages look like specifications. In a 1994 experiment with prototyping, a working Haskell program, solving a programming challenge, was so concise and elegant that it was mistaken for "a mixture of requirements specification and top-level design" by a group of highly experienced software engineers [12, page 14].

The study of the Ada formal description in Sec. 2 is just further evidence that one decade's specification may be a later decade's program.

Also the research on semantics-directed compiler generation [8, 13] in the 1980es and 1990es implicitly conflated specification and implementation. If a programming language implementation can be automatically generated from a sufficiently detailed semantics or specification, then that specification is itself just a program (in a sophisticated language) that can be transformed or compiled into an implementation of the language it specifies. This view is explicit in Tofte's work [24], which also expressed skepticism as to the feasibility and generality of this approach.

Our study in Sec. 2 was loosely inspired by Naur's critique [16, Sections 4 and 5] of Jones and Henhapl's VDM semantics for Algol 60 [11]. But where Naur's paper broadly questions the presumed advantages of formal specifications, taking flaws in Jones and Henhapl's specification as evidence, our view is that even a (formally) flawed formal specification may be valuable because of the insight acquired while developing it.

As mentioned in Sec. 2.3, this view is in fact similar to the 1985 view proposed, in the realm of programming, by Naur himself [17].

7 Conclusion

This paper investigated the relation between specification (or modeling) and programming in the special case where the specification describes a programming language (semantics) and its implementations.

We observed that an Ada formal description written in VDM in 1980 looks very much like a function program today, suggesting that the difference between specification and program may be one of degree. We also argued that while from a formal point of view the specification is incomplete and flawed (and hence its text not usable for purely formal purposes), it was still highly valuable, due to the knowledge acquired by its authors in the process of developing it.

We observed that nondeterminism in a specification may reflect either intended run-time choice in implementations, or permissible choice between possible implementations, but that there is no conventional way to distinguish these roles or binding-times of specification nondeterminism. We also observed that while operational and denotational semantics specification often suggest an implementation, and even axiomatic specifications can sometimes be implemented

(for instance using search or inference), this is much harder when the semantics specifies also the computational cost of the implementation.

Hence there are still simple cases where it is truly hard to regard a semantics (specification) as an implementation (program), or to see how one could mechanically produce an implementation from the semantics.

References

1. Biermann, F., Dou, W., Sestoft, P.: Rewriting high-level spreadsheet structures into higher-order functional programs. In: International Symposium on Practical Aspects of Declarative Languages (2018)
2. Bjørner, D., Jones, C. (eds.): The Vienna Development Method: The Meta-Language, Lecture Notes in Computer Science, vol. 61. Springer-Verlag (1978)
3. Bjørner, D., Oest, O.N. (eds.): Towards a formal description of Ada. Lecture Notes in Computer Science, vol. 98. Springer (1980)
4. Bjørner, D., Gram, C., Oest, O.N., Rystrøm, L.: Dansk datamatik center. In: History of Nordic Computing 3 - Third IFIP WG 9.7 Conference, HiNC 3, Stockholm, Sweden, October 18-20, 2010, pp. 350–359. Springer (2011)
5. Blelloch, G.: Programming parallel algorithms. CACM **39**(3), 85–97 (March 1996)
6. Bock, A., Bøgholm, T., Leth, L., Sestoft, P., Thomsen, B.: Concrete and abstract cost semantics for spreadsheets. Tech. Rep. ITU-TR-2018-203, IT University of Copenhagen (2018), (To appear)
7. Broy, M., Havelund, K., Kumar, R.: Towards a unified view of modeling and programming. In: Margaria, T., Steffen, B. (eds.) 7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. LNCS, vol. 9952, pp. 238–260. Springer (2016)
8. Ganzinger, H., Jones, N. (eds.): Programs as Data Objects, Copenhagen, Denmark, October 1989. Springer-Verlag (1986)
9. Gordon, M., Milner, R., Wadsworth, C.: Edinburgh LCF. A Mechanised Logic of Computation, Lecture Notes in Computer Science, vol. 78. Springer-Verlag (1979)
10. Havelund, K.: Closing the gap between specification and programming: VDM++ and Scala. In: HOWARD-60: A Festschrift on the Occasion of Howard Barringer's 60th Birthday. EPiC Series in Computing, vol. 42, pp. 210–233 (2014)
11. Henhapl, W., Jones, C.B.: A formal definition of Algol 60 as described in the 1975 modified report. In: Bjørner and Jones [2], pp. 305–336
12. Hudak, P., Jones, M.P.: Haskell vs. Ada vs. c++ vs. Awk vs. . . . : An experiment in software prototyping productivity. Tech. Rep. YALEU/DCS/RR-1049, Yale University, Department of Computer Science (October 1994)
13. Jones, N.D. (ed.): Semantics-Directed Compiler Generation, Aarhus, Denmark, January 1980. (Lecture Notes in Computer Science, vol. 94). Springer-Verlag (1980)
14. Kahn, G.: Natural semantics. In: Brandenburg, F., Vidal-Naquet, G., Wirsing, M. (eds.) STACS 87. 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany (Lecture Notes in Computer Science, vol. 247). pp. 22–39. Springer-Verlag (1987)
15. Launchbury, J.: A natural semantics for lazy evaluation. In: Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993. pp. 144–154. ACM (1993)
16. Naur, P.: Formalization in program development. BIT **22**(4), 437–453 (1982)

17. Naur, P.: Programming as theory building. Microprocessing and Microprogramming **15**, 253–261 (1985)
18. Nielson, H., Nielson, F.: Semantics with Applications. An Appetizer. Springer-Verlag (2007)
19. Platek, R.A.: The European formal definition of Ada. A U.S. perspective. IDA Memorandum Report M-389, Institute for Defense Analyses (1988)
20. Sestoft, P.: Analysis and Efficient Implementation of Functional Programs. Ph.D. thesis, DIKU, University of Copenhagen, Denmark (1991), DIKU Research Report 92/6
21. Sestoft, P.: Deriving a lazy abstract machine. Journal of Functional Programming **7**(3), 231–264 (May 1997)
22. Sestoft, P.: Spreadsheet Implementation Technology. Basics and Extensions. MIT Press (2014), ISBN 978-0-262-52664-7. 325 pages
23. Syme, D., Granicz, A., Cisternino, A.: Expert F#. Apress, 4th edn. (2015)
24. Tofte, M.: Compiler Generators. What They Can Do, What They Might Do, and What They Will Probably Never Do. Monographs in Theoretical Computer Science, Springer (1990)
25. Wikipedia: DDC-I. Webpage, at https://en.wikipedia.org/wiki/DDC-I