Eager, Lazy, and Other Executions for Predicative Programming

by

Albert Y. C. Lai

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Eager, Lazy, and Other Executions for Predicative Programming

Albert Y. C. Lai

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2013

Many programs are executed according to the conventional, eager execution order, for which verification of execution costs is well-understood. However, there are other execution orders in use. One such order in common use is lazy execution or lazy evaluation, which is mostly demand-driven. Laziness supports better decompositions of algorithms, e.g., into modular producers and consumers, which enables compositional reasoning of answer correctness, but then timing correctness is more elusive. This thesis gives a formal method for verifying lazy timing, compositional with respect to program structure; it is an extension of a predicative programming theory.

Predicative programming theories are formal methods that unify both specifications and programs as predicates or boolean-typed expressions over memory state and other quantities of interest. Their strengths are mathematical simplicity and support of program development and verification by incremental refinements. Among these theories, Hehner's *a Practical Theory of Programming* has the further strength of leaving termination and timing open rather than a built-in, and therefore is a flexible substrate for various timing schemes corresponding to various execution strategies. We use this substrate for our method for lazy timing.

This thesis also proves soundness of the eager timing scheme in Hehner's work with respect to an eager operational semantics, and our lazy timing scheme with respect to a lazy operational semantics. Thus, if refinements promise an upper time bound, then execution actually stops within that time.

Lastly, this thesis outlines a space of more operational semantics. It is possible ground for more execution strategies.

Acknowledgements

I am most grateful to my supervisor Ric Hehner for guidance and inspiration: bringing me to see the fascinating field of formal programming methodology, teaching how to be formal (symbolic) in arguments for good formatting, economy, and clarity (yes it can be done), exemplifying how to simplify and get to core issues, showing how to seize opportunities to generalize with low cost and big payoff (in particular when I already had an eager execution order and a lazy execution order, the suggestion to cover arbitrary orders), and suggesting how to sanely write a large document such as a thesis—like writing a program by refinement, write down high-level goals, then incrementally elaborate goals and fill in details. I am next grateful to Marsha Chechik for fostering good science and good presentation. I thank the supevisory committee—Ric Hehner, Marsha Chechik, Steve Easterbrook, Azadeh Farzan—and the external examiner Jules Desharnais for suggesting further improvements; for example, Azadeh Farzan alerted me to look for and cite recent work on automatic computing of time bounds. I thank the attendees of IFIP Workgroup 2.3 in summer 2005 for interest and encouragement when I presented early parts of this work. I am grateful that all of the above persons, as well as other faculty and students I met, have been collegial.

This work has been funded by NSERC PGS B.

Contents

Chapter 1

Introduction

Many programming languages stipulate the same order of execution, called *eager* execution, which executes a sequential composition from left to right, among other things. For programs written for eager execution, how to verify execution costs is widely known and well-understood. However, there are other execution orders used by some other programming languages and some systems; these orders are chosen for certain benefits, but their execution costs are often elusive. One such order in common use is *lazy execution*.

1.1 Lazy Execution and Applications

Lazy execution (more often called *lazy evaluation*) roughly means that some parts of a program are not executed until demanded (there is a specified root demand to start the process), and the answers thus computed are shared and reused if aliased; the details are more complicated than this optimistic description, and moreover there are provisions to mark some parts as less lazy. Lazy execution is mostly found in implementations of some functional programming languages such as Gofer, Miranda, and Haskell.

The main application of lazy execution is to support a decomposition of programs into producers and consumers that is hard to carry out in eagerly executed languages. Writing programs in this more well-structured way enables more compositional reasoning of answer correctness. Hughes gives the following motivating example: Square roots can be computed by a function composition (analogous to sequential composition in imperative programming) of a producer of successive approximations (e.g., Newton-Raphson) and a consumer that codifies an accuracy criterion and stops consuming when the criterion is met. Note that the producer does not contain the criterion, yet it is stopped properly by lazy execution; that to change the accuracy criterion, we keep the producer and just swap the consumer; and that to solve a different approximation problem, we keep the consumer and just swap the producer [20].

Nordin and Tolmach have a whole framework of solvers for constraint satisfaction problems based on the decomposition into a producer of candidates, intermediate processors (representing heuristics such as prioritizing and pruning), and a final checker; besides easy experimentation of heuristics by just swapping the intermediate processors, the modularity helps comprehension and proofs [27].

McIlroy has an elegant solution to enumerating the strings of a regular expression using laziness [23]. As a gist of the kind of clean and obvious program structures made possible: If the regular expression is $0^*{+}1^*$, then one program produces the infinite list of ε, 0, 00, 000… another program produces ε, 1, 11, 111… and finally a third program interleaves them.

1.2 The Question of Lazy Execution Time

The question of stating and proving time bounds of lazy executions can be considered harder than that of eager executions. Whereas eager execution runs, for example, an infinite loop either fully or not at all depending on control flow and input data only, lazy execution runs it any number of times depending also on demand. Lazy timing specifications must therefore mention demand information and use more complicated formulas. In addition, since the details of lazy execution are a bit more complicated than an optimistic "only when needed", and since in practice some parts of a program can be marked as less lazy, it could do more work than a human's subjective judgement of "absolutely necessary". This calls for formal methods for verifying lazy execution time bounds, preferrably without mandating a whole-program analysis. This thesis contributes one such method, formalized as an extension of the predicative programming theory named below, although our method is a simplification of previous ones [36, 31]; we also explain why we can afford the simplification.

1.3 Predicative Programming

A *programming theory* formalizes specifications, programs, and satisfactions (criteria for a program to satisfy a specification). *Predicative programming theories* [11, 16] formalize specifications and programs by boolean-typed expressions in which free variables stand for quantities of interest in the problem specification and/or the solution program. (The name *predicative* refers to predicates, an old name for boolean-typed expressions.) With specifications and programs unified, these theories allow hybrid compositions and support stepwise refinement [38] in program development, and therefore satisfaction is also called refinement. Thus, they emphasize deriving programs from specifications, i.e., analysis, verification, and documentation happen in tandem with programming, incrementally and compositionally; most other theories of programming emphasize analysis and verification post-mortem of a monolith, and are mostly silent about documentation.

Different predicative programming theories choose different quantities of interest. For a traditional example, in batch-mode computations, the quantities of interest are initial values and final values of the memory, and so each memory cell is represented by two free variables, say x for its initial value and x' for its final value; incrementing the cell content by one may be specified as $x'{=}x{+}1$. For a more modern example, if a computation takes input and produces output throughout its duration, not just initial values at the beginning and final values at the end, the input history and the output history are also quantities of interest, which are represented respectively by sequence-typed free variables [14]; echoing the input stream to the output stream may be specified as $(\forall i \cdot \mathit{output}\; i{=}\mathit{input}\; i)$ or $\mathit{output}{=}\mathit{input}$.

Among various predicative programming theories, Hehner's *a Practical Theory of Programming* [13, 14, 15] further excludes termination (more precisely, termination under eager execution) as a quantity of interest; it is replaced by initial time and final time (as numeric values and can be infinite), on the ground that termination claims cannot be refuted by observations, while timing claims can be. Accordingly, unlike many predicative programming theories, its constructs (especially sequential composition) and its refinement do not have built-in checks for termination under eager execution. To ensure that a program is productive, one writes eager time bounds in the specification, and proves that they are refined by the program. But the theory makes eager timing optional: one can choose to include it or omit it. In effect, the base theory is neutral on execution orders.

This makes the base theory extensible for different execution orders. In this thesis, we extend it for timing under lazy execution. We also set the stage for more execution orders.

1.4 Structure of This Thesis

This thesis is organized as follows.

- Chapter 2 introduces the predicative programming theory that will be the substrate of this thesis. The theory is a small modification of Hehner's [13, 14, 15]. The chapter poses the question of soundness of refinements, which is then answered in the chapter after. The end of the chapter has bibliographical notes on termination schemes for interested readers.

- Chapter 3 describes a high-level, small-step operational semantics for eager execution, and then uses it to prove soundness of refinements (if the refinements use eager timing).

  Our contribution in this chapter is the proof. (The operational semantics is also a small contribution.)

- Chapter 4 describes our method of lazy timing refinements. Laziness requires expressing how much of the final answer is demanded and how this propagates into the middle of the program; for this, we add usage variables and usage transformation. Equipped with usage variables, we can specify and prove usage-dependent time bounds. Reasoning about these time bounds is compositional with respect to program structure, for example in a sequential composition of a producer and a consumer, the time bound of the producer can be proved independently from the consumer. Our method covers both basic data types (e.g., integers) and lazy algebraic data types (e.g., lazy lists). We provide examples of how to use this method.

  Our contribution in this chapter is the method: usage variables, usage transformation, and usage-dependent time bounds.

- Chapter 5 describes a high-level, small-step operational semantics for lazy execution, shows an example, and then uses it to prove soundness of refinements that use the method in the chapter before.

  Our contribution in this chapter is the operational semantics and the proof.

- Chapter 6 outlines and contributes a space of operational semantics that is more general and with more open ends than both the eager one and the lazy one in the chapters before. It is possible to explore more execution orders and strategies from this space.

- Chapter 7 is the conclusion. It summarizes our contribution and related work, and outlines future work.

Most chapters include discussions of related work.

Chapter 2

A Practical Theory of Programming

A *theory of programming* formalizes specifications, programs, and satisfactions (criteria for a program to satisfy a specification). This chapter outlines the theory of programming used throughout this thesis; it uses part of *A Practical Theory of Programming* [12, 13, 14] and has minor additions.

We use the following symbols, listed in decreasing precedence (also listed in Appendix A):

- $\top$, $\bot$ (boolean values “true” and “false” respectively)
  literals
  parenthesized expressions
- function application written as juxtaposition, e.g., $f\ x$
- $\times$, $/$ (arithmetic)
- infix $+$, $-$ (arithmetic)
- $=$, $\neq$, $<$, $>$, $\leq$, $\geq$ (equality and inequalities; continuing, e.g., $a{=}b{\leq}c$ means $a{=}b$ and $b{\leq}c$)
- $\neg$ (boolean “not”)
- $\wedge$ (boolean “and”)
- $\vee$ (boolean “or”)
- $\Rightarrow$, $\Leftarrow$ (boolean implication, continuing)

- :=, =: (assignment, Section 2.1.2)
- **if then else** (conditional)

  case of (Section 2.1.4)
- . (sequential composition, Section 2.1.2)
- ∀, ∃ (predicate logic quantifiers)

  var, **scope** (Section 2.1.2)
- **=**, **⇒**, **⇐** (same meaning as =, ⇒, and ⇐ respectively, and continuing, but lowest precedence)

A theory of programming defines three things: specification, program, and the satisfaction relation between the two. These are defined in the following sections for the particular theory we use.

2.1 Specifications

The theory of programming we use is a predicative theory [11, 16]. In general, a predicative theory picks some quantities of interest and defines specifications to be boolean expressions (used to be called predicates in the past) having these quantities as free variables. These formalize expectations: a computation is within expectation if and only if the quantities of the computation satisfy the boolean expression.

Below we elaborate on the variables and boolean expressions we use in this thesis.

2.1.1 Quantities of Interest

In our present case, because sequential imperative programming is included, the quantities of interest include the values of the memory variables before and after execution of a given program fragment. Timeliness is also of interest, with a time variable introduced in Section 2.4. There are also usage variables, introduced for lazy execution in Chapter 4. In all three cases—memory, time, usage—we are interested in both before and after, so each state variable becomes a pair of free variables.

Here is the naming convention for the variable values before and after. For a memory variable x, the same name x refers to the value before, also called the pre-value; and the primed name x' refers to the value after, also called the post-value. The same convention applies to the time variable t and later usage variables. Sometimes σ and σ' refer to the aggregates of all pre-values in scope and all post-values in scope, respectively, when stating general statements about specifications in which the actual names do not matter.

Other kinds of quantities are possible in general, though not used in this thesis. If a program communicates with the environment through an unbounded channel, one free variable of a sequence type stands for the complete history of all messages sent and to be sent in chronological order [13]. (An extra pair of memory variables stand for the write cursor and the read cursor.) If a program shares a state variable with another program run concurrently, one free variable of a function type (from time to value) stands for the values of the shared variable at various times [15]. Note that neither the message history nor the shared variable is represented by a pair of pre-value and post-value; each is one single free variable, not two, and already stands for all values of all times.

2.1.2 Boolean Expressions

Any mathematical expression with its result type being boolean and with all free variables being quantities of interest is a specification. It could be formed by relations between quantities and/or composition by logic operators. Here are some examples, with just memory variables x, y, and z for now, and so x, x', y, y', z, z' are the quantities of interest:

$$x' = x+1$$

$$x' = x+1 \land y'=y \land z'=z$$

$$y\neq 0 \land (\exists n\colon int \cdot x=y\times n) \Rightarrow x' = x/y$$

$$\top$$

$$y\neq 0$$

$\perp$

There are some commonly used specifications and operators over specifications:

- $ok \;=\; x'{=}x \wedge y'{=}y \wedge z'{=}z \;=\; \sigma'{=}\sigma$

  corresponds to no-operation, the empty program

- $x{:=}e \;=\; x'{=}e \wedge y'{=}y \wedge z'{=}z$

  corresponds to assignment

- $x{=:}e \;=\; x{=}e \wedge y'{=}y \wedge z'{=}z$

  is backward assignment; in this thesis, we only use it as an accounting device in Chapter 4

- **if** b **then** P **else** $Q \;=\; b{\wedge}P \vee \neg b \wedge Q \;=\; (b{\Rightarrow}P){\wedge}(\neg b{\Rightarrow}Q)$

  corresponds to conditional branching

- $P.Q \;=\; \exists\sigma'' \cdot$ (substitute σ'' for σ' in P)$\wedge$(substitute σ'' for σ in Q)

  corresponds to sequential composition

- **var** $v{:}\,T \cdot P \;=\; \exists v, v'{:}\,T \cdot P$

  corresponds to local memory variable introduction: the fresh pair of pre-value v and post-value v', of domain T, are accessible in P. Often, when the domain is implicit or unimportant, we write **var** $v \cdot P$

- **scope** $v \cdot P$

  corresponds to memory variable hiding: memory variables (pre-values and post-values) other than v (pre-value v and post-value v') are inaccessible in P. We enforce this access restriction syntactically for simplicity. Hiding fewer variables is possible by listing more variables, e.g., **scope** $x, y \cdot P$.

  Note: *A Practical Theory of Programming* [14, 15] has a similar **frame** construct to restrict post-values but still allows access to all pre-values; we deviate from that here for convenience in lazy programs later in this thesis.

We define **scope** by extrapolating from an example. If the memory variables are x, y, and z, we define

$\textbf{scope}\, x, y \cdot P \;=\; P \wedge z'{=}z$

2.1.3 Useful Theorems

There are some useful theorems on the above operators (some theorems are given acronyms for reference):

- $ok.\,P \;=\; P$

  $P.\,ok \;=\; P$

- $P.\,(Q.\,R) \;=\; (P.\,Q).\,R$

  We use this associativity theorem so pervasively that we will seldom cite it.

- **2.1.3-assignment-before**:

  $x{:=}e.\,P \;=\;$ (substitute e for x in P)

  provided e does not mention any post-value.

- **2.1.3-engulf-assignment**:

  $b{\Rightarrow}P.\,x{:=}e \;\Rightarrow\; b{\Rightarrow}(P.\,x{:=}e)$

  provided b does not mention x'.

- **2.1.3-if-distribution**:

  Many operators distribute over if-then-else, in particular we will use:

  $R\wedge\textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ Q \;=\; \textbf{if}\ b\ \textbf{then}\ R\wedge P\ \textbf{else}\ R\wedge Q$

  $(\exists v \cdot \textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ Q) \;=\; \textbf{if}\ b\ \textbf{then}\ (\exists v \cdot P)\ \textbf{else}\ (\exists v \cdot Q)$

  provided b does not mention v.

- $(\textbf{scope}\, v \cdot P.\,Q) \;=\; (\textbf{scope}\, v \cdot P).\,(\textbf{scope}\, v \cdot Q)$

- $(\textbf{scope}\, v \cdot P.\,v{:=}e) \;=\; (\textbf{scope}\, v \cdot P).\,v{:=}e$

  $(\textbf{scope}\, v \cdot v{:=}e.\,P) \;=\; v{:=}e.\,(\textbf{scope}\, v \cdot P)$

  provided e does not mention any variable forbidden by the **scope**.

- $(\textbf{var}\, v \cdot P\,.\, x{:=}e) \;=\; (\textbf{var}\, v \cdot P)\,.\, x{:=}e$

  $(\textbf{var}\, v \cdot x{:=}e\,.\, P) \;=\; x{:=}e\,.\,(\textbf{var}\, v \cdot P)$

  provided x is not v and e does not mention v.

2.1.4 Data Types

While predicative theories of programming are open and flexible about data types, this thesis will make heavy use of the following data types, and so they are worth describing. Before we begin, we must emphasize that data types and their operators are not confined to computers and programming languages; they are also subject matter in specifications and mathematical statements. It is entirely reasonable that some ways of using data types and their operators are rare in programs but common in specifications and proofs. As an example that has already happened, some boolean operators double as common specification operators.

Firstly, as expected, there are the familiar number types (e.g., *nat*, *int*) and the boolean type, with their familiar operators (e.g., + and × for number types, logic operators for the boolean type). Of the boolean type, we say a few more words on one operator, in preparation for algebraic data types covered next.

If b is a boolean operand and $e0$, $e1$ are operands of any type (some programming languages require $e0$ and $e1$ to have the same type), then we have the expression

$\textbf{if}\ b\ \textbf{then}\ e0\ \textbf{else}\ e1$

This if-then-else operator satisfies at least the following laws (some are given names for reference):

- **2.1.4-if-elim**:

  $\textbf{if}\ b\ \textbf{then}\ e\ \textbf{else}\ e \;=\; e$

- **2.1.4-if-resolve**:

  $\textbf{if}\ \top\ \textbf{then}\ e0\ \textbf{else}\ e1 \;=\; e0$

  $\textbf{if}\ \bot\ \textbf{then}\ e0\ \textbf{else}\ e1 \;=\; e1$

- **2.1.4-if-context**:

  within $\textbf{if}\ b\ \textbf{then}\ e0\ \textbf{else}\ e1$:

when rewriting $e0$, b may be assumed

when rewriting $e1$, $\neg b$ may be assumed

- $f\,(\textbf{if}\ b\ \textbf{then}\ e0\ \textbf{else}\ e1) \ \boldsymbol{=}\ \textbf{if}\ b\ \textbf{then}\ f\,e0\ \textbf{else}\ f\,e1$

These laws are consistent with those of the similarly-named specification operator for conditional branching, and so we will use the same name and syntax for both the data-level operator and the specification-level operator.

Next, we cover algebraic data types, which are especially interesting and useful in lazy programs. An algebraic data type is formed by a disjoint union of cartesian products, and recursion (self and mutual) is allowed. The cases of the disjoint union are distinguished by tags. For example, the type of cons-lists of *int*—call it *iclist*—is formed by the disjoint union of

- a singleton (an empty product) for the empty list, and we tag this case *nil*
- the product of *int* and *iclist*, and we tag this case *cons*

As another example, the type of binary trees of *nat*—call it *nbintree*—is formed by the disjoint union of

- a singleton for the empty tree with tag *emp*
- the product of *nat*, *nbintree*, and *nbintree*, with tag *bin*

We write values of algebraic data types as curried function applications of tag names to component values, such as $cons\,3\,(cons\,1\,nil)$ for an *iclist* example and $bin\,4\,emp\,(bin\,0\,emp\,emp)$ for an *nbintree* example. We do not introduce a formal syntax for declaring algebraic data types in this thesis.

Each algebraic data type comes with two kinds of operators: construction and case analysis. Construction operators are simply the tags, e.g., *cons* of arity 2 for *iclist*, *emp* of arity 0 for *nbintree*. They satisfy injectivity laws, e.g., for *iclist*:

- $nil \neq cons\,h\,r$
- $cons\,h\,r = cons\,h1\,r1 \ \boldsymbol{=}\ h{=}h1 \wedge r{=}r1$

In general, for an algebraic data type with tags including *tagj* (of arity *m*) and *tagk* (of arity *n*):

- $tagj\ x_1 \ldots x_m \neq tagk\ y_1 \ldots y_n$
- $tagj\ x_1 \ldots x_m = tagj\ z_1 \ldots z_m \;\boldsymbol{=}\; x_1{=}z_1 \wedge \ldots \wedge x_m{=}z_m$

The case analysis operator, case-of, is conditional branching based on tags, analogous to if-then-else; in addition, for each tag with arity 1 or more, it introduces local names (similar to lambda-bound names) to refer to component values (operands of the tag), so that the branch can use them conveniently. The syntax of case-of is as follows:

For *iclist*:

case c **of** $nil{\rightarrow}e0 \mid cons\ h\ r{\rightarrow}e1$

where h and r are fresh local names, and $e1$ may use them.

For *nbintree*:

case c **of** $emp{\rightarrow}e0 \mid bin\ n\ t0\ t1{\rightarrow}e1$

where n, $t0$, and $t1$ are fresh local names, and $e1$ may use them.

Extrapolating, for an algebraic data type with tags including *tagk* (of arity *n*):

case c **of** $\ldots \mid tagk\ v_1 \ldots v_n{\rightarrow}ek \mid \ldots$

where $v_1, \ldots, v_n$ are fresh local names, and ek may use them.

All tags of the algebraic data type must be covered uniquely.

These case-of operators satisfy at least the following laws (some are given names for reference), analogous to if-then-else:

- **2.1.4-case-elim**:

  case c **of** $tag1 \ldots{\rightarrow}e \mid \ldots \mid tagk \ldots{\rightarrow}e \;\boldsymbol{=}\; e$

  i.e., when every branch is e; provided e does not mention any of the local names

- **2.1.4-case-resolve**:

  case $tagk\, x_1 \ldots x_n$ **of** $tagk\, v_1 \ldots v_n \rightarrow ek \mid \ldots$ $=$ (substitute $x_1, \ldots, x_n$ for $v_1, \ldots, v_n$ in ek)

  if $tagk$ has arity 0, then there is nothing to substitute: **case** $tagk$ **of** $tagk \rightarrow ek \mid \ldots$ $=$ ek

- **2.1.4-case-context**:

  within **case** c **of** $tagk\, v_1 \ldots v_n \rightarrow ek \mid \ldots$:

  when rewriting ek, may assume $c = tagk\, v_1 \ldots v_n$

- f (**case** c **of** $tagk\, v_1 \ldots v_n \rightarrow ek \mid \ldots$) $=$ **case** c **of** $tagk\, v_1 \ldots v_n \rightarrow f\, ek \mid \ldots$

  provided f does not mention any of the local names such as $v_1, \ldots, v_n$

  i.e., f is distributed into or factored out of all branches

Again analogous to if-then-else, we have case-of as both a data operator and a specification operator (the branches are specifications). Of the specification operator, we will use these distribution laws:

- **2.1.4-case-distribution**:

  $R \wedge$**case** c **of** $tagk\, v_1 \ldots v_n \rightarrow P \mid \ldots$ $=$ **case** c **of** $tagk\, v_1 \ldots v_n \rightarrow R \wedge P \mid \ldots$

  provided R does not mention any of the local names

  ($\exists u \cdot$ **case** c **of** $tagk\, v_1 \ldots v_n \rightarrow P \mid \ldots$) $=$ **case** c **of** $tagk\, v_1 \ldots v_n \rightarrow (\exists u \cdot P) \mid \ldots$

  provided c does not mention u

2.2 Syntax of Programs

In predicative theories, programs are defined to be special specifications: They are specifications in that they describe expectations of computer behaviours, and they are special in that they are handed over to computers without further programming, and so they must be limited in expressiveness for computers.

We define programs by syntactic restrictions. We give these restrictions semi-formally: below is a formal grammar with open ends, and following it are further restrictions given informally.

$\langle$program$\rangle$ ::= *ok*

| $\langle$variables$\rangle$:=$\langle$expressions$\rangle$

| **if** $\langle$boolean expression$\rangle$ **then** $\langle$program$\rangle$ **else** $\langle$program$\rangle$

| **case** $\langle$variable$\rangle$ **of** $\langle$case$\rangle$ $(|\langle$case$\rangle)^*$

| $\langle$program$\rangle$. $\langle$program$\rangle$

| **var** $\langle$variables$\rangle \cdot \langle$program$\rangle$

| **scope** $\langle$variables$\rangle \cdot \langle$program$\rangle$

| $\langle$label$\rangle$

$\langle$case$\rangle$ ::= $\langle$tag$\rangle$ $\langle$variable$\rangle^* \rightarrow \langle$program$\rangle$

The non-terminal $\langle$variables$\rangle$ allows a comma-separated list of distinct variables; $\langle$expressions$\rangle$ allows a comma-separated list of expressions of corresponding length. In forward assignment :=, the assigned variables must be memory or time. An "expression" is formed from pre-values of memory and time variables, computer-supported constants, and computer-supported operations. A $\langle$boolean expression$\rangle$ is an "expression" as above, except only memory variables are allowed, plus the requirement that it evaluates to boolean values. In the case-of statement, we restrict the first operand to one variable for simplicity. We leave the type system open.

The non-terminal $\langle$label$\rangle$ allows a specification and treats it as a label, and we impose the following restriction. For each specification S used as a label, a refinement of the form $S \Leftarrow \langle$program$\rangle$ must be given; self- and mutual-references are allowed—the program on the right hand side may use a specification refined by the same or another refinement. The significance of this requirement becomes clear in the next section.

Examples of programs using especially the last requirement are:

- $x' \leq 0$, given $x' \leq 0 \Leftarrow$ **if** $x \leq 0$ **then** *ok* **else** $(x := x - 1 \,.\, x' \leq 0)$
- $x' \leq 0$, given the pair $x' \leq 0 \Leftarrow x' < 0$ and $x' < 0 \Leftarrow x' \leq 0 \,.\, x := x - 1$
- $x' \leq 0$, given $x' \leq 0 \Leftarrow x' \leq 0$

The last two examples look strange, and are further discussed in the next section.

2.3 Satisfaction—Refinement

Satisfaction in the program theory used here is called *refinement*, and is defined as universal implication: specification P (regarded as a problem) is refined by specification S (regarded as a solution) iff

$$\forall \sigma, \sigma' \cdot P \Leftarrow S$$

For convenience and without loss of soundness, often we just state and prove

$$P \Leftarrow S$$

instead. The solution may use the problem as a component, which means pre-fixpoint in denotation and recursion in execution.

2.3.1 Useful Theorems

There are some useful theorems on refinement and operators over specifications.

Operators on specifications are monotonic in the refinement order. In particular, we will use:

- **2.3.1-seq-mono**:
  $(P\,.\,Q) \Leftarrow (P1\,.\,Q1)$ if $P \Leftarrow P1$ and $Q \Leftarrow Q1$
- **2.3.1-var-mono**:
  $(\mathbf{var}\ v \cdot P) \Leftarrow (\mathbf{var}\ v \cdot P1)$ if $P \Leftarrow P1$
- **2.3.1-scope-mono**:
  $(\mathbf{scope}\ v \cdot P) \Leftarrow (\mathbf{scope}\ v \cdot P1)$ if $P \Leftarrow P1$

provided that all the specifications involved comply with the respective scope requirements of the operators.

For refinements involving if-then-else, we will use:

- **2.3.1-refine-by-if**:
  $R \Leftarrow (\mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ Q)$ iff $(b \Rightarrow (R \Leftarrow P)) \wedge (\neg b \Rightarrow (R \Leftarrow Q))$

The major application is dividing the proof of

$$R \Leftarrow \textbf{if } b \textbf{ then } P \textbf{ else } Q$$

into:

- assuming b, proof of $R \Leftarrow P$
- assuming $\neg b$, proof of $R \Leftarrow Q$

Similarly, for refinements involving case-of, we have **2.3.1-refine-by-case**: we will divide the proof of

$$R \Leftarrow \textbf{case } c \textbf{ of } nil \to P \mid cons\ h\ r \to Q$$

into:

- assuming $c = nil$, proof of $R \Leftarrow P$
- assuming $c = cons\ h\ r$, proof of $R \Leftarrow Q$

and likewise for case-of over other algebraic data types.

2.3.2 Example

As a short example, we prove $x' \leq 0 \Leftarrow \textbf{if } x \leq 0 \textbf{ then } ok \textbf{ else } (x := x-1\ .\ x' \leq 0)$ with memory variable x in scope alone. By 2.3.1-refine-by-if:

Assuming $x \leq 0$:

ok

$=$ ⟨definition⟩

$x' = x$

$\Rightarrow$ ⟨assumption: $x \leq 0$⟩

$x' \leq 0$

Assuming $x>0$:

$$x{:=}x-1\,.\,x'\leq 0$$

$=$ $\langle$2.1.3-assignment-before$\rangle$

$$x'\leq 0$$

Although this branch is silly, it will become interesting when we add timing in Section 2.4 below.

2.3.3 On Termination

Some refinements suggest infinite procrastination, e.g.,

- $x'\leq 0 \Leftarrow x'\leq 0$
- the pair $x'\leq 0 \Leftarrow x'<0$ and $x'<0 \Leftarrow x'\leq 0\,.\,x{:=}x-1$

Neither of the refinements, when translated into program code and executed, stops to deliver the promised result; moreover, one of them is an empty loop. They are correct refinements just because they postpone their promise indefinitely.

At first glance, it seems unsatisfactory to allow infinite procrastination as a correct implementation. But whether a program terminates, and more generally when a program terminates, depends on execution order. Already decades ago, two useful execution orders, call-by-name and call-by-value, were identified for both Algol 60 [29] and the lambda calculus [30]. These can be adapted to imperative batch-mode programs as eager execution and lazy execution, respectively, as this thesis will show. What constitutes infinite procrastination in one execution order can become immediate return in another. As an example,

$$x'\leq 0 \;\Leftarrow\; x'\leq 0\,.\,x{:=}0$$

is stuck in the left recursion under eager execution, but is done immediately with the tailing command $x{:=}0$ under lazy execution. With this option open, it is now paramount to keep termination out of correctness of refinements (partial correctness, impartial to termination), isolating out the timing discipline as an add-on to be chosen according to the execution order.

The following section adds the timing discipline for eager execution as in the original theory [12, 13, 14]. In Chapter 4 we show our discipline for lazy execution.

2.4 Termination and Timing

To prove that a refinement represents a timely program, we prove a time bound. The theory supports this by introducing a time variable t, whose pre-value is t and post-value t', for the times before and after a computation, with which it is possible to specify a time bound as an inequality.

There are many ways to interpret and use this time variable: it may be concrete or ghost; it may stand for real time, machine ticks, operation count, or recursive time. Recursive time simply counts the number of recursions (including iterations), which suffices for proving eventuality, and so we will assume it.

When the time variable is present, some common specifications need re-definitions (assume the memory variables are x, y, z):

- σ and σ' include t and t', respectively; so the definition of $P.\,Q$ has one more existentially quantified variable, i.e., σ'' includes t''
- $ok \;=\; x'{=}x \wedge y'{=}y \wedge z'{=}z \wedge t'{=}t \;=\; \sigma'{=}\sigma$
- $x{:=}e \;=\; x'{=}e \wedge y'{=}y \wedge z'{=}z \wedge t'{=}t$
  $t{:=}e \;=\; x'{=}x \wedge y'{=}y \wedge z'{=}z \wedge t'{=}e$
- $x{=:}e \;=\; x{=}e \wedge y'{=}y \wedge z'{=}z \wedge t'{=}t$

Other specifications and specification operators retain their definitions. Useful theorems listed in this chapter still hold.

To prove a time bound for a program, we add time bounds to specifications (including programs), and we prove their refinements. The addition is already done mostly by the re-definitions above, so manual additions are needed only for two kinds of things: specifications, and a time increment $t{:=}t{+}1$ for each recursive call site, since we use recursive time here.

Here is an example. We extend this refinement

with memory variable x alone:

$x'{\le}0 \;\Leftarrow\;$ **if** $x{\le}0$ **then** ok **else** $(x{:=}x{-}1\;.\;x'{\le}0)$

to

with memory variable x and time variable t:

$$S \Leftarrow \textbf{if } x{\le}0 \textbf{ then } ok \textbf{ else } (x{:=}x{-}1\,.\,t{:=}t{+}1\,.\,S)$$

where

$$S = x'{\le}0 \wedge \textbf{if } x{\le}0 \textbf{ then } t'{=}t \textbf{ else } t'{\le}t{+}x$$

The refinement now has x, x', t, and t' in scope; this modifies the meaning of ok and $x{:=}x{-}1$ correctly. The specification S now includes a time bound. The recursive call is sequentially composed with $t{:=}t{+}1$ as required. (In principle it could be composed either before or after; for ease of simplification, under eager timing, we choose before.)

The new refinement can be proved as follows. By 2.3.1-refine-by-if:

Assuming $x{\le}0$:

$S \Leftarrow ok$

$=$ ⟨definitions⟩

$x'{\le}0 \wedge (\textbf{if } x{\le}0 \textbf{ then } t'{=}t \textbf{ else } t'{\le}t{+}x) \Leftarrow x'{=}x \wedge t'{=}t$

$=$ ⟨assumption: $x{\le}0$⟩

$x'{\le}0 \wedge t'{=}t \Leftarrow x'{=}x{\le}0 \wedge t'{=}t$

$=$ ⟨arithmetic⟩

$\top$

Assuming $x{>}0$:

$S \Leftarrow (x{:=}x{-}1\,.\,t{:=}t{+}1\,.\,S)$

$=$ ⟨definition of S⟩

$S \Leftarrow (x{:=}x{-}1\,.\,t{:=}t{+}1\,.\,x'{\le}0 \wedge \textbf{if } x{\le}0 \textbf{ then } t'{=}t \textbf{ else } t'{\le}t{+}x)$

$=$ ⟨2.1.3-assignment-before⟩

$S \Leftarrow x'{\le}0 \wedge \textbf{if } x{-}1{\le}0 \textbf{ then } t'{=}t{+}1 \textbf{ else } t'{\le}t{+}1{+}x{-}1$

$=$ ⟨assumption: $x{>}0$⟩

$S \Leftarrow x'{\le}0 \wedge \textbf{if } x{=}1 \textbf{ then } t'{=}t{+}1 \textbf{ else } t'{\le}t{+}x$

$\Leftarrow$ $\quad\langle$weaken RHS: $t'=t+1$ when $x=1$ is a special case of $t'\le t+x\rangle$

$\quad S \Leftarrow x'\le 0 \wedge t'\le t+x$

$=$ $\quad\langle$definition of $S\rangle$

$\quad x'\le 0 \wedge (\textbf{if } x\le 0 \textbf{ then } t'=t \textbf{ else } t'\le t+x) \Leftarrow x'\le 0 \wedge t'\le t+x$

$=$ $\quad\langle$assumption: $x>0\rangle$

$\quad x'\le 0 \wedge t'\le t+x \Leftarrow x'\le 0 \wedge t'\le t+x$

$=$ $\quad\langle$propositional logic$\rangle$

$\quad \top$

Here is another example: It shows that with this timing scheme we can detect infinite procrastination. Extending this refinement for recursive time

$$x'\le 0 \Leftarrow x'\le 0$$

we can only hope to get and prove (with the extended arithmetic law $1+\infty=\infty$)

$$x'\le 0 \wedge t'=t+\infty \Leftarrow t:=t+1\ .\ x'\le 0 \wedge t'=t+\infty$$

Replacing ∞ by any expression with finite values yields unprovable refinements. Thus the way the theory handles infinite procrastination is not by defining refinement to forbid them, but rather by adding a timing scheme to expose them.

To put it another way, liveness is proved by being turned into safety: introducing an extra variable that increases at critical points (this can be mechanically inserted, such as "before recursion"), and proving a safety bound on the gross increase. Thus terminating refinement can remain as a safety property.

We do not necessarily consider all calls to be recursive calls. For the purpose of proving termination or establishing an asymptotic time bound, we just need one time increment per cycle of calls. In examples in this thesis, we typically have a main program that initializes and then calls a helper, and we do not have a time increment for this call; but the helper calls itself, and we have a time increment for this call.

2.5 The Soundness Question

The theory allows operationally paradoxical refinements in the presence of infinite loops, e.g.,

$$b' \Leftarrow \textbf{if } b \textbf{ then } ok \textbf{ else } (b{:=}\bot \,.\, b')$$

If the program goes into a loop of setting b to $\bot$, how can b ever attain the value of $\top$?

The paradox involves infinite procrastination. In this example, the promise (of setting b to $\top$) is not delivered until time ∞. Anything the program does until then—even operations drifting further and further away from the goal—is therefore fair game. If we allow infinite procrastinations in refinements, we must allow infinite counterproductive procrastinations too.

This paradox is not an inconsistency between refinements and executions: the refinement promises b' at time ∞ (after timing is added), and no operational observation will confirm or refute it. It is unsettling not because of the near-miss refutation, but because of the lack of confirmation. The real question is: what if a refinement promises delivery in finite time, does the execution still confirm it? This will be answered in the affirmative in the next chapters.

2.6 Bibliographical Notes on Termination

For readers interested in termination schemes in programming theories, this section is an overview of how most other predicative and relational programming theories have termination built in. These theories also treat illegal operations (e.g., dividing by zero) like non-termination. They specify (non-)termination and (il)legal operations in some of the following ways:

- Add a termination variable: pre-value says whether this program starts, and post-value says whether this program finishes [18].
- Add a special element to the state space to stand for non-termination, and require every specification to propagate non-termination from pre-state to post-state [7]. (Also Z when proving refinements [39].) This is usually accompanied by the next:
- Use the convention that possible non-termination under a pre-state means that the pre-state leads to all post-states of the state space (both terminating and non-terminating) [11, 7, 18]. (Also Z when proving refinements [39].)

- Use the convention that non-termination under a pre-state means that the pre-state leads to no post-state [24]. (Also Z when writing specifications [39].)

- In every specification, include a set or a condition to stand for a pre-condition for termination [28, 18].

Accordingly, refinement and often sequential composition in these theories contain checks for termination.

Besides predicative and relational theories, most other theories of imperative programming have termination built into the semantics of iteration and recursion [5, 26, 25, 6, 3, 2].

Hehner and Malton have a further discussion of various termination schemes [17].

Chapter 3

Eager Execution

This chapter describes a simple operational semantics for eager execution and proves soundness of eager timing: if a program refines an upper bound on recursive time, execution finishes within that number of recursive calls.

3.1 Eager Operational Semantics

Our operational semantics is a small-step semantics with a high-level execution state, i.e., a collection of rewrite rules over expressions that look like programs.

In more detail, our execution state is a sequential composition of programs, *bindings*, and *right projections* (defined promptly). A *binding* stores memory variables and the time variable as a conjunction of equations, each equation taking the form $\mathit{variable}'\!=\!\mathit{value}$. An example execution state is

$$xs'\!=\!nil \land y'\!=\!0 \land t'\!=\!0\ .\ y\!:=\!y\!+\!1\ .\ xs\!:=\!cons\ y\ xs$$

The binding on the left stores initial or current values. The use of $\mathit{variable}'\!=\!\mathit{value}$ is just right because sequential composition turns it into pre-value for the program that follows; this convention gives our execution states both a predicative reading and an operational reading. The initial value of the time variable is 0 here but can be an arbitrary finite number. The program to be executed is $(y\!:=\!y\!+\!1\ .\ xs\!:=\!cons\ y\ xs)$.

An example of execution illustrates how the execution state evolves:

$$xs'{=}nil \land y'{=}0 \land t'{=}0 \,.\, y{:=}y{+}1 \,.\, xs{:=}cons\ y\ xs$$

$\longrightarrow$ ⟨assignment rule (given below)⟩

$$xs'{=}nil \land y'{=}0{+}1 \land t'{=}0 \,.\, xs{:=}cons\ y\ xs$$

$\longrightarrow$ ⟨evaluation rule (given below)⟩

$$xs'{=}nil \land y'{=}1 \land t'{=}0 \,.\, xs{:=}cons\ y\ xs$$

$\longrightarrow$ ⟨assignment rule⟩

$$xs'{=}cons\ 1\ nil \land y'{=}1 \land t'{=}0$$

and we stop now because the program has been completely “eaten”. The remaining binding has the final answers.

A *right projection* is added by the execution rule for **var** $v \cdot P$ to mark where we can discard local variables. We place this marker to the right of P because eager execution “eats” the program from the left. This marker has also a specification-level meaning: when reading from left to right, it projects the memory state space to omit v. We use the notation *close* v for a right projection for local variable v, and *close* v, w for several local variables at once (here v and w). An example of execution that contains **var** and introduces a right projection:

$$xs'{=}cons\ 1\ nil \land y'{=}1 \land t'{=}0 \,.\, (\mathbf{var}\ v \cdot v{:=}y{+}1 \,.\, y{:=}v)$$

$\longrightarrow$ ⟨local variable introduction rule (given below)⟩

$$v'{=}v \land xs'{=}cons\ 1\ nil \land y'{=}1 \land t'{=}0 \,.\, v{:=}y{+}1 \,.\, y{:=}v \,.\, close\ v$$

$\longrightarrow$ ⟨assignment⟩

$$v'{=}1{+}1 \land xs'{=}cons\ 1\ nil \land y'{=}1 \land t'{=}0 \,.\, y{:=}v \,.\, close\ v$$

$\longrightarrow$ ⟨evaluation⟩

$$v'{=}2 \land xs'{=}cons\ 1\ nil \land y'{=}1 \land t'{=}0 \,.\, y{:=}v \,.\, close\ v$$

$\longrightarrow$ ⟨assignment⟩

$$v'{=}2 \land xs'{=}cons\ 1\ nil \land y'{=}2 \land t'{=}0 \,.\, close\ v$$

$\longrightarrow$ ⟨local variable elimination rule (given below)⟩

$$xs'{=}cons\ 1\ nil \land y'{=}2 \land t'{=}0$$

and we stop now because the program has been completely “eaten”.

Here is the operational semantics. At the beginning, sequentially compose a binding on the left of the program to store initial values, using m to stand for the memory variables:

$$m'{=}initial \land t'{=}0 \,.\, Main$$

To carry out an execution step, find the leftmost subprogram that matches one of the LHS’s of the following rules (they are mutually exclusive), and apply the matching rule to the matching subprogram. (The transition operator $\longrightarrow$ has the same precedence as $\Leftarrow$.)

- Skip:

  $$\begin{array}{ll} & m'{=}a \land t'{=}at \,.\, ok \\ \longrightarrow & m'{=}a \land t'{=}at \end{array}$$

- Assignment: to help state the assignment rule, let us partition the memory variables m into x and y, and assume the assignment statement to be $x{:=}e$.

  $$\begin{array}{ll} & x'{=}ax \land y'{=}ay \land t'{=}at \,.\, x{:=}e \\ \longrightarrow & x'{=}(\text{subst } ax,ay,at \text{ for } x,y,t \text{ in } e) \land y'{=}ay \land t'{=}at \end{array}$$

- Recursive call:

  $$\begin{array}{ll} & m'{=}a \land t'{=}at \,.\, t{:=}t{+}1 \,.\, Label \\ \longrightarrow & m'{=}a \land t'{=}at{+}1 \,.\, Body \end{array}$$

  given the refinement $Label{\Leftarrow}Body$

- Non-recursive call: Some calls are not recursive (e.g., a main program calling a helper just once), and so they do not cost recursive time:

  $$\begin{array}{ll} & m'{=}a \land t'{=}at \,.\, Label \\ \longrightarrow & m'{=}a \land t'{=}at \,.\, Body \end{array}$$

  given the refinement $Label{\Leftarrow}Body$

- Local variable introduction:

$$m'{=}a \wedge t'{=}at \,.\, (\mathbf{var}\; v \cdot P)$$

$$\longrightarrow \quad v'{=}value \wedge m'{=}a \wedge t'{=}at \,.\, P \,.\, close\; v$$

where *value* is an arbitrarily chosen value of the correct data type.

Note: it may be necessary to perform a renaming on **var** $v \cdot P$ before using this rule, so as to avoid name clashes with what's already in *m*. Example:

$$v'{=}0 \wedge x'{=}1 \wedge t'{=}0 \,.\, (\mathbf{var}\; v \cdot v{:=}x \,.\, x{:=}v{+}1)$$

$$= \qquad \langle \text{rename} \rangle$$

$$v'{=}0 \wedge x'{=}1 \wedge t'{=}0 \,.\, (\mathbf{var}\; w \cdot w{:=}x \,.\, x{:=}w{+}1)$$

$$\longrightarrow \qquad \langle \text{local variable introduction; arbitrarily choose 4 initially} \rangle$$

$$w'{=}4 \wedge v'{=}0 \wedge x'{=}1 \wedge t'{=}0 \,.\, w{:=}x \,.\, x{:=}w{+}1$$

- Local variable elimination:

$$v'{=}av \wedge m'{=}a \wedge t'{=}at \,.\, close\; v$$

$$\longrightarrow \quad m'{=}a \wedge t'{=}at$$

- Scope:

$$m'{=}a \wedge t'{=}at \,.\, (\mathbf{scope}\; v \cdot P)$$

$$\longrightarrow \quad m'{=}a \wedge t'{=}at \,.\, P$$

- Conditional branch step 1:

$$m'{=}a \wedge t'{=}at \,.\, \mathbf{if}\; cond\; \mathbf{then}\; P\; \mathbf{else}\; Q$$

$$\longrightarrow \quad \mathbf{if}\; (\text{subst } a{,}at \text{ for } m{,}t \text{ in } cond)\; \mathbf{then}\; (m'{=}a \wedge t'{=}at \,.\, P)\; \mathbf{else}\; (m'{=}a \wedge t'{=}at \,.\, Q)$$

- Conditional branch step 2:

$$\begin{array}{ll} & \textbf{if } \top \textbf{ then } P \textbf{ else } Q \\ \longrightarrow & P \end{array}$$

$$\begin{array}{ll} & \textbf{if } \bot \textbf{ then } P \textbf{ else } Q \\ \longrightarrow & Q \end{array}$$

- Case branch: Suppose the binding $m'{=}a$ contains $x'{=}tag\, e$.

$$\begin{array}{ll} & m'{=}a \wedge t'{=}at\, .\, \textbf{case } x \textbf{ of } tag\, w \rightarrow P \mid \ldots \\ \longrightarrow & w'{=}e \wedge m'{=}a \wedge t'{=}at\, .\, P\, .\, close\, w \end{array}$$

  It may be necessary to rename w to a fresh name, just like local variable introduction.

  This generalizes to tags of other arities in the obvious way.

Expressions bindings are ready for evaluation after assignments; so is a branching condition after conditional branch step 1. We use these simple evaluations:

- Primitive operations: evaluate when all necessary operands are literals, e.g.,

$$\begin{array}{ll} & 1{+}1 \\ \longrightarrow & 2 \end{array}$$

- Conditional expression: evaluate when the condition is a literal:

$$\begin{array}{ll} & \textbf{if } \top \textbf{ then } e0 \textbf{ else } e1 \\ \longrightarrow & e0 \end{array}$$

$$\begin{array}{ll} & \textbf{if } \bot \textbf{ then } e0 \textbf{ else } e1 \\ \longrightarrow & e1 \end{array}$$

- Case analysis expression: evaluate when the argument has its tag exposed, e.g.,

$$\begin{array}{ll} & \textbf{case } \mathit{tag}\, e0 \textbf{ of } \mathit{tag}\, w \to e1 \mid \ldots \\ \longrightarrow & (\text{subst } e0 \text{ for } w \text{ in } e1) \end{array}$$

Execution stops when the binding is the only remaining part of the execution state and all evaluations are finished.

We note a theorem on the relation between execution and refinement.

Theorem 3.1 Using σ to stand for all variables (both memory m and time t), if an execution state $(\sigma'{=}a\,.\,P)$ transits to state $(\sigma'{=}b\,.\,Q)$ after some steps, then $\forall\sigma' \cdot (\sigma'{=}a\,.\,P){\Leftarrow}(\sigma'{=}b\,.\,Q)$.

□

This is obvious for most of the rules, with the exception of some technicality due to **var** and **scope**. Because of **var**, $\sigma'{=}a$ may have more variables than those after the end of P or Q; this can be compensated by limiting $\forall\sigma'$ to the smaller state space at the end. Because the scope rule eliminates the **scope** construct, we may have to put it back when stating $\forall\sigma'\cdot(\sigma'{=}a\,.\,P){\Leftarrow}(\sigma'{=}b\,.\,Q)$.

3.2 Soundness Theorem

This section states and proves the soundness theorem that eager execution results agree with a useful class of refinements, i.e., eager execution delivers the specified memory behaviour and time bound. We first show some examples both covered and not covered by the theorem to introduce the delineating conditions.

Examples not covered by the theorem:

Example 3.1 A specification that simplifies to $\bot$ in some context, and then by logic $P{\Leftarrow}\bot$ vacuously:

$$b \wedge b' \wedge t'{=}t \;\Leftarrow\; b{:=}\bot\,.\, t{:=}t{+}1\,.\, b \wedge b' \wedge t'{=}t$$

Execution will not terminate or deliver $\top$ as the post-value of b. The problem is that under certain pre-values ($b{=}\bot$ in this example), no post-value satisfies the specification $b \wedge b' \wedge t'{=}t$, i.e., under those pre-values, the specification simplifies to $\bot$ regardless of post-values. When this happens, indefinite procrastination is allowed, as the refinement is proved vacuously:

$b{:=}\bot \,.\, t{:=}t{+}1 \,.\, b \wedge b' \wedge t'{=}t$

$=$ ⟨sequential composition, simplify⟩

$\bot \wedge b' \wedge t'{=}t{+}1$

$=$ ⟨propositional logic⟩

$\bot$

$\Rightarrow$ ⟨propositional logic⟩

$b \wedge b' \wedge t'{=}t$

We guard against this by requiring that all programs P involved must, for every pre-value, be satisfiable by some post-value, and so are not vacuous: $(\forall\sigma \cdot \exists\sigma' \cdot P)$. This requirement needs further strengthening; see the next example.

□

Example 3.2 An ill time bound:

(x has type *int*)

$t'{=}t{+}x \;\Leftarrow\; x{:=}x{-}1 \,.\, t{:=}t{+}1 \,.\, t'{=}t{+}x$

Execution will not terminate, let alone meet the time bound (or go back in time!). The problem here is that $t'{=}t{+}x$ can dictate a time decrement (when $x{<}0$ here), which cancels with $t{:=}t{+}1$ for the recursive call, and therefore the refinement can indefinitely procrastinate without being caught. Therefore we require that every program P involved must, for every pre-value, be satisfiable by some post-value that does not decrease time: $(\forall\sigma \cdot \exists\sigma' \cdot P \wedge t'{\geq}t)$. This condition is called *implementable* by Hehner [14, 15]. It suffices to establish this condition for non-compound programs involved (assignment statements, specifications that are refined), since all compositions preserve this condition.

□

Example 3.3 Lacking a time bound:

$$b' \Leftarrow b{:=}\bot \,.\, t{:=}t{+}1 \,.\, b'$$

Execution will not terminate or deliver $\top$ as the post-value of b. This is largely because no time bound is specified, allowing indefinite procrastination. Therefore we require that the starting program P (usually a specification that is refined) to be executed must specify a time bound, i.e., for a suitable function *nat*-valued function f, $(\forall\sigma,\sigma' \cdot P \Rightarrow t' \le t + f\ \sigma)$ should hold. Execution will then take at most $f\ \sigma$ call steps. This condition will be relaxed for a reason explained by Example 3.5 below.

□

Examples covered by the theorem:

Example 3.4 A loop (tail-recursion) with completely specified behaviour:

(x has type *int*)

$P \Leftarrow$ **if** $x{\le}0$ **then** *ok* **else** $(x{:=}x{-}1 \,.\, t{:=}t{+}1 \,.\, P)$

where $P = x' = min\ 0\ x \wedge t' = t + max\ 0\ x$

□

Example 3.5 Behaviour specified with a precondition:

(x has type *int*)

$P \Leftarrow$ **if** $x{=}0$ **then** *ok* **else** $(x{:=}x{-}1 \,.\, t{:=}t{+}1 \,.\, P)$

where $P = x{\ge}0 \Rightarrow x'{=}0 \wedge t'{=}t{+}x$

It only specifies memory behaviour and time bound when the pre-value of x is non-negative. To cover it in the soundness theorem, one condition must be relaxed: in $\forall\sigma,\sigma' \cdot P \Rightarrow t' \le t + f\ \sigma$ the pre-value σ does not have to range over the full state space; it only needs to range over a suitable subspace D, and the soundness theorem considers only executions beginning from pre-values in D, i.e., executing $(\sigma'{=}a \,.\, P)$ where $a{:}D$. The subspace for this example is those states satisfying $x{\ge}0$.

□

Example 3.6 Non-tail recursion and mutual recursion in a system of refinements:

(m and n have type nat)

$P \Leftarrow$ **if** $n{=}0$ **then** ok **else** $(n{:=}n{-}1\,.\,t{:=}t{+}1\,.\,Q\,.\,n{:=}n{+}1)$

$Q \Leftarrow$ **if** $m{=}0$ **then** ok **else** $(m{:=}m{-}1\,.\,t{:=}t{+}1\,.\,P\,.\,m{:=}m{+}1)$

where

$P = m'{=}m \land n'{=}n \land t'{=}t{+}$**if** $n{\le}m$ **then** $2{\times}n$ **else** $2{\times}m{+}1$

$Q = m'{=}m \land n'{=}n \land t'{=}t{+}$**if** $m{\le}n$ **then** $2{\times}m$ **else** $2{\times}n{+}1$

□

We now state the soundness theorem.

Theorem 3.2 Using σ to stand for all variables (memory m and time t), if in a system of refinements,

1. every recursive call is composed with a time increment, i.e., calling S is $(t{:=}t{+}1\,.\,S)$
2. every non-compound program P (assignment statements, labels) used in the refinements satisfies $(\forall\sigma\cdot\exists\sigma'\cdot P\land t'{\ge}t)$, i.e., is implementable
3. the starting program P satisfies $(\forall\sigma{:}\,D\cdot\forall\sigma'\cdot P\Rightarrow t'\le t+f\,\sigma)$, given state subspace D and a nat-valued function f

then

- for each pre-value $a{:}D$ where the time component is finite, $(\sigma'{=}a.\,P)$ executes to some $\sigma'{=}b$ in at most $f\,a$ call steps, and the pair of pre-value a and post-value b satisfies P as a specification.

□

We first justify focusing on the number of recursive calls. We define a program's size by its number of ok's, assignments, **if**'s, **case**'s, **var**'s, **scope**'s, and labels; we define the size of a set of refinements by the sum of program sizes on the right hand side (the bodies). Each size unit except labels takes at most 2 steps to consume (**if** takes two consecutive steps; **var** takes one step and produces *close*, which takes one step later to consume, and so we count two steps and don't count *close* separately). Each label takes 1 call step and expands to more units to be executed, but every cycle of calls contains at least a recursive call step, and so we attribute to n recursive calls at most n+1 expansions, each by at most the size of the set of refinements. So we have

number of all steps

$\le$ 2×((start program size)+(1+number of recursive calls)×(size of the set of refinements))

We write $P \xrightarrow{n} Q$ iff Q is a result of execution starting from P through at most n call steps and unlimited other steps (although we now know their bound). The conclusion of the soundness theorem can be written as:

- $\forall a{:}\, D \cdot \exists b \cdot (\sigma'{=}a.\, P \xrightarrow{f\,a} \sigma'{=}b) \wedge$(substitute a, b for σ, σ' in P)

By Theorem 3.1, transitively we get

$$(\sigma'{=}a.\, P \xrightarrow{n} \sigma'{=}b) \Rightarrow \forall \sigma' \cdot (\sigma'{=}a.\, P) \Leftarrow \sigma'{=}b$$

We next prove an easy part of the conclusion: provided $(\sigma'{=}a.\, P \xrightarrow{n} \sigma'{=}b)$, the part (substitute a, b for σ, σ' in P) holds:

$(\sigma'{=}a.\, P \xrightarrow{n} \sigma'{=}b)$

$\Rightarrow$ ⟨above⟩

$\forall \sigma' \cdot (\sigma'{=}a.\, P) \Leftarrow \sigma'{=}b$

$=$ ⟨sequential composition, simplify⟩

$\forall \sigma' \cdot$ (substitute a for σ in P) $\Leftarrow \sigma'{=}b$

$=$ ⟨predicate calculus⟩

(substitute a, b for σ, σ' in P)

So it remains to prove the $(\sigma'{=}a.\, P \xrightarrow{f\,a} \sigma'{=}b)$ part.

In the proof, we will include premise 3, $(\forall \sigma{:}\, D \cdot \forall \sigma' \cdot P \Rightarrow t' \le t + f\,\sigma)$ for the starting program P, as part of the formal expression to be proved, while leaving the other premises as mostly informal context; so the formal expression is

$$(\forall\sigma{:}\,D\cdot\forall\sigma'\cdot P\Rightarrow t'\le t+f\,\sigma)\Rightarrow\forall a{:}\,D\cdot\exists b\cdot(\sigma'{=}a\,.\,P\xrightarrow{f\,a}\sigma'{=}b)$$

This expression is most suitable for induction on the time bound. Actually, for that induction to work, we will prove a stronger expression (let *at* stand for the time component of initial value *a*):

$$(\forall\sigma{:}\,D\cdot\forall\sigma'\cdot P\Rightarrow t'\le t+f\,\sigma)\Rightarrow\forall a{:}\,D\cdot\exists b\cdot(\sigma'{=}a\,.\,P\xrightarrow{f\,a}\sigma'{=}b)$$

$=$ ⟨move $\forall a$ outer⟩

$$\forall a{:}\,D\cdot(\forall\sigma{:}\,D\cdot\forall\sigma'\cdot P\Rightarrow t'\le t+f\,\sigma)\Rightarrow\exists b\cdot(\sigma'{=}a\,.\,P\xrightarrow{f\,a}\sigma'{=}b)$$

$\Leftarrow$ ⟨specialize σ to a; (subst σ for a in P) $=$ ($\sigma'{=}a\,.\,P$);

let *at* stand for the time component of initial value a⟩

$$\forall a{:}\,D\cdot(\forall\sigma'\cdot(\sigma'{=}a\,.\,P)\Rightarrow t'\le at+f\,a)\Rightarrow\exists b\cdot(\sigma'{=}a\,.\,P\xrightarrow{f\,a}\sigma'{=}b)$$

$\Leftarrow$ ⟨D no longer essential, generalize to the full state space⟩

$$\forall a\cdot(\forall\sigma'\cdot(\sigma'{=}a\,.\,P)\Rightarrow t'\le at+f\,a)\Rightarrow\exists b\cdot(\sigma'{=}a\,.\,P\xrightarrow{f\,a}\sigma'{=}b)$$

$\Leftarrow$ ⟨generalize $f\,a$ to arbitrary $n{:}\,nat$ for induction⟩

$$\forall n{:}\,nat\cdot\forall a\cdot(\forall\sigma'\cdot(\sigma'{=}a\,.\,P)\Rightarrow t'\le at{+}n)\Rightarrow\exists b\cdot(\sigma'{=}a\,.\,P\xrightarrow{n}\sigma'{=}b)$$

$\Leftarrow$ ⟨generalize P to all implementable programs, will help the proof⟩

$$\forall n{:}\,nat\cdot\forall P,a\cdot(\forall\sigma'\cdot(\sigma'{=}a\,.\,P)\Rightarrow t'\le at{+}n)\Rightarrow\exists b\cdot(\sigma'{=}a\,.\,P\xrightarrow{n}\sigma'{=}b)$$

The latter is proved by induction on n:

- Base case. Assume $(\forall\sigma'\cdot(\sigma'{=}a\,.\,P)\Rightarrow t'\le at{+}0)$ (where *at* stands for the time component of a). We just need to show that execution does not hit a recursive call, i.e.,

  $$\sigma'{=}a\,.\,P\xrightarrow{0}\sigma'{=}b\,.\,t{:=}t{+}1\,.\,Label\,.\,More$$

  does not happen. Then $\sigma'{=}a\,.\,P\xrightarrow{0}\sigma'{=}b$ is the only remaining possibility.

  $$(\sigma'{=}a\,.\,P\xrightarrow{0}\sigma'{=}b\,.\,t{:=}t{+}1\,.\,Label\,.\,More)$$

  $\Rightarrow$ ⟨Theorem 3.1; assumption⟩

  $$\forall\sigma'\cdot(\sigma'{=}b\,.\,t{:=}t{+}1\,.\,Label\,.\,More)\Rightarrow t'\le at{+}0$$

$=$ ⟨de Morgan and variations⟩

$\neg(\exists\sigma' \cdot (\sigma'{=}b\ .\ t{:=}t{+}1\ .\ Label\ .\ More) \wedge t'{>}at)$

$=$ ⟨*at* finite⟩

$\neg(\exists\sigma' \cdot (\sigma'{=}b\ .\ t{:=}t{+}1\ .\ Label\ .\ More) \wedge t' \ge at{+}1)$

$=$ ⟨split $\sigma'{=}b$ into $m'{=}bm \wedge t'{=}at$; the time component of b is at too⟩

$\neg(\exists\sigma' \cdot (m'{=}bm \wedge t'{=}at\ .\ t{:=}t{+}1\ .\ Label\ .\ More) \wedge t' \ge at{+}1)$

$=$ ⟨simplify first sequential composition⟩

$\neg(\exists\sigma' \cdot (m'{=}bm \wedge t'{=}at{+}1\ .\ Label\ .\ More) \wedge t' \ge at{+}1)$

$\Rightarrow$ ⟨generalize bm, $at{+}1$ to "all σ"⟩

$\neg(\forall\sigma \cdot \exists\sigma' \cdot (Label\ .\ More) \wedge t' \ge t)$

$=$ ⟨*Label* and *More* are implementable (premise 2)⟩

$\bot$

- Induction step. The induction hypothesis is

$$\forall P, a \cdot (\forall\sigma' \cdot (\sigma'{=}a\ .\ P) \Rightarrow t' \le at{+}n) \Rightarrow \exists b \cdot (\sigma'{=}a\ .\ P \xrightarrow{n} \sigma'{=}b)$$

(where *at* stands for the time component of *a*). Given *P* and *a*, we assume

$$\forall\sigma' \cdot (\sigma'{=}a\ .\ P) \Rightarrow t' \le at{+}n{+}1$$

and prove $\exists b \cdot (\sigma'{=}a\ .\ P \xrightarrow{n+1} \sigma'{=}b)$.

If $\sigma'{=}a\ .\ P \xrightarrow{0} \sigma'{=}b$ without recursive calls, we are done. Otherwise, we hit a state that uses the recursive call rule. The execution up to the recursive call can be summarized below, with $\sigma'{=}a$ split up as $t'{=}at \wedge m'{=}am$:

$t'{=}at \wedge m'{=}am\ .\ P$

$\xrightarrow{0}$ ⟨after some steps except recursive call⟩

$t'{=}at \wedge m'{=}c\ .\ t{:=}t{+}1\ .\ Label\ .\ More$

$\xrightarrow{1}$ ⟨recursive call, with refinement $Label \Leftarrow Body$⟩

$t'=at+1 \wedge m'=c \,.\, Body \,.\, More$

and it remains to prove $\exists b \cdot (t'=at+1 \wedge m'=c \,.\, Body \,.\, More \xrightarrow{n} \sigma'=b)$:

$\exists b \cdot (t'=at+1 \wedge m'=c \,.\, Body \,.\, More \xrightarrow{n} \sigma'=b)$

$\Leftarrow$ ⟨instantiate induction hypothesis: P to $Body.More$,

a to time part $at+1$, memory part c⟩

$\forall \sigma' \cdot (t'=at+1 \wedge m'=c \,.\, Body \,.\, More) \Rightarrow t' \le at+1+n$

$\Leftarrow$ ⟨the execution $t'=at \wedge m'=am \,.\, P \xrightarrow{1} t'=at+1 \wedge m'=c \,.\, Body \,.\, More$ above;

Theorem 3.1; $\Leftarrow$ transitive⟩

$\forall \sigma' \cdot (t'=at \wedge m'=am \,.\, P) \Rightarrow t' \le at+n+1$

$=$ ⟨$t'=at \wedge m'=am = \sigma'=a$; assumption⟩

$\top$

This concludes the proof.

In the proof of Theorem 3.2, the implication

$$(\forall \sigma' \cdot (\sigma'=a.\,P) \Rightarrow t' \le at+n) \Rightarrow \exists b \cdot (\sigma'=a.\,P \xrightarrow{n} \sigma'=b)$$

cannot be strengthened further to an equivalence because P may underestimate the actual cost of the refinement, e.g.,

$$t' \le t+10 \Leftarrow ok$$

In the logical sense, while the antecedent is not a necessary condition, it is sharp. In a broader sense, the condition is necessary: if it is false, there is a refinement of P (not necessarily the given one) that takes more than n recursive calls to execute.

3.3 Bootstrapping of Implementability

To obtain the blessing of the soundness theorem, one must first prove the implementability of the specifications involved, i.e., prove a theorem of the form

$$\forall\sigma \cdot \exists\sigma' \cdot S \wedge t' \geq t$$

Often, it calls for a constructive proof, which is a program for S, which is a set of refinements, which begs the soundness question, which requires one to prove implementability

$$\forall\sigma \cdot \exists\sigma' \cdot S \wedge t' \geq t$$

which calls for a constructive proof...

We now describe how to break this cycle in practical cases. For ease of discussion and without loss of generality, assume the refinement in question is

$$S \Leftarrow \ldots t{:=}t{+}1 \,.\, S \ldots$$

Usually S can be factored as $M \wedge C$, where M focuses on the final answer in the memory variables and C focuses on the time bound (may also contain a helper invariant on the memory variables, weaker than what M specifies), and the following refinements can be proved (replacing S by M and by C):

$$M \Leftarrow \ldots t{:=}t{+}1 \,.\, M \ldots$$
$$C \Leftarrow \ldots t{:=}t{+}1 \,.\, C \ldots$$

The implementability of C is easy to prove by design because this holds for most time bounds and helper invariants in practice, and we use C for bootstrapping. By the soundness theorem, execution using C as the label terminates in the promised time and has a final answer. But it is the same execution and the same refinement scheme using M or even $M \wedge C$ as the label, and so we know $M \wedge C$ has a final answer too:

$$\forall\sigma \cdot \exists\sigma' \cdot M \wedge C$$

Lastly, because M is supposed to leave final time open (only C specifies final time), C is implementable, and the helper invariant in C is implied by M, we conclude

$$\forall\sigma \cdot \exists\sigma' \cdot M \wedge C \wedge t' \geq t$$

So S is implementable, and the refinement for S is sound.

3.4 Related Work

Here are some theories of predicative programming and vicinity that come with operational semantics. A Practical Theory of Programming shows an example of what a compiler may do, and later states soundness without proof [14]. Unifying Theories of Programming contains a predicative theory and shows its correspondence to a rewriting operational semantics [18], which maintains the memory store and the program in a tuple. Call-by-value functional programming, which is deterministic predicative programming with just initial and final values and a restricted specification form (final value = function of initial value), has a time calculus in the same spirit as adding $t{:=}t{+}1$ for recursive calls, with a soundess proof using the call-by-value lambda calculus, with a small technical gap concerning errors such as asking for the head of an empty list [31]. (Our use of unrestricted specifications liberates our scheme from the trap of such error states—just add a precondition!) The Refinement Calculus of Back and von Wright uses two-person games [3]. The theory of the guarded command language and weakest preconditions uses an informal operational semantics [6].

Our operational semantics is closest to that in Unifying Theories of Programming of Hoare and He above, but we go one step further: we keep the memory store in specification form and fuse it with the program. This style makes the relation between execution and refinement more seamless, and it also makes our lazy operational semantics more seamless in Chapter 5, which will need more memory stores at more locations.

Chapter 4

Lazy Timing

As noted in Chapter 2, a theory of programming can be given a timing discipline corresponding to a supposed execution strategy, and Section 2.4 gives the timing discipline corresponding to eager execution. This chapter gives a different one: that to lazy execution. It can be used to prove time bounds of program executions without referring to the operational semantics and without a whole-program analysis.

4.1 Representing Demand: Usage Variables

To account for running time (lazy evaluation or not), we introduce the time variable t as in Section 2.4 for recursive time. And under lazy execution, program execution time depends on which post-values are demanded (among other things), and so we need extra variables to stand for that information (to be mentioned in time bound formulas). The program itself represents how demands on post-values lead to demands on pre-values, and so we also need extra variables to stand for that. For example, let the memory variable have pre-value m and post-value m'. Let the demand on m' be um', which tells whether the post-value m' is needed either as final output or as input to a subsequent computation; let the demand on m be um, which tells whether the pre-value m is needed for this computation. Then a typical specification for some computation on m, with demand and time, goes like

$$m' = f\ m \land um = F\ um' \land t' = t + ft\ m\ um'$$

($t' = t + ft\ m\ um'$ is elaborated at the end of this section after explaining the type and values of um'. $um = F\ um'$ is elaborated in the next section.)

We need to mention demands on pre-values because, through sequential composition, they become demands on post-values of a preceding program, which are obviously needed for the time bound of the preceding program, and ultimately of the whole program. For example,

$$m' = g\ m \wedge um = G\ um' \wedge t' = t + gt\ m\ um'\ .\ m' = f\ m \wedge um = F\ um' \wedge t' = t + ft\ m\ um'$$

$$= \qquad \langle \text{sequential composition} \rangle$$

$$\exists m'', t'', um'' \cdot \quad m'' = g\ m \wedge um = G\ um'' \wedge t'' = t + gt\ m\ um''$$
$$\wedge\ m' = f\ m'' \wedge um'' = F\ um' \wedge t' = t'' + ft\ m''\ um'$$

$$= \qquad \langle \text{predicate calculus} \rangle$$

$$m' = f\ (g\ m) \wedge um = G\ (F\ um') \wedge t' = t + gt\ m\ (F\ um') + ft\ (g\ m)\ um'$$

Our use of sequential composition to complete the demand pathway justifies the designation um' for the demand on m' and similarly um for m. Henceforth we refer to the pair as a *usage variable*, and consider it to be analogous to memory variables and the time variable in the aspect that it comes with a "pre"-value um and a "post"-value um'. The only peculiarity is that demand information flows from um' to um for usage variables, whereas data flows from m to m' for memory variables; however, information flow direction is absent at the predicative level (it belongs to the operational level).

Next, we designate the data types of usage variables and the representation of demand information. In simple cases, a memory variable holds data of a primitive type such as a boolean, an integer, or a character; accordingly, its value is either used or unused, and so the corresponding usage variable can be a boolean. Similarly, a memory variable of an array type can have a usage variable of the boolean array type. However, this scheme does not generalize, and we will not use it.

Richer usage representations than the booleans are needed for lazy algebraic data types. Algebraic data types are described in Section 2.1.4; to recapitulate, they are disjoint unions of cartesian products, and recursion is allowed. For example, the type *iclist* of cons-lists of *int* is formed by the disjoint union of

- a singleton (an empty product) for the empty list, with tag *nil*
- the product of *int* and *iclist*, with tag *cons*

We write values of algebraic data types as curried function applications of tag names to component values, such as *cons* 3 (*cons* 1 *nil*).

A lazy algebraic data type further stipulates that the disjoint union and the components can be used or unused to various degrees. To elaborate, for *iclist* described above, a list could be unused altogether, or used just to the point of resolving *nil* vs *cons*; in the *cons* case, the *int* component could be used or unused, and the list component is the same story all over again. As a concrete example, the list *cons* 1 *nil* admits the following degrees of usage:

- unused
- resolved to *cons*, both components unused
- resolved to *cons*, the 1 is used, the list component is unused
- resolved to *cons*, the 1 is unused, the list component is resolved to *nil*
- resolved to *cons*, the 1 is used, the list component is resolved to *nil*

The list *cons* 3 (*cons* 1 *nil*) admits the following degrees of usage, altogether 11 possibilities, coarsely enumerated here for brevity:

- unused
- resolved to *cons*, the 3 is unused, the list component is any of the above for *cons* 1 *nil*
- resolved to *cons*, the 3 is used, the list component is any of the above for *cons* 1 *nil*

To represent all possibilities of partial usage of lazy algebraic data values, the usage type needs to mimick the type concerned. We adopt the following slightly redundant scheme for ease of statement based on reuse of well-known mathematics. Borrowing from denotational semantics, domain theory, and context analysis [37], we correspond each data type to an extended type: add one special value (traditionally thought of as "no information"), and replace component types by corresponding extended types. We use the symbol “$\boxdot$” for the special value. (The “$\perp$” symbol has already been taken for the boolean “false”.) To illustrate, extended *int* is *int* together with “$\boxdot$”, and extended *iclist* is the disjoint union of:

- $\boxdot$
- a singleton (an empty product) for the empty list, with tag as *nil*
- the product of extended *int* and extended *iclist*, with tag *cons*

In these extended types, partial values are possible, such as:

- $\boxdot$
- $\mathit{cons}\,\boxdot\,\boxdot$
- $\mathit{cons}\;1\,\boxdot$
- $\mathit{cons}\,\boxdot\,\mathit{nil}$
- $\mathit{cons}\;1\;\mathit{nil}$

Note how these incomplete values nicely express all of the different degrees of usage of $\mathit{cons}\;1\;\mathit{nil}$. Therefore, we adopt extended types for usage variables: A component value unused is represented by $\boxdot$, while a component value used is represented by its actual data value. We emphasize that we do not adopt extended types for memory data variables. We do exactly this: given the data type (unextended) of a memory variable, the corresponding usage variable has the corresponding extended type.

Up to this point, a usage variable may take on invalid values, and what constitutes invalid usage values depends on the data value of the corresponding memory variable. If the data value is $\mathit{cons}\;1\;\mathit{nil}$, then invalid usage values are *nil*, $\mathit{cons}\;0\,\boxdot$, $\mathit{cons}\,\boxdot\,(\mathit{cons}\,\boxdot\,\boxdot)$, and many more. We need to constrain valid usage values by data values at every point in the program. Still borrowing from denotational semantics, domain theory, and context analysis [37], we define a partial order $\sqsubseteq$ (same precedence as $=$ and traditionally thought of as “information order”):

- for extended value e, $\boxdot \sqsubseteq e$

- for unextended primitive value x (e.g., a number, a boolean), $x \sqsubseteq x$

- for extended algebraic values $tag\ x_1 \ldots x_n$ and $tag\ y_1 \ldots y_n$ of the same tag, compare componentwise, i.e.,

  $$tag\ x_1 \ldots x_n \sqsubseteq tag\ y_1 \ldots y_n \Leftarrow x_1 \sqsubseteq y_1 \wedge \ldots \wedge x_n \sqsubseteq y_n$$

  this also works for 0-ary tags, for example $nil \sqsubseteq nil$

- co-induction over the above (induction does not suffice: under induction, the infinite cons-list of $\boxdot$'s is not $\sqsubseteq$ the infinite cons-list of 1's)

The constraint on a usage variable *um*, given its memory variable *m*, is then $um \sqsubseteq m$. We stipulate this as an invariant on all programs. More precisely, we stipulate the healthiness condition $um' \sqsubseteq m' \Rightarrow um \sqsubseteq m$, so that a program represents how a valid demand on post-values leads to a valid demand on pre-values. In practice, it is already satisfied by careful definition of programming constructs, and so programmers seldom need to express or verify it explicitly.

The existence of domains for these extended values (technically non-trivial for recursively defined algebraic data types) and their order-theoretic properties are established by Smyth and Plotkin [35]. (Existence alone is established earlier by Scott [33], but we need much stronger properties here.) Below are the properties we will use:

- binary least upper bound $x \sqcup y$ exists if there is an upper bound, i.e., $\exists z \cdot x \sqsubseteq z \wedge y \sqsubseteq z$, e.g., we usually have complete values as upper bounds

- greatest lower bound over a non-empty set exists

We can now express time bounds with usage variables. Starting with primitive types again, if x is a primitive type memory variable, and we want to say that the time cost is 1 if x' is used, and 0 otherwise, then we can write one of the following:

$$t' = t + \textbf{if } ux' = \boxdot \textbf{ then } 0 \textbf{ else } 1$$

$$t' = t + \textbf{if } ux' = x' \textbf{ then } 1 \textbf{ else } 0$$

For lazy algebraic data types, again taking cons-lists for example, if *xs* is a cons-list memory variable, and we want to say that the time cost is the number of cons cells used (count 0 for both *nil* and $\boxdot$), then we define mathematically *ulen* for this count:

$$ulen\,\boxdot = 0$$

$$ulen\,nil = 0$$

$$ulen\,(cons\,h\,r) = 1 + ulen\,r$$

and then we can specify

$$t' = t + ulen\,uxs'$$

4.2 Propagating Demand: Usage Transformation

For each program, we need to derive and augment how it transforms post-usage to pre-usage, since this is required for calculating the result of sequential compositions and composed times. We show how to transform usage for basic programming constructs, which is mechanical. There is no mechanical method for deriving usage transformation of arbitrary specifications, just as there is no mechanical method for writing specifications, being products of negotiation between user wish and programmer wish (pre-usage can be wished upon rather than derived, too); however, we do suggest a guiding principle for writing arbitrary usage transformations, generalizing from those for basic programming constructs.

4.2.1 Assignments without Operations

We begin with *ok*. Suppose the memory variables are x and y; then *ok* was $x'{=}x \wedge y'{=}y \wedge t'{=}t$ before we had usage variables. So how x is used is exactly how x' is used, and similarly for y and y'. We formalize this usage transformation as:

$$x'{=}x \wedge y'{=}y \wedge t'{=}t \wedge ux{=}ux' \wedge uy{=}uy'$$

It fits the spirit of *ok*, which is “post-values equal pre-values”. We now re-define *ok* to add usage as

$$ok \;=\; x'{=}x \land y'{=}y \land t'{=}t \land ux{=}ux' \land uy{=}uy'$$

Henceforth, *ok* does not need explicit usage transformation because it already contains the necessary $ux{=}ux' \land uy{=}uy'$.

Assignment statements without operations come in two flavours: from constant literal c in $x{:=}c$ and from variable y in $x{:=}y$.

In the constant case, $x{:=}c$ expanded to $x'{=}c \land y'{=}y \land t'{=}t$ before we had usage variables. So x is unused, while how y is used is exactly how y' is used. We add this usage transformation as:

$$x'{=}c \land y'{=}y \land t'{=}t \land ux{=}\boxdot \land uy{=}uy'$$

To express this in program notation, we re-define assignment statements to include usage variables:

$$x{:=}e \;=\; x'{=}e \land y'{=}y \land t'{=}t \land ux{=}ux' \land uy{=}uy'$$

where e is an expression

We don't put or hide usage transformation in the definition of assignment statements; this is in line with the spirit of "changing" just one variable and preserving the rest. Setting *ux* and *uy* is separated into the next definition, *backward assignment* for usage variables:

$$ux{=:}e \;=\; x'{=}x \land y'{=}y \land t'{=}t \land ux{=}e \land uy{=}uy'$$

where e is an expression

Sequentially composing assignments and backward assignments gives us a program notation to express both memory changes and usage changes:

$$x'{=}c \land y'{=}y \land t'{=}t \land ux{=}\boxdot \land uy{=}uy'$$

$$=\quad ux{=:}\boxdot.\, x{:=}c$$

The order of composition is not important in this case. (In other cases, putting usage assignments first is advantageous.)

In the case of $x{:=}y$, this expanded to $x'{=}y \land y'{=}y \land t'{=}t$ before we had usage variables. This time, x is unused, but the use of y is a combination of the use of x' and the use of y', in the sense that a part of y is used iff that part is required by ux' or by uy', which is formalized by the least upper bound operator $\sqcup$ (precedence below $+$ and above $=$, $\sqsubseteq$), e.g., $\boxdot\sqcup 2=2$, $cons\,\boxdot\,nil\sqcup cons\,3\,\boxdot = cons\,3\,nil$. We add this usage transformation as:

$$x'=y \wedge y'=y \wedge t'=t \wedge ux=\boxdot \wedge uy=uy'\sqcup ux'$$

$$= \quad ux=:\boxdot\,.\,uy=:uy'\sqcup ux'\,.\,x:=y$$

For backward assignment statements, we expect e to not use usage pre-values; it can use usage post-values and memory values (pre or post), for example $uy'\sqcup ux'$. There are some useful theorems on backward assignment, analogous to those for forward assignment:

- **4.2.1-assignment-after**:

  $P\,.\,u=:e \;\;=\;\;$ (substitute e' for u' in P)

  provided e does not mention u

  where e' means substituting all pre-values by post-values (memory, time, usage) in e

- **4.2.1-engulf-assignment**:

  $u=:e\,.\,b\Rightarrow P \;\;\Rightarrow\;\; b\Rightarrow(u=:e\,.\,P)$

  provided b does not mention u

4.2.2 Operations on Primitive Data

Next, we discuss operations on primitive types, beginning with *strict* operations, i.e., those that unconditionally use all operands. These are one coherent class to treat because usage for primitive types is completely specified by the dichotomy "used" and "unused". To be concrete, suppose the memory variables are $x{:}\,int$ and $y{:}\,int$, and we consider $x:=x+y$. We take the + operation to behave such that if its result is used, both operands are used (we say that + is *strict* in both operands). If x' is used, x and y are used; if y' is used, y is used; otherwise, x and y are unused. Turning it around, x is used iff x' is used, and y is used iff x' or y' is used. This can be expressed as:

$$ux=:(\textbf{if } ux'=\boxdot \textbf{ then } \boxdot \textbf{ else } x)\,.\,uy=:uy'\sqcup(\textbf{if } ux'=\boxdot \textbf{ then } \boxdot \textbf{ else } y)\,.\,x:=x+y$$

The order is only important insofar as feeding the correct version of x and y into the two backward assignments.

The clumsy expression is due to using incomplete values rather than booleans for usage (e.g., with booleans, we could have written $uy{=}{:}uy' \vee ux'$), but this will buy us simpler expressions when it comes to operations on lazy algebraic types. The clumsiness is also mitigated by defining the commonly used:

$$u \triangleright e \;\; \mathbf{=} \;\; \textbf{if } u{=}\boxdot \textbf{ then } \boxdot \textbf{ else } e$$

$$(u_1, \ldots, u_n) \triangleright e \;\; \mathbf{=} \;\; \textbf{if } u_1{=}\boxdot \wedge \ldots \wedge u_n{=}\boxdot \textbf{ then } \boxdot \textbf{ else } e$$

(Precedence of $\triangleright$ is below $\sqcup$ and above $=$.) Then we can rewrite the program with usage as:

$$ux{=}{:}ux' \triangleright x \,.\, uy{=}{:}uy' \sqcup (ux' \triangleright y) \,.\, x{:=}x{+}y$$

If the assignment assigns to x but does not use x as an operand, for example $x{:=}y{+}y$, then the usage of y is as in the above, and the usage of x is just $ux{=}{:}\boxdot$. In full,

$$ux{=}{:}\boxdot \,.\, uy{=}{:}uy' \sqcup (ux' \triangleright y) \,.\, x{:=}y{+}y$$

In general, to add usage transformation to $x{:=}e$, where e is strict in all operands, and all operands and the result are of primitive types:

- if x is an operand within e, add $ux{=}{:}ux' \triangleright x$; if not, add $ux{=}{:}\boxdot$
- if memory variable y other than x is an operand within e, add $uy{=}{:}uy' \sqcup (ux' \triangleright y)$
- other memory variables and their usage variables do not need explicit treatment—all they need is a form of $u{=}{:}u'$, which is already implicit in both forward assignments and the above backward assignments

("add" means sequentially compose before $x{:=}e$; this order is chosen so that the backward assignment statements can depend on memory pre-values).

Some basic operations on some primitive types are conditionally strict, in particular short-circuit boolean operators. For example, with short-circuit $\wedge$, $b \wedge c$ uses b, and it uses c iff b evaluates to $\top$. Usage transformation adds conditionals to the above formulas, e.g., whereas previously we choose $ux{=}{:}ux' \triangleright x$ or $ux{=}{:}\boxdot$ statically, now we let a "run-time" test choose:

- if x is an operand and cx is the condition for strictness in x, add

  $ux{=}{:}\,\mathbf{if}\ cx\ \mathbf{then}\ ux'{\rhd}x\ \mathbf{else}\ \boxdot$

- if memory variable y other than x is an operand and cy is the condition for strictness in y, add

  $uy{=}{:}\,uy'{\sqcup}\mathbf{if}\ cy\ \mathbf{then}\ ux'{\rhd}y\ \mathbf{else}\ \boxdot$

Here are some typical examples with short-circuit $\wedge$:

- $ub{=}{:}\,ub'{\rhd}b\ .\ uc{=}{:}\,uc'{\sqcup}(\mathbf{if}\ b\ \mathbf{then}\ ub'{\rhd}c\ \mathbf{else}\ \boxdot)\ .\ b{:=}b{\wedge}c$

- $ub{=}{:}\,ub'{\sqcup}(uc'{\rhd}b)\ .\ uc{=}{:}(\mathbf{if}\ b\ \mathbf{then}\ uc'{\rhd}c\ \mathbf{else}\ \boxdot)\ .\ c{:=}b{\wedge}c$

- $ua{=}{:}\boxdot\ .\ ub{=}{:}\,ub'{\sqcup}(ua'{\rhd}b)\ .\ uc{=}{:}\,uc'{\sqcup}(\mathbf{if}\ b\ \mathbf{then}\ ua'{\rhd}c\ \mathbf{else}\ \boxdot)\ .\ a{:=}b{\wedge}c$

4.2.3 Constructions of Algebraic Data

For lazy algebraic data types, there are two basic operations: construction and case analysis. This subsection treats construction, and the next subsection treats case analysis. We use cons-lists for example again before generalizing. Construction means building an algebraic data value using tags and storing it in a memory variable, such as $xs{:=}nil$ and $xs{:=}cons\ x\ xs$.

Suppose the memory variables are y and xs, with xs of the cons-list type (y may be of a primitive type or a lazy algebraic type). Usage transformation for $xs{:=}nil$ is easy:

$uxs{=}{:}\boxdot\,.\,xs{:=}nil$

For $xs{:=}cons\ y\ xs$, the pre-value of xs is the tail of the post-value xs', and so is their usage: the extent of using the tail of xs' becomes the extent of using xs, or else both are unused; the use of pre-value y combines the use of post-value y' and the use of the head of xs'. For convenience, we first define *head* and *tail* mathematically:

$head\ \boxdot = \boxdot$

$head\ nil = \boxdot$ (not used here but useful elsewhere)

$head\ (cons\ h\ r){=}h$

$tail\,\boxdot = \boxdot$

$tail\,nil = \boxdot$ (not used here but useful elsewhere)

$tail\,(cons\,h\,r) = r$

Then usage transformation added to $xs := cons\,x\,xs$ can be expressed as:

$uxs =: tail\,uxs'\ .\ ux =: ux' \sqcup head\,uxs'\ .\ xs := cons\,x\,xs$

($head\,nil$ and $tail\,nil$ do not happen here under the assumption $uxs' \sqsubseteq xs'$.)

Generally for a construction assignment statement $xs := tag\,x_1 \ldots x_n$, first define $untag_i$ to select the ith component of tag, or to return $\boxdot$ if inapplicable:

$untag_i\,(tag \ldots u_i \ldots) = u_i$

$untag_i\,u = \boxdot$ otherwise (including when $u{=}\boxdot$)

then we add usage transformations as:

- if xs is the ith operand, add $uxs =: untag_i\,uxs'$; if not, add $uxs{=:}\boxdot$
- if memory variable y other than xs is the ith operand, add $uy =: uy' \sqcup untag_i\,uxs'$

Although $untag_i$ would make no sense for a 0-ary tag such as nil, the above can still be adopted for $xs{:=}nil$ by noting that no variable is an operand, and so the above just adds $uxs{=:}\boxdot$ and spares us the question of $unnil$.

4.2.4 Case Analyses of Algebraic Data

For algebraic data types, case analysis means conditional branching based on tags, and in addition introducing local names (similar to lambda-bound names) to refer to component values, for example:

$x{:=}\textbf{case}\ xs\ \textbf{of}\ nil{\rightarrow}0 \mid cons\,h\,r \rightarrow h{+}1$

$xs{:=}\textbf{case}\ xs\ \textbf{of}\ nil{\rightarrow}nil \mid cons\,h\,r \rightarrow r$

(This subsection covers case analysis at the expression level. Case analysis can also be at the program statement level, e.g., **case** *xs* **of** $nil{\to}(x{:=}0) \mid cons\,h\,r{\to}(y{:=}0)$. The program statement level is covered in Section 4.2.7.)

For case analysis such as $x{:=}$**case** *xs* **of** $nil{\to}e0 \mid cons\,h\,r{\to}e1$, if x' is used to some degree ($ux'{\neq}\boxdot$), the usage of *xs* must be enough to discern its tag (*nil* vs *cons*), and possibly more according to $e0$, $e1$, and uxs'. Other memory variables m are also used according to $e0$, $e1$, and um'. Take for a concrete example $x{:=}$**case** *xs* **of** $nil{\to}y \mid cons\,h\,r{\to}x{+}h$:

- In the *nil* case, it is as though we had $x{:=}y$, which would receive the annotation

  $ux{=:}\boxdot\,.\;uy{=:}uy'{\sqcup}ux'\,.\;uxs{=:}uxs'$

  In addition, if x' is used to some degree, then *xs* is used as much as to find that it is *nil*, so the annotation is increased to

  $ux{=:}\boxdot\,.\;uy{=:}uy'{\sqcup}ux'\,.\;uxs{=:}uxs'{\sqcup}(ux'{\rhd}nil)$

- In the *cons* case, it is as though we had $x{:=}x{+}h$, which would receive the annotation (treating h as a constant for a moment)

  $ux{=:}ux'{\rhd}x\,.\;uxs{=:}uxs'$

  In addition, if x' is used to some degree, then *xs* is used further in two ways. Firstly, it is then used as much as to find that it is *cons*:

  $ux{=:}ux'{\rhd}x\,.\;uxs{=:}uxs'{\sqcup}(ux'{\rhd}cons\;\boxdot\;\boxdot)$

  Secondly, h is part of *xs*, and now is time to account for its usage due to $x{:=}x{+}h$. If h were a memory variable (though it is a local constant), its usage would be annotated as $uh{=:}uh'{\sqcup}(ux'{\rhd}h)$. From this we deduce that the corresponding component of *cons* has usage $ux'{\rhd}h$:

  $ux{=:}ux'{\rhd}x\,.\;uxs{=:}uxs'{\sqcup}(ux'{\rhd}cons\;(ux'{\rhd}h)\;\boxdot)$

  This example can be simplified to

  $ux{=:}ux'{\rhd}x\,.\;uxs{=:}uxs'{\sqcup}(ux'{\rhd}cons\;h\;\boxdot)$

The annotated assignment statement is therefore:

$ux{=:}$**case** *xs* **of** $nil{\to}\boxdot \mid cons\,h\,r{\to}ux'{\rhd}x.$

$uy =: \textbf{case}\ xs\ \textbf{of}\ nil \to uy' \sqcup ux' \mid cons\ h\ r \to uy'$.

$uxs =: \textbf{case}\ xs\ \textbf{of}\ nil \to uxs' \sqcup (ux' \rhd nil) \mid cons\ h\ r \to uxs' \sqcup (ux' \rhd cons\ h\ \boxdot)$.

$x := \textbf{case}\ xs\ \textbf{of}\ nil \to y \mid cons\ h\ r \to x+h$

More generally, for $v := \textbf{case}\ xs\ \textbf{of}\ nil \to e0 \mid cons\ h\ r \to e1$, where v may or may not be xs:

- for memory variables m other than xs (such as v, x, y), add

$$\begin{aligned} um =: \textbf{case}\ xs\ \textbf{of}\ \ & nil \to (\text{usage of } m \text{ in } v := e0) \\ \mid\ & cons\ h\ r \to (\text{usage of } m \text{ in } v := e1) \end{aligned}$$

- for xs, add

$$\begin{aligned} uxs =: \textbf{case}\ xs\ \textbf{of}\ \ & nil \to (\text{usage of } xs \text{ in } v := e0) \sqcup (uv' \rhd nil) \\ \mid\ & cons\ h\ r \to (\text{usage of } xs \text{ in } v := e1) \sqcup (uv' \rhd cons\ hu\ ru) \end{aligned}$$

  where hu and ru are usages of h and r respectively in $v := e1$. To determine hu, suppose h were a memory variable (though it is a local constant), then its usage annotation due to $v := e1$ should look like $uh =: uh' \sqcup hu$, from which we can extract hu. Similarly for r.

This works even if xs is the variable assigned to and an operand in $e0$ or $e1$, for example
$xs := \textbf{case}\ xs\ \textbf{of}\ nil \to nil \mid cons\ h\ r \to cons\ 1\ xs$
is annotated by

$$\begin{aligned} & uxs =: \textbf{case}\ xs\ \textbf{of}\ \ nil \to (\text{usage of } xs \text{ in } xs := nil) \sqcup (uxs' \rhd nil) \\ & \qquad\qquad\qquad \mid\ cons\ h\ r \to (\text{usage of } xs \text{ in } xs := cons\ 1\ xs) \sqcup (uxs' \rhd cons\ hu\ ru) \\ = \ & uxs =: \textbf{case}\ xs\ \textbf{of}\ \ nil \to uxs' \rhd nil \\ & \qquad\qquad\qquad \mid\ cons\ h\ r \to tail\ uxs' \sqcup (uxs' \rhd cons\ \boxdot\ \boxdot) \end{aligned}$$

Generalizing to other lazy algebraic types is straightforward.

4.2.5 Conditional Expressions

This subsection covers assignments that use if-then-else expressions such as

$$x{:=}\textbf{if }y{=}0\textbf{ then }1\textbf{ else }2{\times}y$$

(Conditional statements such as

$$\textbf{if }y{=}0\textbf{ then }x{:=}1\textbf{ else }x{:=}2{\times}y$$

are treated in Section 4.2.7.)

In $x{:=}\textbf{if }y{=}0\textbf{ then }e0\textbf{ else }e1$, if x' is used to some degree ($ux'{\neq}\boxdot$), then y is used as much as to determine whether $y{=}0$ is $\top$ or $\bot$, and in addition according to $e0$, $e1$, and uy'. Other memory variables m are also used according to $e0$, $e1$, and um'. Using the concrete example $x{:=}\textbf{if }y{=}0\textbf{ then }1\textbf{ else }2{\times}y$:

- For x, one case is as though we had $x{:=}1$, and the other case is as though we had $x{:=}2{\times}y$:
  $ux{=:}\textbf{if }y{=}0\textbf{ then }\boxdot\textbf{ else }\boxdot$
  which can be simplified to
  $ux{=:}\boxdot$

- For y, one case is as though we had $x{:=}1$, and the other case is as though we had $x{:=}2{\times}y$:
  $uy{=:}\textbf{if }y{=}0\textbf{ then }uy'\textbf{ else }uy'{\sqcup}(ux'{\triangleright}y)$
  In addition, if x' is used to some degree, then y is used as much as to compute $y{=}0$, so this must be increased by $ux'{\triangleright}y$
  $uy{=:}(ux'{\triangleright}y){\sqcup}(\textbf{if }y{=}0\textbf{ then }uy'\textbf{ else }uy'{\sqcup}(ux'{\triangleright}y))$
  which can be simplified to
  $uy{=:}(ux'{\triangleright}y){\sqcup}uy'$

The annotated assignment statement is therefore

$$ux{=:}\boxdot\ .\ uy{=:}(ux'{\triangleright}y){\sqcup}uy'\ .\ x{:=}\textbf{if }y{=}0\textbf{ then }1\textbf{ else }2{\times}y$$

Other examples may be less simplifiable.

In general, $v{:=}\textbf{if }e\textbf{ then }e0\textbf{ else }e1$ is annotated this way: each memory variable m (including v) receives

$um{=}{:}\quad (uv'\rhd(\text{usage of } m \text{ in } e))$

$\quad\sqcup (\textbf{if } e \textbf{ then } (\text{usage of } m \text{ in } v{:=}e0) \textbf{ else } (\text{usage of } m \text{ in } v{:=}e1))$

Ordering of these usage assignments may be important so that each receives the intended value of uv'; usually the assignment for uv should be left-most.

Usage of m in e can be calculated this way: Image a new memory variable b and the assignment $b{:=}e$, which would be annotated like $um{=}{:}um'\sqcup(ub'\rhd \textit{stuff})$, and the answer is *stuff* (extracted by asserting $ub'\neq\boxdot$ and $um'{=}\boxdot$). Checking this with the previous example, $b{:=}(y{=}0)$ would be annotated by $uy{:=}uy'\sqcup(ub'\rhd y)$, from which y is extracted.

Here are three more examples:

Example 4.1 $x{:=}\textbf{if } y{=}0 \textbf{ then } 1 \textbf{ else } x{+}2$

after annotation is

$ux{=}{:}\textbf{if } y{=}0 \textbf{ then } \boxdot \textbf{ else } ux'\rhd x\,.$

$uy{=}{:}(ux'\rhd y)\sqcup(\textbf{if } y{=}0 \textbf{ then } uy' \textbf{ else } uy')\,.$

$x{:=}\textbf{if } y{=}0 \textbf{ then } 1 \textbf{ else } x{+}2$

which can be simplified to

$ux{=}{:}\textbf{if } y{=}0 \textbf{ then } \boxdot \textbf{ else } ux'\rhd x\,.$

$uy{=}{:}(ux'\rhd y)\sqcup uy'\,.$

$x{:=}\textbf{if } y{=}0 \textbf{ then } 1 \textbf{ else } x{+}2$

□

Example 4.2 $x{:=}\textbf{if } x{=}0 \textbf{ then } y \textbf{ else } x{+}2$

after annotation is

$ux{=}{:}(ux'\rhd x)\sqcup(\textbf{if } x{=}0 \textbf{ then } \boxdot \textbf{ else } ux'\rhd x)\,.$

$uy{=}{:}\textbf{if } x{=}0 \textbf{ then } uy'\sqcup ux' \textbf{ else } uy'\,.$

$x{:=}\textbf{if } x{=}0 \textbf{ then } y \textbf{ else } x{+}2$

which can be simplified to

$ux =: ux' \rhd x$.

$uy =:$ **if** $x=0$ **then** $uy' \sqcup ux'$ **else** uy' .

$x :=$ **if** $x=0$ **then** y **else** $x+2$

□

Example 4.3 In this example, the condition may use one part of the list in *xs*, and the rest of the assignment may use other parts.

$xs :=$ **if** $0=($**case** xs **of** $nil \rightarrow 0 \mid cons\ h\ r \rightarrow h)$ **then** xs

else case xs **of** $nil \rightarrow nil \mid cons\ h\ r \rightarrow r$

Doing it slowly, we need to determine usage in the condition, in $xs:=xs$, and in $xs:=$**case** xs **of** $nil \rightarrow nil \mid cons\ h\ r \rightarrow r$.

Usage of *xs* in the condition is $uxs' \rhd$**case** xs **of** $nil \rightarrow nil \mid cons\ h\ r \rightarrow cons\ h\ \boxdot$.

Usage of *xs* in $xs:=xs$ is uxs'.

Usage of *xs* in $xs:=$**case** xs **of** $nil \rightarrow nil \mid cons\ h\ r \rightarrow r$ is **case** xs **of** $nil \rightarrow uxs' \rhd nil \mid cons\ h\ r \rightarrow uxs' \rhd cons\ \boxdot\ uxs'$.

Therefore, the complete annotation is

$uxs =:\quad (uxs' \rhd$**case** xs **of** $nil \rightarrow nil \mid cons\ h\ r \rightarrow cons\ h\ \boxdot)$

$\sqcup$ (**if** $0=($**case** xs **of** $nil \rightarrow 0 \mid cons\ h\ r \rightarrow h)$ **then** uxs'

else case xs **of** $nil \rightarrow uxs' \rhd nil \mid cons\ h\ r \rightarrow uxs' \rhd cons\ \boxdot\ uxs')$.

$xs :=$ **if** $0=($**case** xs **of** $nil \rightarrow 0 \mid cons\ h\ r \rightarrow h)$ **then** xs

else case xs **of** $nil \rightarrow nil \mid cons\ h\ r \rightarrow r$

which can be somewhat simplified to

$uxs =: uxs' \rhd ($**case** xs **of** $\ nil \rightarrow nil$

$\mid cons\ h\ r \rightarrow$ **if** $0=h$

```
                                        **then** $cons\,h\,\boxdot\sqcup uxs'$
                                        **else**  $cons\,h\,uxs')$ .
    $xs$:=**if** $0$=(**case** $xs$ **of** $nil\rightarrow 0\,|\,cons\,h\,r\rightarrow h$) **then** $xs$
       **else case** $xs$ **of** $nil\rightarrow nil\,|\,cons\,h\,r\rightarrow r$
```

□

4.2.6 General Principle

There is a general principle of usage transformation underlying the above basic operations, which can help design or even derive usage transformation in all cases. We begin by recalling that in algebraic data constructions, where the operation is *cons*, the usage transformation consists of the two inverses, *head* and *tail*. Primitive operations can also be seen in this light with a relaxed sense of inverse, e.g., where the operation is $x{+}y$, the usage transformation seems to be inverting $x{+}y$ back into its operands x and y. This pattern is much less obvious but still present in algebraic data case analyses. It seems that where the operation is $x{:=}f\,x\,y$, the usage transformation inverts it, so that $ux'=f\,ux\,uy$ (relaxed to $ux'\sqsubseteq f\,ux\,uy$ for reasons explained below).

This principle arises from the following consideration. In an operation $x{:=}f\,x\,y$ (which contains $x'=f\,x\,y\wedge y'{=}y$), f is not only mathematically defined but also computationally postulated with operand usage (by our wish or by computer specification). For example we postulate:

- $f\,x\,y=x{+}y$ uses both operands completely
- $f\,x\,y=cons\,x\,y$ itself does not use its operands at all (all uses come from external demands)
- $f\,x\,y=$(**case** x **of** $nil\rightarrow\top\,|\,cons\,h\,t\rightarrow\bot$) uses x only as much as telling its tag and does not use y at all
- $f\,x\,y=$(**case** x **of** $nil\rightarrow 0\,|\,cons\,h\,t\rightarrow h{+}1$) uses x to tell its tag, and in the *cons* case the first component completely for +

We formalize these postulates by extending f or its underlying operations (e.g., +, *cons*, case) to take incomplete value parameters and return possibly incomplete value answers. To express how much an operation uses an operand, we add a definition expressing that the operation returns $\boxdot$ when an incomplete operand lacks what it uses (but otherwise works fine). To formalize the above examples:

- $\boxdot+y=\boxdot \wedge x+\boxdot=\boxdot \wedge \boxdot+\boxdot=\boxdot$ (the last conjunct is redundant if we allow x and y to stand for incomplete values too)
- *cons*: no further definitions: operands are unused
- $(\textbf{case}\ \boxdot\ \textbf{of}\ nil{\to}e0 \mid cons\,h\,t{\to}e1)=\boxdot$
- $(\textbf{case}\ cons\,\boxdot\,nil\ \textbf{of}\ nil{\to}0 \mid cons\,h\,t{\to}h{+}1)=\boxdot$

  (this does not need to be postulated independently; it just combines ordinary case analysis and $\boxdot+y=\boxdot$)

Extending f this way enables the following formalization. When deriving usage transformation, we seek usage pre-values ux and uy that are sufficient and necessary to supply what f uses to produce as much non-$\boxdot$ parts as in ux'. That is, sufficiency means that ux and uy are large enough so that $f\,ux\,uy$ is in turn large enough compared to ux', i.e., $ux' \sqsubseteq f\,ux\,uy$; necessity means that ux and uy are the smallest under sufficiency. (Necessity can be formalized with fairly little gain, so we keep it as an informal side condition.)

Note that usually sufficiency and necessity do not combine to $ux' = f\,ux\,uy$ for reasons such as:

- with $x' = cons\,x\,y$, sufficiency is $ux' \sqsubseteq cons\,ux\,uy$; we can have $ux'=\boxdot$, so $ux' \sqsubset cons\,ux\,uy$
- with $x' = x{+}y \wedge y'=y$, sufficiency is $ux' \sqsubseteq ux{+}uy \wedge uy' \sqsubseteq uy$; we can have $ux'=5$ forcing $uy \neq \boxdot$, and $uy'=\boxdot$ so $uy' \sqsubset uy$

4.2.7 Branching

In this subsection, we treat branching statements (if-then-else and case-of) at the program level, as control-flow constructs that select from alternative programs, rather than as expressions in assignment statements. For two examples:

if y=0 **then** x:=0 **else** y:=0

case xs **of** $nil \rightarrow (z{:=}0) \mid cons\ h\ r \rightarrow (y{:=}y{+}h\ .\ xs{:=}r)$

These pose several possible degrees of laziness, of which some important choices are:

- High precision and laziness: knowing the precise usage information of the branches, if all mentioned memory variables are unused (but other variables may be used), execution skips evaluating the branching condition and selecting one branch. Using the if-then-else example above, if x' and y' are unused and a third variable z' is used, the condition y=0 is unevaluated, since a precise analysis determines that neither x:=0 nor y:=0 affects z' or other post-values.
- Speculative execution: concurrently execute both branches; only when their results are significantly different (the difference affects the demanded parts), evaluate the branching condition.
- Low precision and laziness: not knowing the precise usage information of the branches, execution plays safe by evaluating the branching condition and selecting one branch whenever any memory variable (post-value) in scope is used; skipping happens only when all variables in scope are unused. Using again the if-then-else example above, if z' is used, then even though x' and y' are unused and neither branch affects z', y=0 is still evaluated to select a branch. This reflects the choice that the compiler or interpreter does not bother to analyze the branches, which may be difficult to analyze if they contain recursive calls.

In this thesis, we adopt the latter, which is also the choice of programming languages such as Haskell. We note that other choices could be achieved with aggressive analyses and/or programmer-provided information, with corresponding usage annotation and operational semantics.

Our choice implies that branching statements are less lazy than equivalent assignments using branching expressions. Take the following pair for example:

if y=0 **then** x:=0 **else** x:=1

x:=**if** y=0 **then** 0 **else** 1

Although both programs give the same answers in x' and y', the former (if-then-else statement) is less lazy: if a third variable z' is used, then y is used. The latter (assignment with if-then-else expression) does not use y if z' is used but x' and y' are both unused.

We begin by annotating **if** e **then** P **else** Q. Assume that P and Q are already annotated, and consider the following. If any post-value in scope (say x', y', or z') is used, the memory variables in e are first used as much as to get an answer ($\top$ or $\bot$), so for example uy is at least $(ux', uy', uz')\triangleright$(how e uses y); and then further usage is as per the selected alternative (P or Q), so for example uy is also at least what P or Q requires. This is represented by (say x, y, and z are the only memory variables in scope):

$$\begin{aligned}\exists ux'', uy'', uz'' \cdot\quad & \textbf{if } e \textbf{ then } (\text{subst } ux'',uy'',uz'' \text{ for } ux,uy,uz \text{ in } P)\\ & \textbf{else } (\text{subst } ux'',uy'',uz'' \text{ for } ux,uy,uz \text{ in } Q)\\ & \wedge\ ux = ux'' \sqcup ((ux', uy', uz')\triangleright(\text{how } e \text{ uses } x))\\ & \wedge\ uy = uy'' \sqcup ((ux', uy', uz')\triangleright(\text{how } e \text{ uses } y))\\ & \wedge\ uz = uz'' \sqcup ((ux', uy', uz')\triangleright(\text{how } e \text{ uses } z))\end{aligned}$$

In general, a local name ux'' is introduced existentially to help capture the value of ux governed by P and Q, and the equation

$$ux = ux'' \sqcup ((\text{all usage post-values})\triangleright(\text{how } e \text{ uses } x))$$

is added as a conjunct to state that the overall ux combines the contribution of P or Q and the contribution of e. This huge expression is simplifiable in some cases: Notably, if e does not use x at all, the local name ux'' and the equation for ux can be eliminated:

$$\begin{aligned}& \exists ux'', uy'', uz'' \cdot\quad (\text{the rest})\\ & \qquad \wedge\ ux = ux'' \sqcup ((ux', uy', uz')\triangleright(\text{how } e \text{ uses } x))\\ = \quad & \langle e \text{ does not use } x \text{ at all}\rangle\\ & \exists ux'', uy'', uz'' \cdot\quad (\text{the rest})\\ & \qquad \wedge\ ux = ux'' \sqcup ((ux', uy', uz')\triangleright\boxdot)\\ = \quad & \langle ux'' \sqcup ((ux', uy', uz')\triangleright\boxdot) = ux'' \sqcup \boxdot = ux''\rangle\end{aligned}$$

$\exists ux'', uy'', uz'' \cdot$ (the rest)

$\land\ ux{=}ux''$

$=$ ⟨predicate calculus⟩

$\exists uy'', uz'' \cdot$ (the rest, with ux'' restored to ux)

The part "how e uses m" can be calculated this way: Image a new memory variable b and the assignment $b{:=}e$, which would be annotated like $um{=}{:}\,um' \sqcup (ub' \triangleright \mathit{stuff})$ for each memory variable m in e; then *stuff* is the answer, i.e., simulate $ub' {\neq} \boxdot$ and $um'{=}\boxdot$.

Example 4.4 We annotate **if** $y{=}0$ **then** $x{:=}0$ **else** $y{:=}0$. The memory variables are x, y, and z.
$x{:=}0$ itself after annotation is $ux{=}{:}\boxdot\,.\,x{:=}0$
$y{:=}0$ itself after annotation is $uy{=}{:}\boxdot\,.\,y{:=}0$
The branching condition $y{=}0$ uses y fully but does not use x or z at all.
Therefore, the complete annotation is the following, followed by simplification:

$\exists uy'' \cdot$ **if** $y{=}0$ **then** (subst uy'' for uy in $ux{=}{:}\boxdot\,.\,x{:=}0$)

else (subst uy'' for uy in $uy{=}{:}\boxdot\,.\,y{:=}0$)

$\land\ uy{=}uy'' \sqcup ((ux', uy', uz') \triangleright y)$

$=$ ⟨distribute⟩

if $y{=}0$ **then** $\exists uy'' \cdot$ (subst uy'' for uy in $ux{=}{:}\boxdot\,.\,x{:=}0$) $\land\ uy{=}uy'' \sqcup ((ux', uy', uz') \triangleright y)$

else $\exists uy'' \cdot$ (subst uy'' for uy in $uy{=}{:}\boxdot\,.\,y{:=}0$) $\land\ uy{=}uy'' \sqcup ((ux', uy', uz') \triangleright y)$

$=$ ⟨expand assignment and sequential composition⟩

if $y{=}0$ **then** $\exists uy'' \cdot ux{=}\boxdot \land uy''{=}uy' \land uz{=}uz' \land x'{=}0 \land y'{=}y \land z'{=}z \land uy{=}uy'' \sqcup ((ux', uy', uz') \triangleright y)$

else $\exists uy'' \cdot ux{=}ux' \land uy''{=}\boxdot \land uz{=}uz' \land x'{=}x \land y'{=}0 \land z'{=}z \land uy{=}uy'' \sqcup ((ux', uy', uz') \triangleright y)$

$=$ ⟨predicate calculus⟩

if $y{=}0$ **then** $ux{=}\boxdot \land uz{=}uz' \land x'{=}0 \land y'{=}y \land z'{=}z \land uy{=}uy' \sqcup ((ux', uy', uz') \triangleright y)$

else $ux{=}ux' \land uz{=}uz' \land x'{=}x \land y'{=}0 \land z'{=}z \land uy{=}\boxdot \sqcup ((ux', uy', uz') \triangleright y)$

$=$ ⟨$\boxdot \sqcup u{=}u$;

$uy' \sqcup ((ux', uy', uz') \triangleright y) = (ux', uy', uz') \triangleright y$ assuming $uy' \sqsubseteq y'$ and in context $y'{=}y$⟩

$\textbf{if } y{=}0 \textbf{ then } ux{=}\boxdot \wedge uz{=}uz' \wedge x'{=}0 \wedge y'{=}y \wedge z'{=}z \wedge uy{=}(ux', uy', uz')\rhd y$

$\textbf{else } ux{=}ux' \wedge uz{=}uz' \wedge x'{=}x \wedge y'{=}0 \wedge z'{=}z \wedge uy{=}(ux', uy', uz')\rhd y$

$=$ ⟨contract to assignment and sequential composition⟩

$\textbf{if } y{=}0 \textbf{ then } (ux{=}{:}\boxdot\,.\; uy{=}{:}(ux', uy', uz')\rhd y\,.\; x{:=}0)$

$\textbf{else } (uy{=}{:}(ux', uy', uz')\rhd y\,.\; y{:=}0)$

□

We now turn to case-of statements, say $\textbf{case } xs \textbf{ of } nil{\to}P \mid cons\,h\,r{\to}Q$. These are like if-then-else with an additional requirement. If any post-value in scope (say xs', y', or z') is used, then xs is first used as much as to select a branch (nil or $cons\,\boxdot\,\boxdot$, but not $\boxdot$), so uxs is at least

$$(uxs', uy', uz')\rhd\textbf{case } xs \textbf{ of } nil{\to}nil \mid cons\,h\,r{\to}cons\,\boxdot\,\boxdot$$

and then further usage is as per the selected alternative (P or Q), so uxs is also at least what P or Q requires. Now comes the additional requirement: Q may use xs by directly referring to xs or by indirectly referring to its parts h and r. To help capture the indirect use, we introduce local usage pre-values uh and ur for h and r respectively, and we require the usage annotation of Q to include specifying uh and ur. We skip uh' and ur', replacing them by $\boxdot$ to reflect that there is no h' or r' to be used. With this addition, we can now annotate the overall case-of:

$$\begin{aligned}\exists uxs'', uh, ur \cdot \quad & \textbf{case } xs \textbf{ of } \; nil{\to}(\text{subst } uxs'' \text{ for } uxs \text{ in } P)\\ & \qquad\qquad \mid cons\,h\,r{\to}(\text{subst } uxs'' \text{ for } uxs \text{ in } Q)\\ & \wedge\; uxs{=}uxs''\sqcup((uxs', uy', uz')\rhd\textbf{case } xs \textbf{ of } nil{\to}nil \mid cons\,h\,r{\to}cons\,uh\,ur)\end{aligned}$$

Example 4.5 We annotate $\textbf{case } xs \textbf{ of } nil{\to}(z{:=}0) \mid cons\,h\,r{\to}(y{:=}y{+}h\,.\; xs{:=}r)$. The memory variables are xs, y, and z.

$z{:=}0$ itself is annotated as $uz{=}{:}\boxdot\,.\; z{:=}0$

$y{:=}y{+}h\,.\; xs{:=}r$ itself is annotated, with uh and ur, as

$uy{=}{:}uy'\rhd y\,.\; uxs{=}{:}\boxdot\,.\; uh{=}{:}uy'\rhd h\,.\; ur{=}{:}uxs'\,.\; y{:=}y{+}h\,.\; xs{:=}r$

Note that $y{:=}y{+}h$ would normally lead to $uh{=}{:}uh'\sqcup(uy'\rhd h)$, and we replace uh' by $\boxdot$; similarly

$xs{:=}r$ would normally lead to $ur{=:}ur'{\sqcup}uxs'$, and we replace ur' by $\boxdot$.

The overall annotation is:

$$\begin{array}{ll}\exists uxs'', uh, ur \cdot & \textbf{case}\ xs\ \textbf{of}\ \ nil{\to}(\text{subst}\ uxs''\ \text{for}\ uxs\ \text{in}\ uz{=:}\boxdot\,.\,z{:=}0) \\ & \qquad\qquad\ \mid\ cons\, h\, r{\to}(\text{subst}\ uxs''\ \text{for}\ uxs\ \text{in} \\ & \qquad\qquad\ uy{=:}uy'{\rhd}y\,.\,uxs{=:}\boxdot\,.\,uh{=:}uy'{\rhd}h\,.\,ur{=:}uxs'\,.\,y{:=}y{+}h\,.\,xs{:=}r) \\ & \wedge\ uxs{=}uxs''{\sqcup}((uxs', uy', uz'){\rhd}\textbf{case}\ xs\ \textbf{of}\ nil{\to}nil \mid cons\, h\, r{\to}cons\, uh\, ur)\end{array}$$

□

4.2.8 Adding and Hiding Variables

The construct **var** $z \cdot P$ creates a local memory variable z in the scope of P; this local memory variable is not in scope outside the **var** construct. For lazy programs, each memory variable comes with a pre-value, a post-value, a pre-usage, and a post-usage; so we re-define the **var** construct as:

$$\textbf{var}\ z{:}\,T \cdot P\ \ =\ \ \exists z, z'{:}\,T \cdot \exists uz, uz'{:}\,TE \cdot P$$

where TE refers to the extended type for T. We often leave both T and TE implicit and just write **var** $z \cdot P$.

Usage annotation of **var** $z \cdot P$ is performed as normally done to P, including uz and uz'. In addition, P is sequentially followed by $uz{=:}\boxdot$ to reflect that z' cannot be used outside. So **var** $z \cdot P$ after annotation becomes:

$$\textbf{var}\ z \cdot (P\ \text{annotated})\,.\,uz{=:}\boxdot$$

We choose to make the tailing $uz{=:}\boxdot$ explicit, rather than part of the **var** package, because we often take out the body and calculate on it (e.g., refine, simplify) without the **var**, in which case, (P annotated)$.\,uz{=:}\boxdot$ offers more simplifications than (P annotated) alone.

The construct **scope** $x \cdot P$ hides from P memory variables and usage variables other than those for x, so the scope of P has the memory variable x (along with usage) and the time variable t only. P can read and write x only, and cannot read or write other memory variables; these other memory variables are unchanged. Hiding fewer variables is possible by listing more variables in the construct, e.g., **scope** $x, y \cdot P$ retains x, y, and t in the scope of P. We re-define the **scope** construct by extrapolating from this example: if the memory variables already in scope are w, x, y, and z before entering the **scope** construct, then we re-define:

$$\mathbf{scope}\, x, y \cdot P \;=\; P \wedge w'{=}w \wedge z'{=}z \wedge uw{=}uw' \wedge uz{=}uz'$$

Usage annotation of **scope** $x, y \cdot P$ is performed as normally done to P, including ux, ux', uy, uy', and excluding uw, uw', uz, uz'.

Useful theorems on **var** and **scope** listed in Chapter 2 still hold with the re-definitions for lazy programs. In addition, we state one more on **scope** and backward assignment:

- $(\mathbf{scope}\, m \cdot P.\, u{=}{:}e) \;=\; (\mathbf{scope}\, m \cdot P).\, u{=}{:}e$

  $(\mathbf{scope}\, m \cdot u{=}{:}e.\, P) \;=\; u{=}{:}e.\, (\mathbf{scope}\, m \cdot P)$

  provided u is in scope and e does not mention any variable forbidden by the **scope**

4.3 Lazy Recursive Time

Recursive calls pose several possible degrees of laziness, similar to branching statements in the previous section:

- High precision and laziness: Knowing precisely which memory variables are and are not affected by a recursive call, execution enters or skips the recursive call accordingly.

- Low precision and laziness: Not knowing which memory variables are and are not affected by a recursive call, execution plays safe by entering the recursive call when any memory variable in scope is used, and skipping the recursive call otherwise.

Again as with branching statements, we adopt the latter choice in this thesis, and note that higher precision and laziness is possible with aggressive analysis and/or programmer-provided information.

Consequently, recursive time under lazy evaluation is marked as follows. Each recursive call takes 1 unit time if any of the usage variables in scope is not $\boxdot$, and 0 units time otherwise. This can be written as the following time increment, supposing that the usage variables in scope are *ux* and *uy*:

$$t{:=}t{+}\mathbf{if}\ ux'{\neq}\boxdot \lor uy'{\neq}\boxdot\ \mathbf{then}\ 1\ \mathbf{else}\ 0$$

We sequentially compose this after each recursive call. (It is after for lazy programs so that *ux′* and *uy′* refers to the desired usage.) So for example, the recursive call to *R* is annotated as:

$$R\,.\,t{:=}t{+}\mathbf{if}\ ux'{\neq}\boxdot \lor uy'{\neq}\boxdot\ \mathbf{then}\ 1\ \mathbf{else}\ 0$$

The interaction between recursive calls and **scope** constructs is most noteworthy. Suppose the memory variables are *x*, *y*, *z*, and a **scope** statement shrinks that to *x*, *y*, inside which a recursive call *R* occurs:

$$z{:=}0\,.\,\mathbf{scope}\,x,y\cdot x{:=}0\,.\,R$$

The recursive call occurs with only *x* and *y* in scope, and so the above program is annotated as:

$$uz{=:}\boxdot\,.\,z{:=}0\,.\,\mathbf{scope}\,x,y\cdot ux{=:}\boxdot\,.\,x{:=}0\,.\,R\,.\,t{:=}t{+}\mathbf{if}\ ux'{\neq}\boxdot \lor uy'{\neq}\boxdot\ \mathbf{then}\ 1\ \mathbf{else}\ 0$$

The body of **scope**, including the recursive program *R* stands for, has memory access to *x* and *y* only; if they are unused, the recursive call is skipped. Execution bypasses it and proceeds to *z*:=0 if *uz′* so requires. In other words, the program takes 0 recursive time:

$$uz{=:}\boxdot\,.\,z{:=}0\,.\,(\mathbf{scope}\,x,y\cdot ux{=:}\boxdot\,.\,x{:=}0\,.\,R\,.\,t{:=}t{+}\mathbf{if}\ ux'{\neq}\boxdot \lor uy'{\neq}\boxdot\ \mathbf{then}\ 1\ \mathbf{else}\ 0)\,.\,ux{=:}\boxdot\,.\,uy{=:}\boxdot$$

$\Rightarrow$ ⟨several steps and weakening⟩

$$t'{=}t$$

We do not necessarily consider all calls to be recursive calls, just like in the eager case. In examples in this thesis, we typically have a main program that initializes and then calls a helper, and we do not have a time increment for this call; but the helper calls itself, and we have a time increment for this call.

4.4 Automatic Annotation of Usage and Time

We have described the necessary time increments and usage variable assignments to be inserted into refinements for accounting of lazy recursive time. These annotations can be added mechanically, and our description above is close to an informal algorithm. We can get this far because:

- For data operations, strictness is known, e.g., $x+y$ is fully strict in both operands, and $cons\, x\, xs$ is non-strict in both operands. This determines the corresponding usage assignments.
- For branching statements, we have chosen low precision and laziness (Section 4.2.7). Usage annotation is determined by the memory variables in scope.
- For recursive calls, we have also chosen low precision and laziness (Section 4.3). Time increment is determined by the memory variables in scope.

If we change branching statements and recursive calls to have higher precision and laziness, then their annotations become harder or less mechanical, requiring more analysis or information from the programmer.

In specifications to be refined, especially those refined recursively, usage and timing parts cannot be automatically determined in general, but well-understood subclasses of practical interest can be automated.

4.5 Small Example

A major application of lazy evaluation is the construction of infinite data structures to be consumed finitely. For example, an infinite cons-list is created by a recursive definition, and then only the first cons cell is ever used. The computer should spend no more time than is necessary for the construction of the needed cons cell.

An infinite cons-list of 0's, written as $(\mu s \cdot cons\, 0\, s)$ below, may be created by a program like

$$xs'{=}(\mu s \cdot cons\, 0\, s) \Longleftarrow xs'{=}(\mu s \cdot cons\, 0\, s)\, .\, xs{:=}cons\, 0\, xs$$

The unusual position of the recursion is derived from a Haskell program for the same task:

```
p :: () -> [Int]
p = (0:) . p
```

i.e., functional composition $f \circ g$ typically becomes sequential composition $g\,.\,f$.

(The above solution—both renditions—takes n recursive time to fulfill a demand for $n{+}1$ cons cells. This is of course not the cheapest solution; the cheapest solution just takes 1 step to build 1 self-referencing cons cell. We choose the expensive solution for an easy example of on-demand unlimited recursive time.)

The program augmented with usage and timing is then:

with memory variable xs:

$$Repeat \Leftarrow Repeat\,.\,t{:=}t{+}\mathbf{if}\ uxs'{\neq}\boxdot\ \mathbf{then}\ 1\ \mathbf{else}\ 0\ .\ uxs{=:}tail\,uxs'\,.\,xs{:=}cons\,0\,xs$$

where

$$Repeat = uxs'{\sqsubseteq}xs' \Rightarrow xs'{=}(\mu s \cdot cons\,0\,s) \wedge t' = t + ulen\,(tail\,uxs')$$

In the specification of $Repeat$, we add the assumption $uxs'{\sqsubseteq}xs'$, which is always fulfilled in practice, and is needed in the proof below.

The refinement can be proved this way:

$Repeat\,.\,t{:=}t{+}\mathbf{if}\ uxs'{\neq}\boxdot\ \mathbf{then}\ 1\ \mathbf{else}\ 0\ .$

$uxs{=:}tail\,uxs'\,.\,xs{:=}cons\,0\,xs$

$=$ ⟨definition of $Repeat$⟩

$uxs'{\sqsubseteq}xs' \Rightarrow xs'{=}(\mu s \cdot cons\,0\,s) \wedge t' = t + ulen\,(tail\,uxs')\ .$

$t{:=}t{+}\mathbf{if}\ uxs'{\neq}\boxdot\ \mathbf{then}\ 1\ \mathbf{else}\ 0\ .$

$uxs{=:}tail\,uxs'\,.\,xs{:=}cons\,0\,xs$

$\Rightarrow$ ⟨2.1.3-engulf-assignment (for $t{:=}t{+}\ldots$), simplify⟩

$uxs'{\sqsubseteq}xs' \Rightarrow xs'{=}(\mu s \cdot cons\,0\,s) \wedge t' = t + ulen\,(tail\,uxs') + (\mathbf{if}\ uxs'{\neq}\boxdot\ \mathbf{then}\ 1\ \mathbf{else}\ 0)\ .$

$uxs{=}{:}\mathit{tail}\,uxs'\,.\,xs{:=}\mathit{cons}\,0\,xs$

$=$ ⟨in context $uxs' \sqsubseteq xs' {=} (\mu s \cdot \mathit{cons}\,0\,s)$, simplify time bound⟩

$uxs' \sqsubseteq xs' \;\Rightarrow\; xs' {=} (\mu s \cdot \mathit{cons}\,0\,s) \wedge t' = t + \mathit{ulen}\,uxs'\;.$

$uxs{=}{:}\mathit{tail}\,uxs'\,.\,xs{:=}\mathit{cons}\,0\,xs$

$=$ ⟨4.2.1-assignment-after (for $uxs{=}{:}\mathit{tail}\,uxs'$)⟩

$\mathit{tail}\,uxs' \sqsubseteq xs' \;\Rightarrow\; xs' {=} (\mu s \cdot \mathit{cons}\,0\,s) \wedge t' = t + \mathit{ulen}\,(\mathit{tail}\,uxs')\;.$

$xs{:=}\mathit{cons}\,0\,xs$

$\Rightarrow$ ⟨2.1.3-engulf-assignment (for $xs{:=}\mathit{cons}\,0\,xs$), simplify⟩

$\mathit{tail}\,uxs' \sqsubseteq \mathit{tail}\,xs' \;\Rightarrow\; xs' {=} \mathit{cons}\,0\,(\mu s \cdot \mathit{cons}\,0\,s) \;\wedge\; t' = t + \mathit{ulen}\,(\mathit{tail}\,uxs')$

$\Rightarrow$ ⟨$\mathit{tail}\,uxs' \sqsubseteq \mathit{tail}\,xs' \;\Leftarrow\; uxs' \sqsubseteq xs'$⟩

$uxs' \sqsubseteq xs' \;\Rightarrow\; xs' {=} \mathit{cons}\,0\,(\mu s \cdot \mathit{cons}\,0\,s) \;\wedge\; t' = t + \mathit{ulen}\,(\mathit{tail}\,uxs')$

$=$ ⟨definition of *Repeat*⟩

Repeat

We can write a consumer that tests whether the produced *xs* is empty or not, which consumes one cons cell (and does not use its components). For simplicity, we assert right here that the answer is used and the list is not used further:

with memory variables *xs* and *y*:

$\mathit{Null} \;=\; uy{=}{:}\boxdot\,.\; uxs{=}{:}uxs' \sqcup (uy' \triangleright \mathbf{case}\; xs\; \mathbf{of}\; \mathit{nil} {\to} \mathit{nil} \mid \mathit{cons}\,h\,t {\to} \mathit{cons}\,\boxdot\,\boxdot)\;.$

$\qquad y{:=}(\mathbf{case}\; xs\; \mathbf{of}\; \mathit{nil} {\to} \top \mid \mathit{cons}\,h\,t {\to} \bot)\;.$

$\qquad uy{=}{:}y\,.\; uxs{=}{:}\boxdot$

This consumer can be simplified to:

$uy{=}{:}\boxdot\,.\; uxs{=}{:}uxs' \sqcup (uy' \triangleright \mathbf{case}\; xs\; \mathbf{of}\; \mathit{nil} {\to} \mathit{nil} \mid \mathit{cons}\,h\,t {\to} \mathit{cons}\,\boxdot\,\boxdot)\;.$

$y{:=}\mathbf{case}\; xs\; \mathbf{of}\; \mathit{nil} {\to} \top \mid \mathit{cons}\,h\,t {\to} \bot\;.$

$uy{=}{:}y\,.\; uxs{=}{:}\boxdot$

$=$ ⟨simplify sequential composition (merge 2nd and 3rd lines)⟩

$uy{=}{:}\boxdot\ .\ uxs{=}{:}uxs' \sqcup (uy' \rhd \textbf{case}\ xs\ \textbf{of}\ nil{\to}nil \mid cons\ h\ t{\to}cons\ \boxdot\ \boxdot)\ .$

$xs'{=}xs \wedge y'{=}(\textbf{case}\ xs\ \textbf{of}\ nil{\to}\top \mid cons\ h\ t{\to}\bot) \wedge uy{=}y \wedge uxs{=}\boxdot \wedge t'{=}t$

$=$ ⟨simplify sequential composition⟩

$xs'{=}xs \wedge y'{=}(\textbf{case}\ xs\ \textbf{of}\ nil{\to}\top \mid cons\ h\ t{\to}\bot)$

$\wedge\ uy{=}\boxdot \wedge uxs{=}(\textbf{case}\ xs\ \textbf{of}\ nil{\to}nil \mid cons\ h\ t{\to}cons\ \boxdot\ \boxdot) \wedge t'{=}t$

When composing *Repeat* with *Null*, exactly 0 time unit should be spent in *Repeat* (1 top-level non-recursive call and 0 recursive calls to produce 1 cons cell). We can prove:

$(\textbf{scope}\, xs \cdot Repeat).\, Null$

$=$ ⟨definitions of *Repeat* and **scope**; *Null* as calculated above⟩

$(uxs' {\sqsubseteq} xs'\ \Rightarrow\ xs'{=}\mu s \cdot cons\ 0\ s \wedge t' = t{+}ulen\,(tail\ uxs')) \wedge y'{=}y \wedge uy{=}uy'\ .$

$xs'{=}xs \wedge y'{=}(\textbf{case}\ xs\ \textbf{of}\ nil{\to}\top \mid cons\ h\ t{\to}\bot)$

$\wedge\ uy{=}\boxdot \wedge uxs{=}(\textbf{case}\ xs\ \textbf{of}\ nil{\to}nil \mid cons\ h\ t{\to}cons\ \boxdot\ \boxdot) \wedge t'{=}t$

$=$ ⟨definition of sequential composition⟩

$\exists xs'', y'', uxs'', uy'' \cdot\ (uxs'' {\sqsubseteq} xs''\ \Rightarrow\ xs''{=}\mu s \cdot cons\ 0\ s \wedge t'' = t{+}ulen\,(tail\ uxs''))$

$\wedge\ y''{=}y \wedge uy{=}uy''$

$\wedge\ xs'{=}xs'' \wedge y'{=}(\textbf{case}\ xs''\ \textbf{of}\ nil{\to}\top \mid cons\ h\ t{\to}\bot)$

$\wedge\ uy''{=}\boxdot \wedge uxs''{=}(\textbf{case}\ xs''\ \textbf{of}\ nil{\to}nil \mid cons\ h\ t{\to}cons\ \boxdot\ \boxdot) \wedge t'{=}t''$

$=$ ⟨in context: $uxs''{=}(\textbf{case}\ xs''\ \textbf{of}\ nil{\to}nil \mid cons\ h\ t{\to}cons\ \boxdot\ \boxdot) {\sqsubseteq} xs''$⟩

$\exists xs'', y'', uxs'', uy'' \cdot\ xs''{=}\mu s \cdot cons\ 0\ s \wedge t'' = t{+}ulen\,(tail\ uxs'') \wedge y''{=}y \wedge uy{=}uy''$

$\wedge\ xs'{=}xs'' \wedge y'{=}(\textbf{case}\ xs''\ \textbf{of}\ nil{\to}\top \mid cons\ h\ t{\to}\bot)$

$\wedge\ uy''{=}\boxdot \wedge uxs''{=}(\textbf{case}\ xs''\ \textbf{of}\ nil{\to}nil \mid cons\ h\ t{\to}cons\ \boxdot\ \boxdot) \wedge t'{=}t''$

$=$ ⟨in context: xs'' is a *cons*⟩

$\exists xs'', y'', uxs'', uy'' \cdot\ xs''{=}\mu s \cdot cons\ 0\ s \wedge t'' = t{+}ulen\,(tail\ uxs'') \wedge y''{=}y \wedge uy{=}uy''$

$\wedge\ xs'{=}xs'' \wedge y'{=}\bot \wedge uy''{=}\boxdot \wedge uxs''{=}cons\ \boxdot\ \boxdot \wedge t'{=}t''$

$=$ ⟨predicate calculus⟩

$xs'{=}\mu s \cdot cons\, 0\, s \wedge y'{=}\bot \wedge uy{=}\boxdot \wedge t' = t + ulen\, (tail\, (cons\, \boxdot\, \boxdot))$

$=$ ⟨definitions of *ulen* and *tail*⟩

$xs'{=}\mu s \cdot cons\, 0\, s \wedge y'{=}\bot \wedge uy{=}\boxdot \wedge t' = t{+}0$

This kind of reasoning is compositional with respect to program structure: the proof of the refinement of *Repeat* is independent of *Null*, the simplification of *Null* is independent of *Repeat*, and calculating their sequential composition requires just their respective specifications, not their implementation details.

4.6 Larger Example

In this example, we have a less trivial pair of producer and consumer, involving most programming constructs introduced. First the producer: It produces an infinite list consisting of the positive integers 1, 2… in that order.

with memory variable *xs*:

$xs' = from\, 1 \;\Leftarrow\; \mathbf{var}\, c \cdot c{:=}1\, .\, xs' = from\, c$

with memory variables *c* and *xs*:

$xs' = from\, c \;\Leftarrow\; \mathbf{var}\, c0 \cdot c0{:=}c\, .\, c{:=}c{+}1\, .\, (\mathbf{scope}\, c, xs \cdot xs' = from\, c)\, .\, xs{:=}cons\, c0\, xs$

where we define

$from\, k = cons\, k\, (from\, (k{+}1))$

for infinite lists of natural numbers from a given start

We focus on the second, recursive refinement. After usage annotation and guessing a specification, it is:

with memory variables *c* and *xs*:

$Pos \;\Leftarrow\; \mathbf{var}\, c0 \cdot uc0{=}{:}\boxdot\, .\, uc{=}{:}uc' \sqcup uc0'\, .\, c0{:=}c\, .$

$uc\,{=}{:}\,uc' \rhd c$. $c := c+1$.

(**scope** $c, xs \cdot Pos$. $t := t+$**if** $uc' \neq \boxdot \lor uxs' \neq \boxdot$ **then** 1 **else** 0) .

$uxs\,{=}{:}\,tail\; uxs'$. $uc0\,{=}{:}\,uc0' \sqcup head\; uxs'$. $xs := cons\; c0\; xs$.

$uc0\,{=}{:}\,\boxdot$

where

$Pos \;=\; uc' = \boxdot \land uxs' \sqsubseteq xs' \;\Rightarrow\; t' = t + ulen\,(tail\; uxs') \land xs' = from\; c$

We give our specification *Pos* the precondition $uc' = \boxdot$ as we know it from the context (inside a **var** c), and the precondition $uxs' \sqsubseteq xs'$ again. Having these right here shortens the specification (e.g., so we do not bother to say what happens if $uc' \neq \boxdot$) and helps simplifications in the proof of the refinement.

The proof: first the right-hand side without the **var** $c0$:

$uc0\,{=}{:}\,\boxdot$. $uc\,{=}{:}\,uc' \sqcup uc0'$. $c0 := c$.

$uc\,{=}{:}\,uc' \rhd c$. $c := c+1$.

(**scope** $c, xs \cdot Pos$. $t' := t+$**if** $uc' \neq \boxdot \lor uxs' \neq \boxdot$ **then** 1 **else** 0) .

$uxs\,{=}{:}\,tail\; uxs'$. $uc0\,{=}{:}\,uc0' \sqcup head\; uxs'$. $xs := cons\; c0\; xs$.

$uc0\,{=}{:}\,\boxdot$

$=$ ⟨definition of *Pos*⟩

$uc0\,{=}{:}\,\boxdot$. $uc\,{=}{:}\,uc' \sqcup uc0'$. $c0 := c$.

$uc\,{=}{:}\,uc' \rhd c$. $c := c+1$.

(**scope** $c, xs \cdot uc' = \boxdot \land uxs' \sqsubseteq xs' \;\Rightarrow\; t' = t + ulen\,(tail\; uxs') \land xs' = from\; c$.

$t := t+$**if** $uc' \neq \boxdot \lor uxs' \neq \boxdot$ **then** 1 **else** 0) .

$uxs\,{=}{:}\,tail\; uxs'$. $uc0\,{=}{:}\,uc0' \sqcup head\; uxs'$. $xs := cons\; c0\; xs$.

$uc0\,{=}{:}\,\boxdot$

$\Rightarrow$ ⟨2.1.3-engulf-assignment (for $t := t+\ldots$), simplify⟩

$uc0\,{=}{:}\,\boxdot$. $uc\,{=}{:}\,uc' \sqcup uc0'$. $c0 := c$.

$uc =: uc' \rhd c \ .\ c := c+1\ .$

$(\textbf{scope}\ c, xs \cdot uc' = \boxdot \wedge uxs' \sqsubseteq xs' \Rightarrow$

$t' = t + ulen\ (tail\ uxs') + (\textbf{if}\ uxs' \neq \boxdot\ \textbf{then}\ 1\ \textbf{else}\ 0) \wedge xs' = from\ c)\ .$

$uxs =: tail\ uxs'\ .\ uc0 =: uc0' \sqcup head\ uxs'\ .\ xs := cons\ c0\ xs\ .$

$uc0 =: \boxdot$

$=$ ⟨simplify time bound⟩

$uc0 =: \boxdot\ .\ uc =: uc' \sqcup uc0'\ .\ c0 := c\ .$

$uc =: uc' \rhd c\ .\ c := c+1\ .$

$(\textbf{scope}\ c, xs \cdot uc' = \boxdot \wedge uxs' \sqsubseteq xs' \Rightarrow t' = t + ulen\ uxs' \wedge xs' = from\ c)\ .$

$uxs =: tail\ uxs'\ .\ uc0 =: uc0' \sqcup head\ uxs'\ .\ xs := cons\ c0\ xs\ .$

$uc0 =: \boxdot$

$\Rightarrow$ ⟨definition of **scope**; weaken⟩

$uc0 =: \boxdot\ .\ uc =: uc' \sqcup uc0'\ .\ c0 := c\ .$

$uc =: uc' \rhd c\ .\ c := c+1\ .$

$uc' = \boxdot \wedge uxs' \sqsubseteq xs' \Rightarrow t' = t + ulen\ uxs' \wedge xs' = from\ c \wedge c0' = c0 \wedge uc0 = uc0'\ .$

$uxs =: tail\ uxs'\ .\ uc0 =: uc0' \sqcup head\ uxs'\ .\ xs := cons\ c0\ xs\ .$

$uc0 =: \boxdot$

$=$ ⟨2.1.3-assignment-before (for $c := c+1$)⟩

$uc0 =: \boxdot\ .\ uc =: uc' \sqcup uc0'\ .\ c0 := c\ .$

$uc =: uc' \rhd c\ .$

$uc' = \boxdot \wedge uxs' \sqsubseteq xs' \Rightarrow t' = t + ulen\ uxs' \wedge xs' = from\ (c+1) \wedge c0' = c0 \wedge uc0 = uc0'\ .$

$uxs =: tail\ uxs'\ .\ uc0 =: uc0' \sqcup head\ uxs'\ .\ xs := cons\ c0\ xs\ .$

$uc0 =: \boxdot$

$\Rightarrow$ ⟨4.2.1-engulf-assignment (for $uc =: uc' \rhd c$), simplify⟩

$uc0 =: \boxdot\ .\ uc =: uc' \sqcup uc0'\ .\ c0 := c\ .$

$uc'=\boxdot \land uxs' \sqsubseteq xs' \ \Rightarrow\ t' = t + ulen\ uxs' \ \land\ xs' = from\,(c+1) \ \land\ c0' = c0 \ \land\ uc0 = uc0'$.

$uxs =: tail\ uxs'$. $uc0 =: uc0' \sqcup head\ uxs'$. $xs := cons\ c0\ xs$.

$uc0 =: \boxdot$

$\Rightarrow$ ⟨2.1.3-assignment-before (for $c0:=c$),

4.2.1-engulf-assignment (for $uc =: uc' \sqcup uc0'$ then $uc0 =: \boxdot$),

simplify⟩

$uc'=\boxdot \land uxs' \sqsubseteq xs' \ \Rightarrow\ t' = t + ulen\ uxs' \ \land\ xs' = from\,(c+1) \ \land\ c0' = c \ \land\ uc0 = \boxdot$.

$uxs =: tail\ uxs'$. $uc0 =: uc0' \sqcup head\ uxs'$. $xs := cons\ c0\ xs$.

$uc0 =: \boxdot$

$\Rightarrow$ ⟨4.2.1-assignment after (for $uxs =: tail\ uxs'$ then $uc0 =: uc0' \sqcup head\ uxs'$),⟩

$uc'=\boxdot \land tail\ uxs' \sqsubseteq xs' \ \Rightarrow\ t' = t + ulen\,(tail\ uxs') \ \land\ xs' = from\,(c+1) \ \land\ c0' = c \ \land\ uc0 = \boxdot$.

$xs := cons\ c0\ xs$.

$uc0 =: \boxdot$

$=$ ⟨expand sequential composition, predicate calculus⟩

$(\exists xs'' \cdot \quad (uc'=\boxdot \land tail\ uxs' \sqsubseteq xs'' \Rightarrow$

$\qquad t' = t + ulen\,(tail\ uxs') \land xs'' = from\,(c+1) \land c0' = c \land uc0 = \boxdot)$

$\quad \land\ xs' = cons\ c0'\ xs'')$.

$uc0 =: \boxdot$

$=$ ⟨in context $xs' = cons\ c0'\ xs''$, $xs'' = tail\ xs'$⟩

$(\exists xs'' \cdot \quad (uc'=\boxdot \land tail\ uxs' \sqsubseteq tail\ xs' \Rightarrow$

$\qquad t' = t + ulen\,(tail\ uxs') \land xs'' = from\,(c+1) \land c0' = c \land uc0 = \boxdot)$

$\quad \land\ xs' = cons\ c0'\ xs'')$.

$uc0 =: \boxdot$

$\Rightarrow$ ⟨weaken: $(p \Rightarrow q) \land r\ \Rightarrow\ p \Rightarrow q \land r$⟩

$(\exists xs'' \cdot uc'=\boxdot \land tail\ uxs' \sqsubseteq tail\ xs' \Rightarrow$

$$t' = t + ulen\,(tail\,uxs') \wedge xs'' = from\,(c+1) \wedge c0' = c \wedge uc0 = \boxdot \wedge xs' = cons\,c0'\,xs'') .$$

$uc0 =: \boxdot$

$=$ ⟨predicate calculus⟩

$uc' = \boxdot \wedge tail\,uxs' \sqsubseteq tail\,xs' \Rightarrow$

$t' = t + ulen\,(tail\,uxs') \wedge xs' = cons\,c\,(from\,(c+1)) \wedge c0' = c \wedge uc0 = \boxdot$.

$uc0 =: \boxdot$

$\Rightarrow$ ⟨$tail\,uxs' \sqsubseteq tail\,xs' \Leftarrow uxs' \sqsubseteq xs'$; definition of $from$⟩

$uc' = \boxdot \wedge uxs' \sqsubseteq xs' \Rightarrow t' = t + ulen\,(tail\,uxs') \wedge xs' = from\,c \wedge c0' = c \wedge uc0 = \boxdot$.

$uc0 =: \boxdot$

$=$ ⟨4.2.1-assignment after⟩

$uc' = \boxdot \wedge uxs' \sqsubseteq xs' \Rightarrow t' = t + ulen\,(tail\,uxs') \wedge xs' = from\,c \wedge c0' = c \wedge uc0 = \boxdot$

and therefore with the **var** $c0$:

$\textbf{var}\ c0 \cdot uc0 =: \boxdot \,.\, uc =: uc' \sqcup uc0' \,.\, c0 := c$.

$uc =: uc' \rhd c \,.\, c := c+1$.

$(\textbf{scope}\ c, xs \cdot Pos \,.\, t := t + \textbf{if}\ uc' \neq \boxdot \vee uxs' \neq \boxdot\ \textbf{then}\ 1\ \textbf{else}\ 0)$.

$uxs =: tail\,uxs' \,.\, uc0 =: uc0' \sqcup head\,uxs' \,.\, xs := cons\,c0\,xs$.

$uc0 =: \boxdot$

$\Rightarrow$ ⟨the calculation above, 2.3.1-var-mono⟩

$\textbf{var}\ c0 \cdot uc' = \boxdot \wedge uxs' \sqsubseteq xs' \Rightarrow t' = t + ulen\,(tail\,uxs') \wedge xs' = from\,c \wedge c0' = c \wedge uc0 = \boxdot$

$=$ ⟨definition of **var**⟩

$\exists c0, c0', uc0, uc0' \cdot uc' = \boxdot \wedge uxs' \sqsubseteq xs' \Rightarrow t' = t + ulen\,(tail\,uxs') \wedge xs' = from\,c \wedge c0' = c \wedge uc0 = \boxdot$

$=$ ⟨predicate calculus⟩

$uc' = \boxdot \wedge uxs' \sqsubseteq xs' \Rightarrow t' = t + ulen\,(tail\,uxs') \wedge xs' = from\,c$

$=$ ⟨definition of Pos⟩

Pos

The complete producer is

$\mathbf{var}\, c \cdot uc{=}{:}\boxdot\, .\, c{:=}1\, .\, Pos\, .\, uc{=}{:}\boxdot$

after usage annotation. We can simplify it to see the overall effect:

$\mathbf{var}\, c \cdot uc{=}{:}\boxdot\, .\, c{:=}1\, .\, Pos\, .\, uc{=}{:}\boxdot$

$=$ ⟨definition of *Pos*⟩

$\mathbf{var}\, c \cdot uc{=}{:}\boxdot\ .\ c{:=}1\ .$

$uc'{=}\boxdot \wedge uxs' {\sqsubseteq} xs' \ \Rightarrow\ t' = t + ulen\,(tail\, uxs') \wedge xs' = from\, c\ .$

$uc{=}{:}\boxdot$

$=$ ⟨2.1.3-assignment-before, 4.2.1-assignment-after⟩

$\mathbf{var}\, c \cdot uc{=}{:}\boxdot\ .$

$uc'{=}\boxdot \wedge uxs' {\sqsubseteq} xs' \ \Rightarrow\ t' = t + ulen\,(tail\, uxs') \wedge xs' = from\, 1$

$=$ ⟨expand sequential composition, predicate calculus⟩

$\mathbf{var}\, c \cdot uc{=}\boxdot \wedge (uxs' {\sqsubseteq} xs' \ \Rightarrow\ t' = t + ulen\,(tail\, uxs') \wedge xs' = from\, 1)$

$=$ ⟨definition of **var**; predicate calculus⟩

$uxs' {\sqsubseteq} xs' \ \Rightarrow\ t' = t + ulen\,(tail\, uxs') \wedge xs' = from\, 1$

The complete producer or its recursive part promises an infinite list but takes only as much recursive time as the length of the actually used prefix.

We now program a consumer: It sums the first *n* items in the given list *xs* (or all of the list if it has fewer than *n* items).

with memory variables *xs*, *n* (type *nat*), and *s*:

$s' = add\, n\, xs \ \Leftarrow\ s{:=}0\ .\ s' = s + add\, n\, xs$

```
$s' = s + add\, n\, xs \ \Leftarrow$ $\mathbf{if}\ n{=}0\ \mathbf{then}\ ok$
                        $\mathbf{else\ case}\ xs\ \mathbf{of}\ \ nil \to ok$
                                 $\mid\ cons\, h\, r \to (s{:=}s{+}h\ .\ xs{:=}r\ .\ n{:=}n{-}1\ .\ s' = s + add\, n\, xs)$
```

where we define

$add\,0\,r = 0$

$add\,(k{+}1)\,nil = 0$

$add\,(k{+}1)\,(cons\,h\,r) = h + add\,k\,r$

for the sum of a list up to a given item count

Again, focusing on the second, recursive refinement, we annotate usage and guess a specification:

$Sum \Leftarrow$

$\exists un'' \cdot \quad un = un'' \sqcup ((uxs', un', us') \triangleright n)$

$\quad \wedge$ **if** $n{=}0$ **then** (subst un'' for un in ok)

$\quad$ **else** $\exists uxs'', uh, ur \cdot \quad uxs = uxs'' \sqcup ((uxs', un', us') \triangleright$**case** xs **of** $\;nil \to nil$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid cons\,h\,r \to cons\,uh\,ur)$

$\qquad\qquad \wedge$ **case** xs **of** $\;nil \to$(subst un'', uxs'' for un, uxs in ok)

$\qquad\qquad\qquad \mid cons\,h\,r \to$(subst un'', uxs'' for un, uxs in

$\qquad\qquad\qquad us =: us' \triangleright s\,.\; uh =: us' \triangleright h\,.\; s := s{+}h\,.$

$\qquad\qquad\qquad uxs =: \boxdot\,.\; ur =: uxs'\,.\; xs := r\,.$

$\qquad\qquad\qquad un =: un' \triangleright n\,.\; n := n{-}1\,.$

$\qquad\qquad\qquad Sum\,.$

$\qquad\qquad\qquad t := t{+}$**if** $uxs' {\neq} \boxdot \vee un' {\neq} \boxdot \vee us' {\neq} \boxdot$ **then** 1 **else** 0)

where

$Sum = us' {\neq} \boxdot \wedge uxs' {=} \boxdot \;\Rightarrow\; s' = s + add\,n\,xs \wedge t' = t + min\,n\,(ulen\,xs) \wedge uxs = utake\,n\,xs$

$utake\,0\,r = \boxdot$

$utake\,(k{+}1)\,nil = nil$

$utake\,(k{+}1)\,(cons\,h\,r) = cons\,h\,(utake\,k\,r)$

so $utake\,n\,xs$ indicates using up to n items of xs but no more

In our specification for *Sum*, we add the preconditions $us'{\neq}\boxdot$ and $uxs'{=}\boxdot$ to reflect our intended use of this program: we will use the final sum s' but not the final list xs' (in other words, we do not use the list any further than what *Sum* uses). Having these preconditions shortens the specification to what we want to show here and helps simplifications in the proof of the refinement. And as it happens, if s' is used, then whether n' is used or not does not matter.

To prove the refinement, we distribute $\wedge$ and $\exists$ into if-then-else and case-of (2.1.3-if-distribution and 2.1.4-case-distribution), then apply 2.3.1-refine-by-if and 2.3.1-refine-by-case to split the refinement into 3 smaller refinements, each under its respective assumption:

Assuming $n{=}0$:

$\exists un'' \cdot \quad un = un'' \sqcup ((uxs', un', us') \rhd n)$

$\quad \wedge$ (subst un'' for un in ok)

$=$ $\quad\langle$definition of ok and the substitution$\rangle$

$\exists un'' \cdot \quad un = un'' \sqcup ((uxs', un', us') \rhd n)$

$\quad \wedge\ xs'{=}xs \wedge n'{=}n \wedge s'{=}s \wedge t'{=}t \wedge uxs{=}uxs' \wedge un''{=}un' \wedge us{=}us'$

$\Rightarrow$ $\quad\langle$weaken, predicate calculus$\rangle$

$s'{=}s \wedge t'{=}t \wedge uxs{=}uxs'$

$\Rightarrow$ $\quad\langle$weaken$\rangle$

$us'{\neq}\boxdot \wedge uxs'{=}\boxdot \Rightarrow s'{=}s \wedge t'{=}t \wedge uxs{=}uxs'$

$\Rightarrow$ $\quad\langle$in context: $n{=}0$ and $uxs'{=}\boxdot$; so $uxs{=}\boxdot{=}utake\,0\,xs$$\rangle$

$us'{\neq}\boxdot \wedge uxs'{=}\boxdot \Rightarrow s' = s{+}add\,n\,xs \wedge t' = t{+}min\,n\,(ulen\,xs) \wedge uxs{=}utake\,n\,xs$

$=$ $\quad\langle$definition of *Sum*$\rangle$

Sum

Assuming $n{\neq}0$ and $xs{=}nil$:

$\exists un'' \cdot \quad un = un'' \sqcup ((uxs', un', us') \rhd n)$

$\land\ \exists uxs'', uh, ur \cdot \quad uxs = uxs'' \sqcup ((uxs', un', us') \rhd nil)$

$\land\ (\text{subst } un'', uxs'' \text{ for } un, uxs \text{ in } ok)$

= ⟨definition of *ok* and the substitution⟩

$\exists un'' \cdot \quad un = un'' \sqcup ((uxs', un', us') \rhd n)$

$\land\ \exists uxs'', uh, ur \cdot \quad uxs = uxs'' \sqcup ((uxs', un', us') \rhd nil)$

$\land\ xs' = xs \land n' = n \land s' = s \land t' = t \land uxs'' = uxs' \land un'' = un' \land us = us'$

⇒ ⟨weaken, predicate calculus⟩

$s' = s \land t' = t \land uxs = uxs' \sqcup ((uxs', un', us') \rhd nil)$

⇒ ⟨weaken⟩

$us' \neq \boxdot \land uxs' = \boxdot \Rightarrow s' = s \land t' = t \land uxs = uxs' \sqcup ((uxs', un', us') \rhd nil)$

= ⟨in context: $us' \neq \boxdot \land uxs' = \boxdot$⟩

$us' \neq \boxdot \land uxs' = \boxdot \Rightarrow s' = s \land t' = t \land uxs = nil$

= ⟨in context: $xs = nil$; so $ulen\, xs = 0$ and $nil = utake\, n\, xs$⟩

$us' \neq \boxdot \land uxs' = \boxdot \Rightarrow s' = s + add\, n\, xs \land t' = t + min\, n\, (ulen\, xs) \land uxs = utake\, n\, xs$

= ⟨definition of *Sum*⟩

Sum

Assuming $n \neq 0 \land xs = cons\, h\, r$, and so $n > 0$: first simplify without the existential quantifier:

$us =: us' \rhd s\ .\ uh =: us' \rhd h\ .\ s := s + h\ .$

$uxs =: \boxdot\ .\ ur =: uxs'\ .\ xs := r\ .$

$un =: un' \rhd n\ .\ n := n - 1\ .$

$Sum\ .$

$t := t + \textbf{if}\ uxs' \neq \boxdot \lor un' \neq \boxdot \lor us' \neq \boxdot\ \textbf{then}\ 1\ \textbf{else}\ 0$

= ⟨definition of *Sum*⟩

$us =: us' \rhd s\ .\ uh =: us' \rhd h\ .\ s := s + h\ .$

$uxs =: \boxdot\ .\ ur =: uxs'\ .\ xs := r\ .$

$un{=:}un'\rhd n\ .\ n{:=}n{-}1\ \ .$

$us'{\neq}\boxdot\wedge uxs'{=}\boxdot\ \Rightarrow\ s'=s+add\,n\,xs\ \wedge\ t'=t+min\,n\,(ulen\,xs)\ \wedge\ uxs=utake\,n\,xs\ \ .$

$t{:=}t{+}\mathbf{if}\ uxs'{\neq}\boxdot\vee un'{\neq}\boxdot\vee us'{\neq}\boxdot\ \mathbf{then}\ 1\ \mathbf{else}\ 0$

$\Rightarrow$ ⟨2.1.3-engulf-assignment, simplify⟩

$us{=:}us'\rhd s\ .\ uh{=:}us'\rhd h\ .\ s{:=}s{+}h\ \ .$

$uxs{=:}\boxdot\ .\ ur{=:}uxs'\ .\ xs{:=}r\ \ .$

$us'{\neq}\boxdot\wedge uxs'{=}\boxdot\ \Rightarrow\ s'=s+add\,n\,xs\ \wedge\ t'=t+min\,n\,(ulen\,xs)\ \wedge\ uxs=utake\,n\,xs$

$\Rightarrow$ ⟨2.1.3-assignment-before, 4.2.1-engulf-assignment, simplify;
one detail goes $ur{=}uxs''{=}utake\,(n{-}1)\,r$⟩

$us{=:}us'\rhd s\ .\ uh{=:}us'\rhd h\ .\ s{:=}s{+}h\ .$

$us'{\neq}\boxdot\wedge uxs'{=}\boxdot$

$\Rightarrow\ s'=s+add\,(n{-}1)\,r\ \wedge\ t'=t+min\,(n{-}1)\,(ulen\,r)+1\ \wedge\ ur{=}utake\,(n{-}1)\,r\ \wedge\ uxs{=}\boxdot$

$\Rightarrow$ ⟨2.1.3-assignment-before, 4.2.1-engulf-assignment, simplify⟩

$us'{\neq}\boxdot\wedge uxs'{=}\boxdot$

$\Rightarrow\ s'=s+h+add\,(n{-}1)\,r\ \wedge\ t'=t+min\,(n{-}1)\,(ulen\,r)+1\ \wedge\ ur{=}utake\,(n{-}1)\,r\ \wedge\ uxs{=}\boxdot$

$=$ ⟨in context: $n{>}0$, $xs{=}cons\,h\,r$; definitions of *add* and *ulen*⟩

$us'{\neq}\boxdot\wedge uxs'{=}\boxdot$

$\Rightarrow\ s'=s+add\,n\,xs\ \wedge\ t'=t+min\,n\,(ulen\,xs)\ \wedge\ ur{=}utake\,(n{-}1)\,r\ \wedge\ uh{=}h\ \wedge\ uxs{=}\boxdot$

then with the existential quantifier:

$$\begin{aligned}\exists uxs'', uh, ur\cdot\quad & uxs{=}uxs''\sqcup((uxs', un', us')\rhd cons\,uh\,ur)\\ & \wedge\ (us'{\neq}\boxdot\wedge uxs'{=}\boxdot\Rightarrow\quad s'=s+add\,n\,xs\ \wedge\ t'=t+min\,n\,(ulen\,xs)\\ &\qquad\qquad\wedge\ ur{=}utake\,(n{-}1)\,r\ \wedge\ uh{=}h\ \wedge\ uxs''{=}\boxdot)\end{aligned}$$

$\Rightarrow$ ⟨weaken; in context $uxs''{=}\boxdot$, $us'{\neq}\boxdot$⟩

$$\begin{aligned}\exists uxs'', uh, ur\cdot us'{\neq}\boxdot\wedge uxs'{=}\boxdot\Rightarrow\quad & uxs{=}cons\,uh\,ur\ \wedge\ s'=s+add\,n\,xs\ \wedge\ t'=t+min\,n\,(ulen\,xs)\\ & \wedge\ ur{=}utake\,(n{-}1)\,r\ \wedge\ uh{=}h\ \wedge\ uxs''{=}\boxdot\end{aligned}$$

$\Rightarrow$ ⟨predicate calculus⟩

$us'{\neq}\boxdot \land uxs'{=}\boxdot \;\Rightarrow\; uxs{=}cons\, h\,(utake\,(n{-}1)\,r) \land s' = s{+}add\,n\,xs \land t' = t{+}min\,n\,(ulen\,xs)$

$=$ ⟨defintion of *utake*; in context $xs{=}cons\,h\,r$⟩

$us'{\neq}\boxdot \land uxs'{=}\boxdot \;\Rightarrow\; uxs{=}utake\,n\,xs \land s' = s{+}add\,n\,xs \land t' = t{+}min\,n\,(ulen\,xs)$

$=$ ⟨definition of *Sum*⟩

Sum

The complete consumer is $us{=:}\boxdot.\; s{:=}0.\, Sum$ after usage annotation. We can simplify it to see the overall effect too:

$us{=:}\boxdot.\; s{:=}0.\, Sum$

$=$ ⟨definition of *Sum*⟩

$us{=:}\boxdot.\; s{:=}0.\, (us'{\neq}\boxdot \land uxs'{=}\boxdot \;\Rightarrow\; uxs{=}utake\,n\,xs \land s' = s{+}add\,n\,xs \land t' = t{+}min\,n\,(ulen\,xs))$

$=$ ⟨2.1.3-assignment-before, simplify sequential composition⟩

$us{=}\boxdot \land (us'{\neq}\boxdot \land uxs'{=}\boxdot \;\Rightarrow\; uxs{=}utake\,n\,xs \land s'{=}add\,n\,xs \land t' = t{+}min\,n\,(ulen\,xs))$

The complete consumer or its recursive part takes n recursive time (or as much as the length of xs if shorter) to compute the sum, and uses just the portion of the list being summed, provided that the sum is used but the list is not used further.

Putting the producer and the consumer together, and enforcing our intended usage at the end, we have:

with memory variables xs, n, and s:

$(\mathbf{scope}\, xs \cdot \mathbf{var}\, c \cdot uc{=:}\boxdot.\; c{:=}1.\, Pos.\, uc{=:}\boxdot)\,.$

$us{=:}\boxdot.\; s{:=}0.\, Sum\,.$

$uxs{=:}\boxdot.\; us{=:}s$

$=$ ⟨as calculated before⟩

$(\mathbf{scope}\, xs \cdot uxs'{\sqsubseteq}xs' \;\Rightarrow\; t' = t{+}ulen\,(tail\,uxs') \land xs'{=}from\,1)\,.$

$us{=}\boxdot \land (us'{\neq}\boxdot \land uxs'{=}\boxdot \;\Rightarrow\; uxs{=}utake\,n\,xs \land s'{=}add\,n\,xs \land t' = t{+}min\,n\,(ulen\,xs))\,.$

$uxs{=}{:}\boxdot\,.\ us{=}{:}s$

$=\qquad\langle$4.2.1-assignment-after$\rangle$

$(\mathbf{scope}\, xs\cdot uxs'{\sqsubseteq}xs' \Rightarrow t'=t+ulen\,(tail\,uxs')\wedge xs'{=}from\,1)\ .$

$us{=}\boxdot\wedge uxs{=}utake\,n\,xs\wedge s'{=}add\,n\,xs\wedge t'=t+min\,n\,(ulen\,xs)$

$=\qquad\langle$definition of **scope**$\rangle$

$(uxs'{\sqsubseteq}xs' \Rightarrow t'=t+ulen\,(tail\,uxs')\wedge xs'{=}from\,1)\wedge n'{=}n\wedge s'{=}s\wedge un{=}un'\wedge us{=}us'\ .$

$us{=}\boxdot\wedge uxs{=}utake\,n\,xs\wedge s'{=}add\,n\,xs\wedge t'=t+min\,n\,(ulen\,xs)$

$=\qquad\langle$sequential composition in detail$\rangle$

$\exists xs'', n'', s'', uxs'', un'', us''\cdot (uxs''{\sqsubseteq}xs'' \Rightarrow t''=t+ulen\,(tail\,uxs'')\wedge xs''{=}from\,1)\wedge$

$\qquad n''{=}n\wedge s''{=}s\wedge un{=}un''\wedge us{=}us''\wedge$

$\qquad us''{=}\boxdot\wedge uxs''{=}utake\,n''\,xs''\wedge s'{=}add\,n''\,xs''\wedge$

$\qquad t'=t''+min\,n''\,(ulen\,xs'')$

$=\qquad\langle utake\,n''\,xs''{\sqsubseteq}xs'';\ ulen\,(tail\,(utake\,n''\,xs''))=max\,0\,(n''-1);\ ulen\,(from\,1)=\infty\rangle$

$\exists xs'', n'', s'', uxs'', un'', us''\cdot t''=t+max\,0\,(n''-1)\wedge xs''{=}from\,1\wedge$

$\qquad n''{=}n\wedge s''{=}s\wedge un{=}un''\wedge us{=}us''\wedge$

$\qquad us''{=}\boxdot\wedge uxs''{=}utake\,n''\,xs''\wedge s'{=}add\,n''\,xs''\wedge t'{=}t''{+}n''$

$=\qquad\langle$predicate calculus$\rangle$

$us{=}\boxdot\wedge s'{=}add\,n\,(from\,1)\wedge t'=t+n+max\,0\,(n-1)$

$=\qquad\langle$definitions of *add* and *from*; algebra$\rangle$

$us{=}\boxdot\wedge s'{=}n{\times}(n{+}1)/2\wedge t'=t+n+max\,0\,(n-1)$

The composition computes the sum of 1 to n (inclusive) and takes $n+max\,0\,(n-1)$ recursive time, where $max\,0\,(n-1)$ units are spent in the producer and another n are in the consumer.

4.7 Related Work

4.7.1 Lazy UTP (Guttmann)

Guttmann also has a predicative theory of lazy imperative programming [10, 9], with termination (or liveness) rather than time, and with higher-order procedures. In this theory, programs are again relations (or boolean expressions) between pre-values and post-values, but the state space consists of all extended values (whereas our memory state space contains only complete values), and there are further restrictions or healthiness conditions. The most important healthiness conditions are downward closure in pre-values and upward closure in post-values: Predicate P satisfies them iff:

$$\rho \sqsubseteq \sigma \wedge P\,\sigma\,\sigma' \Rightarrow P\,\rho\,\sigma'$$

$$P\,\sigma\,\sigma' \wedge \sigma' \sqsubseteq \tau' \Rightarrow P\,\sigma\,\tau'$$

Recursive programs are defined by greatest (weakest, least refined) fixed points.

Compared to Guttmann's theory, our theory has time but does not have higher-order procedures, its memory state space remains simple, and it uses post-fixed points (refinements) for recursion, so that specifications or summaries of recursive programs do not have to be exact. Post-fixed points together with timing and a soundness theorem (next chapter) is as adequate as fixed points.

4.7.2 Context Analysis (Wadler and Hughes, Sands)

Our usage variables, extended types, and usage transformations are a vast simplification of *context analysis* of Wadler and Hughes, which is used for lazy timing of first-order functional programs [37, 36]. On top of that, Sands adds higher-order functional programs [31, 32], which we omit. Below, we explain context analysis, why it needs its complexity, and why our scheme does not need that complexity.

Both our usage scheme and context analysis begin with the design that demands on answers are represented by incomplete versions of the answers, so that what is omitted in the incomplete version stands for what is unused. But functional programming tries not to refer to the answer directly, and so context analysis does not name and pass around the incomplete version either.

Instead, it abstracts answer usage into a function that maps answers to incomplete versions; such a function is called a *context*. For example, to say that the answer, a cons-list, is used for just its first *nil* or *cons*, a context function α is defined to satisfy

$$\alpha\, nil = nil$$

$$\alpha\,(cons\; h\; t) = cons\; \boxdot\; \boxdot$$

And so $\alpha\,(f\; m)$ is analogous to our um', but it is never used. There is a validity condition $\forall y \cdot \alpha\,(\alpha\; y) = \alpha\; y \sqsubseteq y$, analogous to our $um' \sqsubseteq m'$. Lazy time is an expression in m and α constructed to examine the function α, which essentially lifts examining um' to the function level. To treat the propagation from answer usage to parameter usage, a second-order function Fc that maps contexts for answers to contexts for parameters is derived, under the constraint $\forall x \cdot \alpha\,(f\; x) \sqsubseteq f\,(Fc\; \alpha\; x)$, analogous to our $um' \sqsubseteq f\; um$. Lazy time expressions typically expand to mention $Fc\; \alpha$, which lifts usage transformation to the second-order function level.

The space of incomplete answers in context analysis has one more element $\lightning$ than our extended type, with $\lightning \sqsubset \boxdot$. This extra element serves two technical purposes. First, a context can indicate the rejection of certain answers as wrong answers by mapping them to $\lightning$, e.g., to say that the answer must be a *cons*, not a *nil*, and that the *cons* cell is just used for its being *cons*, a context function α is defined to satisfy

$$\alpha\, nil = \lightning$$

$$\alpha\,(cons\; h\; t) = cons\; \boxdot\; \boxdot$$

Second, context analysis takes $\boxdot$ to mean both "unused" and "no answer" (the latter because of erroneous programs such as $head\; nil = \boxdot$). Some context functions must reject "no answer" as a wrong answer, too, to express strictness, e.g.,

$$\alpha\, \boxdot = \lightning$$

$$\alpha\, nil = \lightning$$

$$\alpha\,(cons\; h\; t) = cons\; \boxdot\; \boxdot$$

As a validity condition, every context maps $\lightning$ to $\lightning$.

Our scheme uses incomplete answers themselves for usage directly because we already refer to complete answers directly; accordingly, our usage transformation needs only be first-order functions over incomplete answers. Our scheme does not need ↯ because predicative programming already provides mechanisms against wrong answers: the specification of a producer asserts postconditions on answers, and the specification of a consumer asserts preconditions on parameters; wrong answers can be neglected or handled arbitrarily. Our scheme takes $\boxdot$ to mean only "unused" because predicative programming already expresses "no answer" by under-specification. Essentially, we have an expressive specification language and we intend to use it.

All in all, in context analysis, the desire to abstract away from the final value leads to lifting usage representation and manipulation by one function order, and together with lack of expressive specifications, also leads to lifting all partial orders involved by one more bottom to account for errors. Both kinds of lifting add complexity and corner cases. Our scheme does not need the abstraction or more error handling, and so it is both one function-order lower and one point lower, skipping a lot of machinery and case analyses.

Chapter 5

Lazy Execution

Lazy execution roughly means that some parts of a program are not executed until demanded (there is a specified root demand to start the process) and the answers thus computed are shared and reused if aliased. In more detail: The environment specifies a root demand on certain post-values of the whole program, or an output command constitutes a root demand on some of the post-values of the program statement just before it. Lazy execution traces backwards for program statements that affect the demanded values, propagating demands to other values; this determines which statements to execute and how many recursive calls to make. The exact rules are in this chapter.

5.1 Lazy Operational Semantics

Our operational semantics is a small-step semantics with a high-level execution state, i.e., a collection of rewrite rules over expressions that look like programs.

In more detail, our execution state is a sequential composition of programs and *bindings*. A *binding* stores memory variables and usage as a conjunction of equations, each equation taking the form *variable′=expression* for memory variables and *variable=expression* for usage variables. An example execution state is

$xs'{=}nil \land y'{=}0\,.$

$y{:=}y{+}1\,.\,xs{:=}cons\;y\;xs\,.$

$$xs'{=}xs \wedge y'{=}y \wedge uy{=}\boxdot \wedge uxs{=}\textbf{case}\ xs\ \textbf{of}\ nil{\to}nil \mid cons\ h\ r{\to}cons\ \boxdot\ \boxdot$$

The binding on the left stores initial values (it lacks usage variables for reasons below). The program to be executed is ($y{:=}y{+}1\ .\ xs{:=}cons\ y\ xs$) (we omit usage and time annotations for reasons below). The binding on the right contains usage and will store final answers; currently it stands for a demand on just the tag of *xs* and no demand on *y*.

An example of execution illustrates how the execution state evolves:

$$xs'{=}nil \wedge y'{=}0\ .$$

$$y{:=}y{+}1\ .\ xs{:=}cons\ y\ xs\ .$$

$$xs'{=}xs \wedge y'{=}y \wedge uy{=}\boxdot \wedge uxs{=}\textbf{case}\ xs\ \textbf{of}\ nil{\to}nil \mid cons\ h\ r{\to}cons\ \boxdot\ \boxdot$$

$\longrightarrow$ $\langle$memory assignment rule (given below)$\rangle$

$$xs'{=}nil \wedge y'{=}0\ .$$

$$y{:=}y{+}1\ .$$

$$xs'{=}cons\ y\ xs \wedge y'{=}y \wedge uy{=}\boxdot \wedge uxs{=}\boxdot$$

and we stop now because all demands are fulfilled: the final *xs* is now known to be a *cons*, leading to no further demand on the penultimate *xs*. So in general, the rightmost binding serves to store incomplete answers, which may actually be adequate to fulfill demands. (We see that it is not needed for eager execution in Chapter 3.) Also, in general, the usage part of the rightmost binding represents demands on the penultimate variables rather than the final ones.

Here is the operation semantics. At the beginning, sequentially compose a binding on the left of the program to store initial values, and sequentially compose a binding on the right for final answers and usage. Let *m* stand for the memory variables and *um* for the usage variables:

$$m'{=}initial\ .\ Main\ .\ m'{=}m \wedge um{=}demand$$

(We either strip annotations in *Main* before execution or keep but ignore them during execution.) In general, an execution state may contain more bindings than the leftmost and the rightmost ones. The execution rules below will show how they come up.

To carry out an execution step, find the rightmost subprogram that matches one of the LHS's of the following rules (they are mutually exclusive), and apply the matching rule to the matching subprogram. Since we strip or ignore annotations, there are no rules for usage backward assignments or time increments. Some rules spawn subexecutions. Each rule also indicates its recursive time cost on its arrow.

- Skip:

$$ok\,.\,m'{=}a \land um{=}d$$
$$\xrightarrow{0}\quad m'{=}a \land um{=}d$$

- Memory assignment:

$$x{:=}e\,.\,m'{=}a \land um{=}d$$
$$\xrightarrow{0}\quad m'{=}(\text{subst } e \text{ for } x \text{ in } a) \land um{=}d1$$

  where $d1$ is updated usage from d and usage transformation according to $x{:=}e$.

  Example:

$$x{:=}y{+}2\;.\;x'{=}x{+}1 \land y'{=}x \land ux{=}x \land uy{=}\boxdot$$
$$\xrightarrow{0}\quad x'{=}y{+}2{+}1 \land y'{=}y{+}2 \land ux{=}\boxdot \land uy{=}y$$

- Recursive call:

$$Label\,.\,m'{=}a \land um{=}d$$
$$\xrightarrow{1}\quad Body\,.\,m'{=}a \land um{=}d$$

  given the refinement $Label{\Leftarrow}Body$

- Non-recursive call: Some calls are not considered recursive calls (e.g., a main program calling a helper just once), and so they do not cost recursive time:

$$Label\,.\,m'{=}a \land um{=}d$$
$$\xrightarrow{0}\quad Body\,.\,m'{=}a \land um{=}d$$

  given the refinement $Label{\Leftarrow}Body$

- Local variable introduction:

$$(\mathbf{var}\, v \cdot P)\,.\; m'{=}a \wedge um{=}d$$

$$\xrightarrow{0}\quad P\,.\; v'{=}v \wedge m'{=}a \wedge uv{=}\boxdot \wedge um{=}d$$

Note: it may be necessary to perform a renaming on **var** $v \cdot P$ before using this rule, so as to avoid name clashes with what's already in $m'{=}a$. Example:

$$(\mathbf{var}\, v \cdot v{:=}x\,.\; x{:=}x{+}1)\,.\; v'{=}x \wedge x'{=}x{+}1 \wedge uv{=}v \wedge ux{=}x$$

$$= \qquad \langle\text{rename}\rangle$$

$$(\mathbf{var}\, w \cdot w{:=}x\,.\; x{:=}x{+}1)\,.\; v'{=}x \wedge x'{=}x{+}1 \wedge uv{=}v \wedge ux{=}x$$

$$\xrightarrow{0} \qquad \langle\text{local variable introduction}\rangle$$

$$w{:=}x\,.\; x{:=}x{+}1\,.\; w'{=}w \wedge v'{=}x \wedge x'{=}x{+}1 \wedge uw{=}\boxdot \wedge uv{=}v \wedge ux{=}x$$

- Scope used: To execute

$$(\mathbf{scope}\, v \cdot P)\,.\; m'{=}a \wedge um{=}d$$

where $um{=}d$ says that some of v is demanded:

$$(\mathbf{scope}\, v \cdot P)\,.\; m'{=}a \wedge um{=}d$$

$$\xrightarrow{0}\quad P\,.\; m'{=}a \wedge um{=}d$$

- Scope unused: Assume that the memory variables are partitioned into v and y. To execute

$$More\,.\; (\mathbf{scope}\, v \cdot P)\,.\; v'{=}av \wedge y'{=}ay \wedge uv{=}\boxdot \wedge uy{=}d$$

where none of v is demanded: first perform a subexecution on

$$More\,.\; v'{=}v \wedge y'{=}y \wedge uv{=}\boxdot \wedge uy{=}d$$

until it stops. Suppose the subexecution can be summarized as

$$\mathit{More}\,.\,v'{=}v \wedge y'{=}y \wedge uv{=}\boxdot \wedge uy{=}d$$

$$\xrightarrow{n}\quad \mathit{More1}\,.\,z'{=}e \wedge v'{=}b \wedge y'{=}c \wedge uz{=}\boxdot \wedge uv{=}\boxdot \wedge uy{=}\boxdot$$

(The extra $z'{=}e$ and uz stand for possible extra variables gained when executing *More*; omit if that does not happen.) Then we continue with

$$\mathit{More}\,.\,(\textbf{scope}\,v\cdot P)\,.\,v'{=}av \wedge y'{=}ay \wedge uv{=}\boxdot \wedge uy{=}d$$

$$\xrightarrow{n}\quad \mathit{More1}\,.\,v1'{=}v \wedge y1'{=}y \wedge z'{=}e \wedge v'{=}b \wedge y'{=}c\,.\,(\textbf{scope}\,v\cdot P)\,.$$

$$v1'{=}v1 \wedge y1'{=}y1$$

$$\wedge\ v'{=}(\text{subst (subst } v1{,}y1 \text{ for } v{,}y \text{ in } c) \text{ for } y \text{ in } av)$$

$$\wedge\ y'{=}(\text{subst (subst } v1{,}y1 \text{ for } v{,}y \text{ in } c) \text{ for } y \text{ in } ay)$$

$$\wedge\ uv{=}\boxdot \wedge uy{=}\boxdot$$

where $v1$ and $y1$ are fresh names.

This requires some explanation. We skip the (**scope** $v\cdot P$) fragement in our execution because v is not demanded, but we keep the fragment for possible future execution (there may be an outer execution re-visiting it later). And we like to copy the result c of y to the rightmost binding; this copying must be done carefully because c may contain references to variables before the **scope** statement; we add the local variables $v1$ and $y1$ to set up that reference correctly.

On the other hand, when $v1$ or $y1$ is unnecessary for the copying, we omit it. Here we show omitting both in the parent execution, if copying c needs neither:

$$\mathit{More}\,.\,(\textbf{scope}\,v\cdot P)\,.\,v'{=}av \wedge y'{=}ay \wedge uv{=}\boxdot \wedge uy{=}d$$

$$\xrightarrow{n}\quad \mathit{More1}\,.\,z'{=}e \wedge v'{=}b \wedge y'{=}c\,.\,(\textbf{scope}\,v\cdot P)\,.$$

$$v'{=}(\text{subst } c \text{ for } y \text{ in } av)$$

$$\wedge\ y'{=}(\text{subst } c \text{ for } y \text{ in } ay)$$

$$\wedge\ uv{=}\boxdot \wedge uy{=}\boxdot$$

Example of needing $v1$ and $y1$: (with abbreviation $ytag = \textbf{case}\ y\ \textbf{of}\ nil \rightarrow nil \mid cons\ h\ r \rightarrow cons\ \boxdot\ \boxdot$)

Execute

$$y{:=}nil\ .\ v{:=}0\ .\ y{:=}cons\ v\ y\ .$$
$$(\textbf{scope}\ v \cdot v{:=}1)\ .$$
$$v'=v{+}1 \wedge y'=y \wedge uv{=}\boxdot \wedge uy{=}ytag$$

Subexecution for scope unused:

– $y{:=}nil\ .\ v{:=}0\ .\ y{:=}cons\ v\ y\ .\ v'=v \wedge y'=y \wedge uv{=}\boxdot \wedge uy{=}ytag$

$\longrightarrow$ ⟨assignment⟩

$y{:=}nil\ .\ v{:=}0\ .\ v'=v \wedge y'=cons\ v\ y \wedge uv{=}\boxdot \wedge uy{=}\boxdot$

Subexecution stops.

Continue:

$$y{:=}nil\ .\ v{:=}0\ .\ y{:=}cons\ v\ y\ .$$
$$(\textbf{scope}\ v \cdot v{:=}1)\ .$$
$$v'=v{+}1 \wedge y'=y \wedge uv{=}\boxdot \wedge uy{=}ytag$$

$\longrightarrow$ ⟨after subexecution for scope unused⟩

$$y{:=}nil\ .\ v{:=}0\ .\ v1'=v \wedge y1'=y \wedge v'=v \wedge y'=cons\ v\ y\ .$$
$$(\textbf{scope}\ v \cdot v{:=}1)\ .$$
$$v1'=v1 \wedge y1'=y1 \wedge v'=v{+}1 \wedge y'=cons\ v1\ y1 \wedge uv{=}\boxdot \wedge uy{=}\boxdot$$

- Conditional branch: to execute

$$More\ .\ (\textbf{if}\ cond\ \textbf{then}\ P\ \textbf{else}\ Q)\ .\ m'=a \wedge um=d$$

first perform a subexecution on

$$More\ .\ m'=m \wedge um=d1$$

where $d1$ has just the demands for deciding *cond*. Suppose the subexecution stops and can be summarized as

$$More\,.\,m'{=}m \wedge um{=}d1$$

$$\xrightarrow{n}\quad More1\,.\,z'{=}bz \wedge m'{=}b \wedge uz{=}\boxdot \wedge um{=}\boxdot$$

(The extra $z'{=}bz$ and uz stand for possible extra variables gained when executing *More*; omit if that does not happen.) Then we continue with

$$More\,.\,(\textbf{if } cond \textbf{ then } P \textbf{ else } Q)\,.\,m'{=}a \wedge um{=}d$$

$$\xrightarrow{n}\quad More1\,.\,z'{=}bz \wedge m'{=}b\,.\,P\,.\,m'{=}a \wedge um{=}d$$

if (subst b for m in *cond*) evaluates to $\top$; or

$$More\,.\,(\textbf{if } cond \textbf{ then } P \textbf{ else } Q)\,.\,m'{=}a \wedge um{=}d$$

$$\xrightarrow{n}\quad More1\,.\,z'{=}bz \wedge m'{=}b\,.\,Q\,.\,m'{=}a \wedge um{=}d$$

if (subst b for m in *cond*) evaluates to $\bot$

- Case branch: to execute

$$More\,.\,(\textbf{case } x \textbf{ of } tag\; w \to P \mid \ldots)\,.\,m'{=}a \wedge um{=}d$$

first perform a subexecution on

$$More\,.\,m'{=}m \wedge um{=}d1$$

where $d1$ just demands the tag of x, e.g., $um{=}d1$ contains

$$ux{=}\textbf{case } x \textbf{ of } tag\; w \to tag\; \boxdot \mid tag2\; h\; r \to tag2\; \boxdot\; \boxdot \mid \ldots$$

Suppose the subexecution stops and can be summerized as

$$More\,.\,m'{=}m \wedge um{=}d1$$

$$\xrightarrow{n}\quad More1\,.\,z'{=}bz \wedge m'{=}b \wedge uz{=}\boxdot \wedge um{=}\boxdot$$

where $m'{=}b$ contains $x'{=}tag\,e$. (The extra $z'{=}bz$ and uz stand for possible extra variables gained when executing *More*; omit if that does not happen.) Then we continue with

$$\begin{array}{ll} & More\,.\,(\textbf{case}\ x\ \textbf{of}\ tag\,w{\rightarrow}P\,|\ldots)\,.\,m'{=}a\wedge um{=}d \\ \xrightarrow{n} & More1\,.\,w'{=}e\wedge z'{=}bz\wedge m'{=}b\,.\,P\,.\,w'{=}w\wedge m'{=}a\wedge uw{=}\boxdot\wedge um{=}d \end{array}$$

It may be necessary to rename w to a fresh name, just like local variable introduction.

This generalizes to tags of other arities in the obvious way.

- Binding merge:

$$\begin{array}{ll} & m'{=}b\,.\,m'{=}a\wedge um{=}d \\ \xrightarrow{0} & m'{=}(\text{subst } b \text{ for } m \text{ in } a)\wedge um{=}d1 \end{array}$$

where $d1$ is updated usage from d by usage transformation according to expressions in b (think of $m'{=}b$ as $m{:=}b$).

Because both main executions and subexecutions may encounter local variable introduction, it is possible that the left binding has extra variables not in the right binding, and vice versa. Suppose the left binding has an extra variable z and the right binding has an extra variable v. The binding merge rule is then

$$\begin{array}{ll} & z'{=}bz\wedge m'{=}b\,.\,v'{=}av\wedge m'{=}a\wedge uv{=}dv\wedge um{=}d \\ \xrightarrow{0} & z'{=}bz\wedge v'{=}(\text{subst } b \text{ for } m \text{ in } av)\wedge m'{=}(\text{subst } b \text{ for } m \text{ in } a)\wedge uz{=}\boxdot\wedge uv{=}dv1\wedge um{=}d1 \end{array}$$

where $dv1$ and $d1$ are updated usage from dv and d by usage transformation according to expressions in b (think of $m'{=}b$ as $m{:=}b$).

Example:

$$\begin{array}{ll} & z'{=}x\wedge x'{=}y{+}2\wedge y'{=}y\,.\,v'{=}x\wedge x'{=}x{+}1\wedge y'{=}x\wedge uv{=}\boxdot\wedge ux{=}x\wedge uy{=}\boxdot \\ \xrightarrow{0} & z'{=}x\wedge v'{=}y{+}2\wedge x'{=}y{+}2{+}1\wedge y'{=}y{+}2\wedge uz{=}\boxdot\wedge uv{=}\boxdot\wedge ux{=}\boxdot\wedge uy{=}y \end{array}$$

The above rules add local variables but do not drop them. The right moment for dropping local variables is subtle: for example, a local variable may become irrelevant in a subexecution, but it may still be relevant in a parent execution. The safety of dropping local variables depends on all ancestor execution states.

- Garbage collection: If a local variable v is no longer mentioned globally, except as v' in a rightmost binding, then it can be dropped:

$$\begin{array}{ll} & v'{=}a \wedge m'{=}A \wedge uv{=}\boxdot \wedge um{=}d \\ \xrightarrow{0} & m'{=}A \wedge um{=}d \end{array}$$

  (We assume that local variables are properly initialized one way or another, so that by the time they can be discarded, there is no pending demand on them.

Expressions in bindings may be ready for evaluation after certain memory assignments and binding merges; conditional branching also involves evaluation. We use these simple evaluations:

- Primitive operations: evaluate when all necessary operands are literals, e.g.,

$$\begin{array}{ll} & 1{+}1 \\ \longrightarrow & 2 \end{array}$$

  Otherwise leave unevaluated, e.g., leave $x{+}1$ as is.

- Conditional expression: evaluate when the condition is a literal:

$$\begin{array}{ll} & \textbf{if } \top \textbf{ then } e0 \textbf{ else } e1 \\ \longrightarrow & e0 \end{array}$$

$$\begin{array}{ll} & \textbf{if } \bot \textbf{ then } e0 \textbf{ else } e1 \\ \longrightarrow & e1 \end{array}$$

  Otherwise leave unevaluated, e.g., leave **if** $x{=}0$ **then** $e0$ **else** $e1$ unevaluated.

- Case analysis expression: evaluate when the argument has its tag exposed, e.g.,

$$\begin{array}{ll} & \textbf{case}\ \mathit{tag}\ e0\ \textbf{of}\ \mathit{tag}\ w{\to}e1\,|\ldots \\ \longrightarrow & \text{(subst } e0 \text{ for } w \text{ in } e1\text{)} \end{array}$$

  Otherwise leave unevaluated, e.g., leave $\textbf{case}\ \mathit{xs}\ \textbf{of}\ \mathit{nil}{\to}e1\,|\ldots$ unevaluated.

An execution or subexecution stops when the rightmost binding has $\mathit{um}{=}\boxdot$ in its usage part, so that there is no pending demand. Of course, as said in the above rules, when a subexecution stops, the parent execution may have to continue (until the parent execution in turn meets the stopping criterion).

We omit annotations during execution (except for usage in the rightmost binding) for several reasons. The initial time is usually inaccessible, making the time variable less useful. Usage annotations of branching statements are too bulky to keep during execution; moreover, actual usage values at any given point of time can be strictly less than those annotations: The annotation has the least upper bound of

- what the branching condition needs
- what the taken branch needs

but as seen in the execution rules, we first spawn a subexecution using only the former, and then we resume with only the latter. It is less useful to carry this annotation around.

5.2 Example

Here is an example of lazy execution, using a pair of producer and consumer from the previous chapter. The given refinements are:

with memory variables c and xs:

$\mathit{Pos} \Leftarrow \textbf{var}\, c0 \cdot c0{:=}c\,.\, c{:=}c{+}1\,.\,(\textbf{scope}\, c, \mathit{xs} \cdot \mathit{Pos})\,.\, \mathit{xs}{:=}\mathit{cons}\, c0\, \mathit{xs}$

with memory variables xs, n (type nat), and s:

$Sum \Leftarrow$ **if** $n{=}0$ **then** ok

else case xs **of** $nil{\to}ok$

$\mid cons\ h\ r{\to}(s{:=}s{+}h\ .\ xs{:=}r\ .\ n{:=}n{-}1\ .\ Sum)$

and the program to be executed is

with memory variables xs, n (type nat), and s:

initial values $n{=}1$, $xs{=}nil$ (should not matter), and $s{=}2$ (should not matter)

(**scope** $xs \cdot$ **var** $c \cdot c{:=}1\,.\,Pos)\,.\,s{:=}0\,.\,Sum$

we demand only the final s

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2\ .$

(**scope** $xs \cdot$ **var** $c \cdot c{:=}1\,.\,Pos)\,.\,s{:=}0\,.\,Sum\ .$

$n'{=}n \wedge xs'{=}xs \wedge s'{=}s \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}s$

$\longrightarrow$ $\langle$non-recursive call$\rangle$

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2\ .$

(**scope** $xs \cdot$ **var** $c \cdot c{:=}1\,.\,Pos)\,.\,s{:=}0\ .$

if $n{=}0$ **then** ok

else case xs **of** $nil{\to}ok$

$\mid cons\ h\ r{\to}(s{:=}s{+}h\ .\ xs{:=}r\ .\ n{:=}n{-}1\ .\ Sum)\ .$

$n'{=}n \wedge xs'{=}xs \wedge s'{=}s \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}s$

Subexecution for conditional branch:

- $n'{=}1 \wedge xs'{=}nil \wedge s'{=}2\ .$

  (**scope** $xs \cdot$ **var** $c \cdot c{:=}1\,.\,Pos)\,.\,s{:=}0\ .$

  $n'{=}n \wedge xs'{=}xs \wedge s'{=}s \wedge un{=}n \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

  $\longrightarrow$ $\langle$memory assignment$\rangle$

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2\,.$

$(\textbf{scope}\, xs \cdot \textbf{var}\, c \cdot c{:=}1\,.\, Pos)\,.$

$n'{=}n \wedge xs'{=}xs \wedge s'{=}0 \wedge un{=}n \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

Subexecution for scope unused:

– $n'{=}1 \wedge xs'{=}nil \wedge s'{=}2\,.$

$n'{=}n \wedge xs'{=}xs \wedge s'{=}s \wedge un{=}n \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

$\longrightarrow$ ⟨binding merge⟩

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2 \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

Subexecution stops.

Continue:

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2\,.$

$(\textbf{scope}\, xs \cdot \textbf{var}\, c \cdot c{:=}1\,.\, Pos)\,.$

$n'{=}n \wedge xs'{=}xs \wedge s'{=}0 \wedge un{=}n \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

$\longrightarrow$ ⟨after subexecution for scope unused⟩

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2\,.$

$(\textbf{scope}\, xs \cdot \textbf{var}\, c \cdot c{:=}1\,.\, Pos)\,.$

$n'{=}1 \wedge xs'{=}xs \wedge s'{=}0 \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

Subexecution stops.

Continue (note $n'{=}1$ makes $n{=}0$ false, choose the **else** branch):

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2\,.$

$(\textbf{scope}\, xs \cdot \textbf{var}\, c \cdot c{:=}1\,.\, Pos)\,.\ s{:=}0\,.$

$\textbf{if}\ n{=}0\ \textbf{then}\ ok$

$\textbf{else case}\ xs\ \textbf{of}\quad nil{\rightarrow}ok$

$\quad\quad\quad\quad \mid cons\, h\, r{\rightarrow}(s{:=}\, s{+}h\,.\, xs{:=}r\,.\, n{:=}n{-}1\,.\, Sum)\,.$

$n'{=}n \land xs'{=}xs \land s'{=}s \land un{=}\boxdot \land uxs{=}\boxdot \land us{=}s$

$\longrightarrow$ ⟨after subexecution for conditional branch⟩

$n'{=}1 \land xs'{=}nil \land s'{=}2\,.$

$(\mathbf{scope}\, xs \cdot \mathbf{var}\, c \cdot c{:=}1\,.\, Pos)\,.$

$n'{=}1 \land xs'{=}xs \land s'{=}0\,.$

$\mathbf{case}\ xs\ \mathbf{of}\ \ nil{\to}ok$

$\quad\quad\mid cons\, h\, r \to (s{:=}s{+}h\,.\, xs{:=}r\,.\, n{:=}n{-}1\,.\, Sum)\,.$

$n'{=}n \land xs'{=}xs \land s'{=}s \land un{=}\boxdot \land uxs{=}\boxdot \land us{=}s$

Subexecution for case branch:

- (abbreviate $xstag{=}\mathbf{case}\ xs\ \mathbf{of}\ nil{\to}nil \mid cons\, h\, r \to cons\, \boxdot\, \boxdot$)

  $n'{=}1 \land xs'{=}nil \land s'{=}2\,.$

  $(\mathbf{scope}\, xs \cdot \mathbf{var}\, c \cdot c{:=}1\,.\, Pos)\,.$

  $n'{=}1 \land xs'{=}xs \land s'{=}0\,.$

  $n'{=}n \land xs'{=}xs \land s'{=}s \land un{=}\boxdot \land uxs{=}xstag \land us{=}\boxdot$

  $\longrightarrow$ ⟨binding merge⟩

  $n'{=}1 \land xs'{=}nil \land s'{=}2\,.$

  $(\mathbf{scope}\, xs \cdot \mathbf{var}\, c \cdot c{:=}1\,.\, Pos)\,.$

  $n'{=}1 \land xs'{=}xs \land s'{=}0 \land un{=}\boxdot \land uxs{=}xstag \land us{=}\boxdot$

  $\longrightarrow$ ⟨scope used⟩

  $n'{=}1 \land xs'{=}nil \land s'{=}2\,.$

  $(\mathbf{var}\, c \cdot c{:=}1\,.\, Pos)\,.$

  $n'{=}1 \land xs'{=}xs \land s'{=}0 \land un{=}\boxdot \land uxs{=}xstag \land us{=}\boxdot$

  $\longrightarrow$ ⟨local variable introduction⟩

  $n'{=}1 \land xs'{=}nil \land s'{=}2\,.$

  $c{:=}1\,.\, Pos\,.$

$c'{=}c \wedge n'{=}1 \wedge xs'{=}xs \wedge s'{=}0 \wedge uc{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}xstag \wedge us{=}\boxdot$

$\longrightarrow$ ⟨non-recursive call⟩

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c{:=}1$.

$(\textbf{var}\ c0 \cdot c0{:=}c\ .\ c{:=}c{+}1\ .\ (\textbf{scope}\ c, xs \cdot Pos)\ .\ xs{:=}cons\ c0\ xs)$.

$c'{=}c \wedge n'{=}1 \wedge xs'{=}xs \wedge s'{=}0 \wedge uc{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}xstag \wedge us{=}\boxdot$

$\longrightarrow$ ⟨local variable introduction⟩

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c{:=}1$.

$c0{:=}c\ .\ c{:=}c{+}1\ .\ (\textbf{scope}\ c, xs \cdot Pos)\ .\ xs{:=}cons\ c0\ xs$.

$c0'{=}c0 \wedge c'{=}c \wedge n'{=}1 \wedge xs'{=}xs \wedge s'{=}0 \wedge uc0{=}\boxdot \wedge uc{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}xstag \wedge us{=}\boxdot$

$\longrightarrow$ ⟨memory assignment⟩

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c{:=}1$.

$c0{:=}c\ .\ c{:=}c{+}1\ .\ (\textbf{scope}\ c, xs \cdot Pos)$.

$c0'{=}c0 \wedge c'{=}c \wedge n'{=}1 \wedge xs'{=}cons\ c0\ xs \wedge s'{=}0 \wedge uc0{=}\boxdot \wedge uc{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

Subexecution stops.

Continue:

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$(\textbf{scope}\ xs \cdot \textbf{var}\ c \cdot c{:=}1\ .\ Pos)$.

$n'{=}1 \wedge xs'{=}xs \wedge s'{=}0$.

$\textbf{case}\ xs\ \textbf{of}\quad nil{\to}ok$

$\qquad\qquad \mid\ cons\ h\ r{\to}(s{:=}s{+}h\ .\ xs{:=}r\ .\ n{:=}n{-}1\ .\ Sum)$.

$n'{=}n \wedge xs'{=}xs \wedge s'{=}s \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}s$

$\longrightarrow$ ⟨after subexecution for case branch:

the first 4 lines come from the subexecution,

extended with $h'{=}c0 \wedge r'{=}xs$ for the *cons* case;

the 5th line is the chosen branch of **case**,

the last 2 lines are the bindings after,

extended with $h'{=}h \wedge r'{=}r$ for the *cons* case$\rangle$

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c{:=}1$.

$c0{:=}c$. $c{:=}c{+}1$. (**scope** $c, xs \cdot Pos$) .

$h'{=}c0 \wedge r'{=}xs \wedge c0'{=}c0 \wedge c'{=}c \wedge n'{=}1 \wedge xs'{=}cons\, c0\, xs \wedge s'{=}0$.

$s{:=}s{+}h$. $xs{:=}r$. $n{:=}n{-}1$. Sum .

$h'{=}h \wedge r'{=}r \wedge n'{=}n \wedge xs'{=}xs \wedge s'{=}s$

$\wedge\ uh{=}\boxdot \wedge ur{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}s$

$\xrightarrow{1}$ $\langle$recursive call$\rangle$

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c{:=}1$.

$c0{:=}c$. $c{:=}c{+}1$. (**scope** $c, xs \cdot Pos$) .

$h'{=}c0 \wedge r'{=}xs \wedge c0'{=}c0 \wedge c'{=}c \wedge n'{=}1 \wedge xs'{=}cons\, c0\, xs \wedge s'{=}0$.

$s{:=}s{+}h$. $xs{:=}r$. $n{:=}n{-}1$.

if $n{=}0$ **then** ok

else case xs **of** $nil{\to}ok$

$\mid cons\, h\, r{\to}(s{:=}s{+}h\ .\ xs{:=}r\ .\ n{:=}n{-}1\ .\ Sum)$.

$h'{=}h \wedge r'{=}r \wedge n'{=}n \wedge xs'{=}xs \wedge s'{=}s$

$\wedge\ uh{=}\boxdot \wedge ur{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}s$

Subexecution for conditional branch (demands n):

- $n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c{:=}1$.

$c0{:=}c$. $c{:=}c{+}1$. ($\textbf{scope}\ c, xs \cdot Pos$) .

$h'{=}c0 \land r'{=}xs \land c0'{=}c0 \land c'{=}c \land n'{=}1 \land xs'{=}cons\ c0\ xs \land s'{=}0$.

$s{:=}s{+}h$. $xs{:=}r$. $n{:=}n{-}1$.

$h'{=}h \land r'{=}r \land c0'{=}c0 \land c'{=}c \land n'{=}n \land xs'{=}xs \land s'{=}s$

$\land\ uh{=}\boxdot \land ur{=}\boxdot \land uc0{=}\boxdot \land uc{=}\boxdot \land un{=}n \land uxs{=}\boxdot \land us{=}\boxdot$

$\longrightarrow$ ⟨memory assignment (3 times)⟩

$n'{=}1 \land xs'{=}nil \land s'{=}2$.

$c{:=}1$.

$c0{:=}c$. $c{:=}c{+}1$. ($\textbf{scope}\ c, xs \cdot Pos$) .

$h'{=}c0 \land r'{=}xs \land c0'{=}c0 \land c'{=}c \land n'{=}1 \land xs'{=}cons\ c0\ xs \land s'{=}0$.

$h'{=}h \land r'{=}r \land c0'{=}c0 \land c'{=}c \land n'{=}n{-}1 \land xs'{=}r \land s'{=}s{+}h$

$\land\ uh{=}\boxdot \land ur{=}\boxdot \land uc0{=}\boxdot \land uc{=}\boxdot \land un{=}n \land uxs{=}\boxdot \land us{=}\boxdot$

$\longrightarrow$ ⟨binding merge⟩

$n'{=}1 \land xs'{=}nil \land s'{=}2$.

$c{:=}1$.

$c0{:=}c$. $c{:=}c{+}1$. ($\textbf{scope}\ c, xs \cdot Pos$) .

$h'{=}c0 \land r'{=}r \land c0'{=}c0 \land c'{=}c \land n'{=}1{-}1 \land xs'{=}xs \land s'{=}0{+}c0$

$\land\ uh{=}\boxdot \land ur{=}\boxdot \land uc0{=}\boxdot \land uc{=}\boxdot \land un{=}\boxdot \land uxs{=}\boxdot \land us{=}\boxdot$

$\longrightarrow$ ⟨evaluate $1{-}1$⟩

$n'{=}1 \land xs'{=}nil \land s'{=}2$.

$c{:=}1$.

$c0{:=}c$. $c{:=}c{+}1$. ($\textbf{scope}\ c, xs \cdot Pos$) .

$h'{=}c0 \land r'{=}r \land c0'{=}c0 \land c'{=}c \land n'{=}0 \land xs'{=}xs \land s'{=}0{+}c0$

$\land\ uh{=}\boxdot \land ur{=}\boxdot \land uc0{=}\boxdot \land uc{=}\boxdot \land un{=}\boxdot \land uxs{=}\boxdot \land us{=}\boxdot$

Subexecution stops.

Continue (note that n'=0 makes n=0 true, choose the **then** branch):

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c{:=}1$.

$c0{:=}c$. $c{:=}c{+}1$. (**scope** $c, xs \cdot Pos$) .

$h'{=}c0 \wedge r'{=}xs \wedge c0'{=}c0 \wedge c'{=}c \wedge n'{=}1 \wedge xs'{=}cons\ c0\ xs \wedge s'{=}0$.

$s{:=}s{+}h$. $xs{:=}r$. $n{:=}n{-}1$.

if $n{=}0$ **then** ok

else case xs **of** $nil{\rightarrow}ok$

$\quad\quad\quad\quad\quad\quad \mid cons\ h\ r{\rightarrow}(s{:=}s{+}h\ .\ xs{:=}r\ .\ n{:=}n{-}1\ .\ Sum)$.

$h'{=}h \wedge r'{=}r \wedge n'{=}n \wedge xs'{=}xs \wedge s'{=}s$

$\wedge\ uh{=}\boxdot \wedge ur{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}s$

$\longrightarrow$ ⟨after subexecution for conditional branch⟩

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c{:=}1$.

$c0{:=}c$. $c{:=}c{+}1$. (**scope** $c, xs \cdot Pos$) .

$h'{=}c0 \wedge r'{=}r \wedge c0'{=}c0 \wedge c'{=}c \wedge n'{=}0 \wedge xs'{=}xs \wedge s'{=}0{+}c0$.

ok .

$h'{=}h \wedge r'{=}r \wedge n'{=}n \wedge xs'{=}xs \wedge s'{=}s$

$\wedge\ uh{=}\boxdot \wedge ur{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}s$

$\longrightarrow$ ⟨skip⟩

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c{:=}1$.

$c0{:=}c$. $c{:=}c{+}1$. (**scope** $c, xs \cdot Pos$) .

$h'{=}c0 \wedge r'{=}r \wedge c0'{=}c0 \wedge c'{=}c \wedge n'{=}0 \wedge xs'{=}xs \wedge s'{=}0{+}c0$.

$\quad h'{=}h \wedge r'{=}r \wedge n'{=}n \wedge xs'{=}xs \wedge s'{=}s$

$\wedge\ uh{=}\boxdot \wedge ur{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}s$

$\longrightarrow$ ⟨binding merge⟩

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c{:=}1$.

$c0{:=}c$. $c{:=}c{+}1$. ($\mathbf{scope}\, c, xs \cdot Pos$) .

$\quad h'{=}c0 \wedge r'{=}r \wedge c0'{=}c0 \wedge c'{=}c \wedge n'{=}0 \wedge xs'{=}xs \wedge s'{=}0{+}c0$

$\wedge\ uh{=}\boxdot \wedge ur{=}\boxdot \wedge uc0{=}c0 \wedge uc{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

$\longrightarrow$ ⟨garbage collection: h and r⟩

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c{:=}1$.

$c0{:=}c$. $c{:=}c{+}1$. ($\mathbf{scope}\, c, xs \cdot Pos$) .

$c0'{=}c0 \wedge c'{=}c \wedge n'{=}0 \wedge xs'{=}xs \wedge s'{=}0{+}c0 \wedge uc0{=}c0 \wedge uc{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

Subexecution for scope unused:

- $n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

  $c{:=}1$.

  $c0{:=}c$. $c{:=}c{+}1$.

  $c0'{=}c0 \wedge c'{=}c \wedge n'{=}n \wedge xs'{=}xs \wedge s'{=}s \wedge uc0{=}c0 \wedge uc{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

  $\longrightarrow$ ⟨memory assignment⟩

  $n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

  $c{:=}1$.

  $c0{:=}c$.

  $c0'{=}c0 \wedge c'{=}c{+}1 \wedge n'{=}n \wedge xs'{=}xs \wedge s'{=}s \wedge uc0{=}c0 \wedge uc{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

  $\longrightarrow$ ⟨memory assignment⟩

  $n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c{:=}1$.

$c0'{=}c \wedge c'{=}c{+}1 \wedge n'{=}n \wedge xs'{=}xs \wedge s'{=}s \wedge uc0{=}\boxdot \wedge uc{=}c \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

$\longrightarrow$ ⟨memory assignment⟩

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c0'{=}1 \wedge c'{=}1{+}1 \wedge n'{=}n \wedge xs'{=}xs \wedge s'{=}s \wedge uc0{=}\boxdot \wedge uc{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

$\longrightarrow$ ⟨evaluate 1+1⟩

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c0'{=}1 \wedge c'{=}2 \wedge n'{=}n \wedge xs'{=}xs \wedge s'{=}s \wedge uc0{=}\boxdot \wedge uc{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

Subexecution stops.

Continue:

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c{:=}1$.

$c0{:=}c$. $c{:=}c{+}1$. ($\mathbf{scope}\, c, xs \cdot Pos$) .

$c0'{=}c0 \wedge c'{=}c \wedge n'{=}0 \wedge xs'{=}xs \wedge s'{=}0{+}c0 \wedge uc0{=}c0 \wedge uc{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

$\longrightarrow$ ⟨after subexecution for scope unused⟩

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c0'{=}1 \wedge c'{=}2 \wedge n'{=}n \wedge xs'{=}xs \wedge s'{=}s$.

($\mathbf{scope}\, c, xs \cdot Pos$) .

$c0'{=}1 \wedge c'{=}2 \wedge n'{=}0 \wedge xs'{=}xs \wedge s'{=}0{+}1 \wedge uc0{=}\boxdot \wedge uc{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

$\longrightarrow$ ⟨evaluate 0+1⟩

$n'{=}1 \wedge xs'{=}nil \wedge s'{=}2$.

$c0'{=}1 \wedge c'{=}2 \wedge n'{=}n \wedge xs'{=}xs \wedge s'{=}s$.

($\mathbf{scope}\, c, xs \cdot Pos$) .

$c0'{=}1 \wedge c'{=}2 \wedge n'{=}0 \wedge xs'{=}xs \wedge s'{=}1 \wedge uc0{=}\boxdot \wedge uc{=}\boxdot \wedge un{=}\boxdot \wedge uxs{=}\boxdot \wedge us{=}\boxdot$

Execution stops now. The rightmost binding has the answer $s'{=}1$. The number of recursive calls used is 1, as predicated on page 79 by $t' = t{+}n{+}max\,0\,(n{-}1)$ with $n{=}1$.

5.3 The Prospect of Speculative Execution

The execution rules above are not the laziest possible. It is possible to be lazier in conditional branching: concurrently execute both branches, and if they give the same result, we can ignore the condition. More generally, if the results are possibly different but the same up to what is demanded, we can still ignore the condition. For example:

$$
\begin{aligned}
& \mathit{More}\,.\,(\textbf{if } b \textbf{ then } xs{:=}cons\,y\,nil \textbf{ else } xs{:=}cons\,0\,xs)\,.\,xs'{=}xs \wedge y'{=}y \\
\longrightarrow\quad & \mathit{More}\,.\,\textbf{if } b \textbf{ then } (xs{:=}cons\,y\,nil\,.\,xs'{=}xs \wedge y'{=}y) \textbf{ else } (xs{:=}cons\,0\,xs\,.\,xs'{=}xs \wedge y'{=}y) \\
\longrightarrow\quad & \mathit{More}\,.\,\textbf{if } b \textbf{ then } xs'{=}cons\,y\,nil \wedge y'{=}y \textbf{ else } xs'{=}cons\,0\,xs \wedge y'{=}y
\end{aligned}
$$

If the only demand is on the tag of *xs*, this can stop now, and without a subexecution to resolve *b*, since in both branches *xs* has the same tag *cons*. (If the demand is more than that, a subexecution to resolve *b* will start sooner or later.) Similar speculative executions are also possible for case branching.

Although not covered in this thesis, these speculative executions can be made precise by appropriate usage transformations and execution rules—possibly more complicated ones. There is also space for variations.

5.4 Soundness Theorem

We now state and prove the soundness theorem that lazy execution takes no more recursive time than promised by refinements according to the previous chapter. Like in the eager case, there are premises.

Theorem 5.1 Using *m* for memory variables, *um* for usage variables, and *t* for time variables, we suppose these premises on programs and refinements:

1. Refinements have usage and recursive time annotations as in the previous chapter.

2. Every non-compound program P (memory assignment statements, labels) used in the refinements satisfies

$$\forall m, t, um' \cdot \exists m', t', um \cdot P \wedge t' \geq t$$

   we call this *implementable* in this chapter.

3. The starting program *Main* satisfies

$$\forall m\colon M \cdot \forall um'\colon U \cdot \forall m', t, t', um \cdot um' \sqsubseteq m' \wedge Main \Rightarrow t' \leq t + f\, m\, um'$$

   given memory state subspace M, usage subspace U, and a *nat*-valued function f.

   For example *Main* may be the specification

$$x \neq nil \wedge ux' \neq \boxdot \;\Rightarrow\; x' = nil \wedge t' = t + ulen\, ux'$$

   Then M is the subspace for $x \neq nil$, and U is the subspace for $ux' \neq \boxdot$

Then we have

- for each memory pre-value $i\colon M$ and usage $d\colon U$ satisfying $(\forall m' \cdot (m{:=}i\,.\,Main) \Rightarrow d \sqsubseteq m')$, execution of $(m'{=}i\,.\,Main\,.\,m'{=}m \wedge u{=}d)$ stops in at most $f\; i\; d$ recursive call steps

□

As in the eager case, to prove this using induction, we will prove a stronger statement (generalize $f\, m\, um'$ to n in the consequent, specialize m to i and um' to d in the antecedent):

$$\forall n\colon nat \cdot \forall P, i, d \cdot \quad (\forall m', u, t, t' \cdot (m{:=}i\,.\,P\,.\,um{=:}d) \Rightarrow t' \leq t{+}n) \wedge (\forall m' \cdot (m{:=}i\,.\,P) \Rightarrow d \sqsubseteq m')$$
$$\Rightarrow\; (m'{=}i\,.\,P\,.\,m'{=}m \wedge u{=}d \text{ stops in } n \text{ recursive calls})$$

where P ranges over all programs satisfying the premises.

- Base case: Let annotated program P, initial value a, and demand d be given and assume

$$\forall m', u, t, t' \cdot (m{:=}a\,.\,P\,.\,um{=:}d) \Rightarrow t' \leq t{+}0$$

Let *sP* be *P* without annotation (usage or time). So the starting execution state is

$$m'{=}a\,.\,sP\,.\,m'{=}m \wedge um{=}d$$

If the recursive call rule is used (in the main execution or a subexecution), i.e., we hit an execution state

$$sMore1\,.\,Label\,.\,m'{=}b \wedge um{=}d1$$

and $d1$ has enough demands so the recursive call is executed, then, putting back usage and time for reasoning at the refinement level (let $More1$ be $sMore1$ with annotations):

$More1\,.\,Label\,.\,t{:=}t{+}\textbf{if }um'{\neq}\boxdot\textbf{ then }1\textbf{ else }0\,.\,m'{=}b \wedge um{=}d1 \wedge t'{=}t$

$=$ ⟨simplify rightmost composition; $d1$ has enough demands⟩

$More1\,.\,Label\,.\,m'{=}b \wedge um{=}d1 \wedge t'{=}t{+}1$

$=$ ⟨create assignment⟩

$More1\,.\,Label\,.\,m'{=}b \wedge um{=}d1 \wedge t'{=}t\,.\,t{:=}t{+}1$

We note that each execution step and subexecution takes a prefix subprogram of the parent execution state and simplifies, so ($More1\,.\,Label\,.\,m'{=}b \wedge um{=}d1 \wedge t'{=}t\,.\,t{:=}t{+}1$) is essentially a prefix subprogram of ($m{:=}a\,.\,P\,.\,um{=}{:}d$) with simplifications. Since the parent program refines $t'{\le}t$, the prefix subprogram does too. But this reaches a contradiction:

Let

$Q \;=\; More1\,.\,Label\,.\,m'{=}b \wedge um{=}d1 \wedge t'{=}t$

$Q12 \;=\;$ (subst m'',um'',t'' for m',um',t' in Q)

We prove:

$\neg\forall m, m', um, um', t, t' \cdot (Q\,.\,t{:=}t{+}1) \Rightarrow t'{\le}t$

$=$ ⟨expand assignment⟩

$\neg\forall m, m', um, um', t, t' \cdot (Q\,.\,m'{=}m \wedge um{=}um' \wedge t'{=}t{+}1) \Rightarrow t'{\le}t$

$=$ ⟨de Morgan and variations⟩

$\exists m, m', um, um', t, t' \cdot (Q\ .\ m'{=}m \wedge um{=}um' \wedge t'{=}t{+}1) \wedge t'{>}t$

$=$ ⟨expand sequential composition⟩

$\exists m, m', m'', um, um', um'', t, t', t'' \cdot Q12 \wedge m'{=}m'' \wedge um''{=}um' \wedge t'{=}t''{+}1 \wedge t'{>}t$

$=$ ⟨predicate calculus⟩

$\exists m, m'', um, um'', t, t'' \cdot Q12 \wedge t''{+}1{>}t$

$=$ ⟨arithmetic⟩

$\exists m, m'', um, um'', t, t'' \cdot Q12 \wedge t''{\ge}t$

$=$ ⟨rename⟩

$\exists m, m', um, um', t, t' \cdot Q \wedge t'{\ge}t$

$\Leftarrow$ ⟨generalize⟩

$\forall m, um', t \cdot \exists m', um, t' \cdot Q \wedge t'{\ge}t$

$=$ ⟨Q is implementable (premise 2)⟩

$\top$

- Induction step: Let annotated program P, initial value a, and demand d be given, and assume

  $$\forall m', um, t, t' \cdot (m{:=}a\,.\,P\,.\,um{=}{:}d) \Rightarrow t' \le t{+}n{+}1$$

  Let sP be P without annotation. So the starting execution state is

  $$m'{=}a\ .\ sP\ .\ m'{=}m \wedge um{=}d$$

  There are two cases. If execution does not use the recursive call rule, we are done. If the recursive call rule is used, then we proceed as follows.

  Take note of the first time the recursive call rule is used. We focus on the first instance because we will go back to P and inline the call there:

  Let iP be P modified by inlining non-recursive calls (replace labels by what they are refined by) and inlining the recursive call noted above, without the time increment: replace

$$\mathit{Label}\,.\; t{:=}t{+}(\mathbf{if}\ um'{\neq}\boxdot\ \mathbf{then}\ 1\ \mathbf{else}\ 0)$$

by *Body*, where the refinement used for the recursive call is $\mathit{Label}{\Leftarrow}\mathit{Body}$.

Let *siP* be *iP* without annotations. Then we note that

$$\forall m', um, t, t' \cdot (m{:=}a\,.\,iP\,.\,um{=:}d) \Rightarrow t' \le t{+}n$$

Then by induction, $m'{=}a\,.\,siP\,.\,m'{=}m \wedge um{=}d$ finishes execution using at most n recursive call steps. Therefore, $m'{=}a\,.\,sP\,.\,m'{=}m \wedge um{=}d$ follows the same steps plus one more recursive call step, and so it takes at most $n{+}1$ recursive call steps.

5.5 Related Work

Launchbury describes an operational semantics for lazy functional programs [22] at a similar level as ours. Its execution state looks like a special program fragment, and the memory store is represented by let-bindings. It uses big steps, while ours uses small steps; our choice of the small-step way is fairly arbitrary, although an upside is that some rules are more succint. The two operational semantics have the same essence, apart from the divide between functional and imperative, and between big-step and small-step. Our contribution is not in inventing a lazy operational semantics, but rather in writing it imperatively and linking it to refinements and their time predictions.

Sinot improves upon Launchbury's operational semantics to correspond better to code optimizations done by modern compilers [34]; those code optimizations concern higher-order functions, e.g., lambda-lifting. Since this thesis does not cover higher-order functions, it makes little difference whether one compares ours with Launchbury's older or Sinot's newer.

Although Wadler and Hughes [37, 36] and Sands [31, 32] have calculi for lazy timing based on context analysis, they are not proved sound with respect to a lazy operational semantics. In Sands's case, there is a partial proof: the lazy time calculus is bounded above by an eager time calculus, and the eager time is proved sound with respect to an eager operational semantics, provided that the program does not run into an error (such as the head of an empty list).

Lastly, this author has also written a lazy operational semantics [21] for a subset of Haskell. It uses small steps and expression graphs; the latter are equivalent to Launchbury's use of let-bindings but more succint and visually clearer. It is unrefereed and unofficial, as its main purpose is tutorial.

Chapter 6

A Space of Operational Semantics

In this chapter, we outline a possible space of operational semantics for the predicative programming theory in Chapter 2. It is small-step and high-level, i.e., rewrite rules over expressions that are close to programs. It contains eager execution (Chapter 3) and lazy execution (Chapter 5) as instances (except each has its own additions for book-keeping); it can also be a space for exploring speculative execution and other orders.

6.1 Execution State

An execution state is a program with *bindings* sequentially composed to some subprograms. Bindings store memory variables (can be initial, final, or intermediate). In the following example, bindings are $x'{=}3$, $x'{=}x{\times}2$, and $x'{=}x$.

$x'{=}3 \,.\, x{:=}x{+}1 \,.$

$\textbf{if } x{>}0 \textbf{ then } (x{:=}x{+}1 \,.\, x'{=}x{\times}2) \textbf{ else } (x'{=}x \,.\, x{:=}x{-}1)$

A binding stores state variables as a conjunction of equations, each equation taking the form *variable′=expression*.

6.2 Execution rules

Here are the execution rules. They can be applied to any matching subprogram in any order. Some rules use the bidirectional arrow $\longleftrightarrow$ to mean that they can be applied in either direction.

Let σ stand for the state variables, a, b, e stand for expressions.

- Binding creation:

$$\begin{array}{ll} & P \\ \longleftrightarrow & P\,.\,\sigma'{=}\sigma \end{array}$$

$$\begin{array}{ll} & P \\ \longleftrightarrow & \sigma'{=}\sigma\,.\,P \end{array}$$

- Binding merge:

$$\begin{array}{ll} & \sigma'{=}a\,.\,\sigma'{=}b \\ \longleftrightarrow & \sigma'{=}(\text{subst } a \text{ for } \sigma \text{ in } b) \end{array}$$

- Skip:

$$\begin{array}{ll} & ok\,.\,\sigma'{=}a \\ \longleftrightarrow & \sigma'{=}a \end{array}$$

$$\begin{array}{ll} & \sigma'{=}a\,.\,ok \\ \longleftrightarrow & \sigma'{=}a \end{array}$$

- Assignment:

$$\begin{array}{ll} & \sigma{:=}e\,.\,\sigma'{=}a \\ \longleftrightarrow & \sigma'{=}(\text{subst } e \text{ for } \sigma \text{ in } a) \end{array}$$

$$\begin{array}{ll} & \sigma'{=}a\,.\,\sigma{:=}e \\ \longleftrightarrow & \sigma'{=}(\text{subst } a \text{ for } \sigma \text{ in } e) \end{array}$$

- Call (recursive or non-recursive): given refinement *Label* $\Leftarrow$ *Body*

$$\begin{array}{ll} & \mathit{Label} \\ \longrightarrow & \mathit{Body} \end{array}$$

- Local variable introduction:

$$\begin{array}{ll} & (\mathbf{var}\, v \cdot P)\,.\,\sigma'{=}a \\ \longleftrightarrow & (\mathbf{var}\, v \cdot P\,.\, v'{=}v \wedge \sigma'{=}a) \end{array}$$

$$\begin{array}{ll} & \sigma'{=}a\,.\,(\mathbf{var}\, v \cdot P) \\ \longleftrightarrow & (\mathbf{var}\, v \cdot v'{=}v \wedge \sigma'{=}a\,.\, P) \end{array}$$

It may be necessary to rename v to a fresh name before using this rule to avoid name clashes.

- Osmosis:

$$\begin{array}{ll} & (\mathbf{var}\, v \cdot P)\,.\,\sigma{:=}e \\ \longleftrightarrow & (\mathbf{var}\, v \cdot P\,.\,\sigma{:=}e) \end{array}$$

$$\begin{array}{ll} & \sigma{:=}e\,.\,(\mathbf{var}\, v \cdot P) \\ \longleftrightarrow & (\mathbf{var}\, v \cdot \sigma{:=}e\,.\,P) \end{array}$$

It may be necessary to rename v to a fresh name before using this rule to avoid name clashes.

- Local variable elimination:

$$\begin{array}{ll} & (\mathbf{var}\, v \cdot v'{=}b \wedge \sigma'{=}a) \\ \longleftrightarrow & \sigma'{=}a \end{array}$$

It may be necessary to rename v to a fresh name before using this rule to avoid name clashes (especially when running this rule backwards).

- Scope:

$$\begin{array}{ll} & \sigma'{=}a \,.\, (\mathbf{scope}\, v \cdot P) \\ \longrightarrow & \sigma'{=}a \,.\, P \end{array}$$

$$\begin{array}{ll} & (\mathbf{scope}\, v \cdot P) \,.\, \sigma'{=}a \\ \longrightarrow & P \,.\, \sigma'{=}a \end{array}$$

- Conditional branch step 1:

$$\begin{array}{ll} & \sigma'{=}a \,.\, \mathbf{if}\ cond\ \mathbf{then}\ P\ \mathbf{else}\ Q \\ \longleftrightarrow & \mathbf{if}\ (\text{subst } a \text{ for } \sigma \text{ in } cond)\ \mathbf{then}\ (\sigma'{=}a \,.\, P)\ \mathbf{else}\ (\sigma'{=}a \,.\, Q) \end{array}$$

- Conditional branch step 2:

$$\begin{array}{ll} & \mathbf{if}\ \top\ \mathbf{then}\ P\ \mathbf{else}\ Q \\ \longrightarrow & P \end{array}$$

$$\begin{array}{ll} & \mathbf{if}\ \bot\ \mathbf{then}\ P\ \mathbf{else}\ Q \\ \longrightarrow & Q \end{array}$$

- Case branch: Suppose the binding $\sigma'{=}a$ contains $x'{=}tag\, e$.

$$\begin{array}{ll} & \sigma'{=}a \,.\, \mathbf{case}\ x\ \mathbf{of}\ tag\, w \rightarrow P \mid \ldots \\ \longleftrightarrow & (\mathbf{var}\, w \cdot w'{=}e \wedge \sigma'{=}a \,.\, P) \end{array}$$

Similarly for tags with other arities.

It may be necessary to rename w to a fresh name before using this rule to avoid name clashes.

- Branch distribution:

$$(\textbf{if}\ \mathit{cond}\ \textbf{then}\ P\ \textbf{else}\ Q)\,.\,R$$
$$\longleftrightarrow\ \textbf{if}\ \mathit{cond}\ \textbf{then}\ (P\,.\,R)\ \textbf{else}\ (Q\,.\,R)$$

$$(\textbf{case}\ x\ \textbf{of}\ \mathit{tag}\ w \to P \mid \ldots)\,.\,R$$
$$\longleftrightarrow\ \textbf{case}\ x\ \textbf{of}\ \mathit{tag}\ w \to (P\,.\,R) \mid \ldots$$

Expressions in bindings and branching conditions may be ready for evaluation after some assignments, binding merges, and conditional branching steps 1. We use these simple evaluations:

- Primitive operations: evaluate when all necessary operands are literals, e.g.,

$$1+1$$
$$\longleftrightarrow\ 2$$

- Conditional expression: evaluate when the condition is a literal:

$$\textbf{if}\ \top\ \textbf{then}\ e0\ \textbf{else}\ e1$$
$$\longleftrightarrow\ e0$$

$$\textbf{if}\ \bot\ \textbf{then}\ e0\ \textbf{else}\ e1$$
$$\longleftrightarrow\ e1$$

- Case analysis expression: evaluate when the argument has its tag exposed, e.g.,

$$\textbf{case}\ \mathit{tag}\ e0\ \textbf{of}\ \mathit{tag}\ w \to e1 \mid \ldots$$
$$\longleftrightarrow\ (\text{subst}\ e0\ \text{for}\ w\ \text{in}\ e1)$$

6.2.1 Execution and Refinement

Most rules are designed so that, if $P \longrightarrow Q$, then we have $(\forall \sigma, \sigma' \cdot P{\Leftarrow}Q)$. (And so if $P \longleftrightarrow Q$ then $(\forall \sigma, \sigma' \cdot P{=}Q)$.) An exception is when the scope rule is involved: execution strips **scope** for convenience, so from the execution

$$
\begin{array}{ll}
 & x'{=}0 \wedge v'{=}1 \,.\, (\textbf{scope}\, v \cdot v' \le v{+}3) \\
\longrightarrow & \qquad \langle \text{scope} \rangle \\
 & x'{=}0 \wedge v'{=}1 \,.\, v' \le v{+}3 \\
\longrightarrow & \qquad \langle \text{call, with refinement } v' \le v{+}3 \;\Leftarrow\; v{:=}v{+}1 \,.\, v' \le v{+}2 \rangle \\
 & x'{=}0 \wedge v'{=}1 \,.\, v{:=}v{+}1 \,.\, v' \le v{+}2
\end{array}
$$

we cannot deduce

$$
\begin{array}{ll}
\forall x, v, x', v' \cdot & (x'{=}0 \wedge v'{=}1 \,.\, (\textbf{scope}\, v \cdot v' \le v{+}3)) \\
 & \Leftarrow (x'{=}0 \wedge v'{=}1 \,.\, v{:=}v{+}1 \,.\, v' \le v{+}2)
\end{array}
$$

However, this violation is technical rather than essential. If we put back the **scope** construct at the refinement level, we can deduce

$$
\begin{array}{ll}
\forall x, v, x', v' \cdot & (x'{=}0 \wedge v'{=}1 \,.\, (\textbf{scope}\, v \cdot v' \le v{+}3)) \\
 & \Leftarrow (x'{=}0 \wedge v'{=}1 \,.\, (\textbf{scope}\, v \cdot (v{:=}v{+}1 \,.\, v' \le v{+}2)))
\end{array}
$$

6.3 Examples

Here are two short examples of uncommon execution orders. Though uncommon, they have some practical uses.

Example 6.1 The first example mixes lazy execution with speculative execution. At a branching statement, it is possible to carry out lazy execution in both branches (in parallel or interleaved) without waiting for the branching condition:

$R\,.$

if $n{=}0$

then $(P\,.\,xs{:=}cons\,3\,xs\,.\,xs{:=}cons\,1\,xs)$

else $(Q\,.\,xs{:=}cons\,2\,xs\,.\,xs{:=}cons\,1\,xs)\,.$

$xs'{=}xs \wedge n'{=}n$

$\longrightarrow$ ⟨branch distribution⟩

$R.$

if $n{=}0$

then $(P\,.\,xs{:=}cons\,3\,xs\,.\,xs{:=}cons\,1\,xs\,.\,xs'{=}xs \wedge n'{=}n)$

else $(Q\,.\,xs{:=}cons\,2\,xs\,.\,xs{:=}cons\,1\,xs\,.\,xs'{=}xs \wedge n'{=}n)$

$\longrightarrow$ ⟨assignment⟩

$R.$

if $n{=}0$

then $(P\,.\,xs{:=}cons\,3\,xs\,.\,xs'{=}cons\,1\,xs \wedge n'{=}n)$

else $(Q\,.\,xs{:=}cons\,2\,xs\,.\,xs'{=}cons\,1\,xs \wedge n'{=}n)$

$\longrightarrow$ ⟨branch distribution in reverse⟩

$R\,.$

if $n{=}0$

then $(P\,.\,xs{:=}cons\,3\,xs)$

else $(Q\,.\,xs{:=}cons\,2\,xs)\,.$

$xs'{=}cons\,1\,xs \wedge n'{=}n$

If this suffices for a certain demand (for example, only the first item of xs' is required), then execution stops here, without executing R to resolve the condition $n{=}0$. (It is also possible to execute R in parallel with the above.)

□

Example 6.2 The second example mixes speculative execution with eager execution. At a branching statement, both branches are executed eagerly without waiting for the branching condition:

$n'{=}2 \wedge y'{=}1\ .\ \textbf{if}\ p\,y\ \textbf{then}\ (n{:=}n{\times}n\ .\ P)\ \textbf{else}\ (n{:=}n{+}n\ .\ Q)$

$\longrightarrow \qquad \langle\text{conditional branch step 1}\rangle$

$\textbf{if}\ p\ 1\ \textbf{then}\ (n'{=}2 \wedge y'{=}1\ .\ n{:=}n{\times}n\ .\ P)\ \textbf{else}\ (n'{=}2 \wedge y'{=}1\ .\ n{:=}n{+}n\ .\ Q)$

$\longrightarrow \qquad \langle\text{assignment}\rangle$

$\textbf{if}\ p\ 1\ \textbf{then}\ (n'{=}4 \wedge y'{=}1\ .\ P)\ \textbf{else}\ (n'{=}4 \wedge y'{=}1\ .\ Q)$

This order may achieve good performance if there are spare processors to execute both branches concurrently, p takes a long time to compute or y takes a long time to load, and n is already in cache/register.

□

The decision to use these and other speculative execution orders could be made at compile time based on aggressive analysis, at run time based on monitoring, or a mixture of both. How to reason about their timing in refinements is future work.

6.4 Parallel Execution

Our execution rules allow some kind of parallel execution of sequential programs, since bindings can occur at multiple points in the program, and execution rules can be applied to those points simultaneously. For example:

$x'{=}2 \wedge y'{=}2\ .\ x{:=}x{+}1\ .\ P\ .\ y{:=}1\ .\ Q$

$\longrightarrow \qquad \langle\text{binding creation; add parentheses for emphasis}\rangle$

$(x'{=}2 \wedge y'{=}2\ .\ x{:=}x{+}1\ .\ P).\ (x'{=}x \wedge y'{=}y\ .\ y{:=}1\ .\ Q)$

$\longrightarrow \qquad \langle\text{assignment at both bindings}\rangle$

$(x'{=}3 \wedge y'{=}2\ .\ P).\ (x'{=}x \wedge y'{=}1\ .\ Q)$

For more parallelism, it is also possible to extend the programming language by a parallel operator. Following Hehner [15], we introduce independent composition $P||Q$ (same operator precedence as sequential composition) with this meaning: partition the memory variables into two disjoint sets, say x and y; then P owns x (can read and write) and sees y as a constant, and Q owns y (can read and write) and sees x as a constant. This ensures freedom from conflicts, and so at the refinement level, ignoring time and space, we have

$$P\|Q \;=\; P\wedge Q$$

At the execution level, we now add these rules:

- Fork:

$$\begin{aligned} & x'=a\wedge y'=b\,.\,(P\|Q) \\ \longleftrightarrow\quad & (x'=a\wedge y'=b\,.\,P)\|(x'=a\wedge y'=b\,.\,Q) \end{aligned}$$

- Join:

$$\begin{aligned} & x'=a1\wedge y'=b \,\|\, x'=a\wedge y'=b1 \\ \longleftrightarrow\quad & x'=a1\wedge y'=b1 \end{aligned}$$

  (The left operand owns x, and so we only care about its $x'=a1$. The right operand owns y, and so we only care about its $y'=b1$.)

Example 6.3 Using parallel assignment to swap two variables:

$$\begin{aligned} & x'=0\wedge y'=1\,.\,(x{:=}y\|y{:=}x) \\ \longrightarrow\quad & \quad\langle\text{fork}\rangle \\ & (x'=0\wedge y'=1\,.\,x{:=}y)\|(x'=0\wedge y'=1\,.\,y{:=}x) \\ \longrightarrow\quad & \quad\langle\text{assignment}\rangle \\ & x'=1\wedge y'=1 \,\|\, x'=0\wedge y'=0 \\ \longrightarrow\quad & \quad\langle\text{join}\rangle \\ & x'=1\wedge y'=0 \end{aligned}$$

□

The fork rule favours eager execution. Adding independent composition to lazy execution is future work.

6.5 Related Work

As mentioned in Chapters 3 and 5, there are other operational semantics for eager predicative programming [18] and lazy functional programming [22, 34]. As for a whole space of operational semantics, although the lambda calculus has enjoyed one since its conception, ours is new for predicative programming.

Chapter 7

Conclusion

7.1 Summary of Contributions

Using a predicative programming theory without built-in termination as the substrate, we have contributed an eager operational semantics, a lazy operational semantics, and a space for more operational semantics. These operational semantics are high-level and small-step, i.e., execution states are close to program expressions, and execution goes through transition rules. The transition rules have a close relation with refinements.

Using the eager operational semantics, we have proved soundness of refinements that assume eager timing: If refinements promise a recursive time bound, then execution stops in at most that many recursive calls.

On the laziness front, we have contributed a method of stating and proving refinements that assumes lazy timing. It consists of usage variables and values to express how much of the final answer is demanded, usage transformations to propagate them (backwards) to the middle of programs, and demand-dependent recursive time. The method is compositional with respect to program structure (usage variables serve as the interface) and mathematically simpler than previous compositional methods. We have proved soundness of this method using the lazy operational semantics, while previous compositional methods do not have similar soundness proofs.

7.2 Summary of Related Work

We have discussed related work and comparisons in most chapters. Here we recapitulate some noteworthy ones and add a few on automatic finding of time bounds.

Our eager operational semantics is close to that of Hoare and He in Unifying Theories of Programming [18]. Their execution state is a tuple of memory variable bindings and a program; ours represents the bindings as a special specification (e.g., $x'{=}2 \wedge y'{=}0$ to bind x to 2 and y to 0), and so we can replace the tuple by a sequential composition. This allows us to relate execution with refinement more smoothly. It is also easier to modify for laziness and other execution strategies, where it is convenient to insert bindings into arbitrary points in the program.

Our usage variables, usage values, and usage transformations for lazy refinements are vastly simpler than context analysis used by Wadler and Hughes [37, 36] and by Sands [31, 32]. Whereas we propagate usage values backwards, they propagate contexts (functions from usage values to usage values) backwards. Whereas our usage transformations map usage values to usage values, they need context transformations—functions from contexts to contexts, which is second-order. On top of that, they need an extra element in their partial orders to stand for errors in programs (such as the head of the empty list), whereas we have a full-fledged specification language and we can use preconditions to guard against such errors. Our work's shortcoming compared to Sands's is that ours does not cover higher-order functions; however, we have a soundness proof.

Our lazy operational semantics is new for imperative and predicative programming, but it has the same essence as those for lazy functional programming by Launchbury [22] and by Sinot [34].

Both the eager timing scheme of Hehner's theory [14, 15] and our lazy time scheme are for verification rather than synthesis: they give the proof obligations for given time bounds, but they do not give the time bounds. Automatic computing of time bounds and other bounds for common, practical cases is covered by other work, and here we just cite a few recent examples: Albert et al. [1], Hoffmann et al. [19], and Zuleger et al. [40].

7.3 Future Work

To most programmers who use lazy programming languages, the two difficulties and mysteries are time costs and space costs. We have given a compositional solution to that of time costs; it remains to give a compositional solution to that of space costs. Existing work consists of global, non-compositional approaches using a space operational semantics directly [4], and compositional methods that do not calculate space costs, but only prove that certain program replacements do not increase space costs [8].

Our method for lazy timing covers only first-order programming. There is future prospect in adding higher-order functions and procedures. As Hughes points out, better modularization of programs is enabled by both laziness and higher-order functions [20], and we have only covered laziness. It may be possible to improve upon existing work by Sands [31, 32] and by Guttmann [10, 9].

Our method for lazy timing supports only memory variables and sequential programs. Extending it for I/O, communication, and concurrency is future work.

We have touched upon more possibilities of execution strategies by outlining a space of operational semantics. Some partly lazy strategies and some partly speculative strategies are appearing in practical software and hardware, and it will be useful to find formal methods to calculate their time and space costs.

Appendix A

Notation and Precedence

In decreasing order of precedence:

- $\top$, $\bot$ (boolean values “true” and “false” respectively)

  $\boxdot$ (domain-theoretic bottom, Section 4.1)

  literals

  parenthesized expressions

- function application written as juxtaposition, e.g., $f\ x$

- $\times$, $/$ (arithmetic)

- infix $+$, $-$ (arithmetic)

- $\sqcup$ (domain-theoretic least upper bound, Section 4.2.1)

- $\triangleright$ (Section 4.2.2)

- $=$, $\neq$, $<$, $>$, $\leq$, $\geq$ (equality and inequalities)

  $\sqsubseteq$, $\sqsubset$ (domain-theoretic order, Section 4.1)

  (all continuing, e.g., $a{=}b{\leq}c$ means $a{=}b$ and $b{\leq}c$)

- $\neg$ (boolean “not”)

- $\wedge$ (boolean “and”)

- $\lor$ (boolean “or”)

- $\Rightarrow$, $\Leftarrow$ (boolean implication, continuing)

- :=, =: (assignment, Section 2.1.2)

- **if then else** (conditional)

  case of (Section 2.1.4)

- . (sequential composition, Section 2.1.2)

  || (parallel composition, Section 6.4

- $\forall$, $\exists$ (predicate logic quantifiers)

  var, **scope** (Section 2.1.2)

- $\boldsymbol{=}$, $\boldsymbol{\Rightarrow}$, $\boldsymbol{\Leftarrow}$ (same meaning as =, $\Rightarrow$, and $\Leftarrow$ respectively, and continuing, but lowest precedence)

  $\longrightarrow$, $\xrightarrow{n}$ (execution, continuing, Sections 3.1, 5.1, 6.2)

Bibliography

[1] Elvira Albert, Samir Genaim, and Abu Naser Masud. More precise yet widely applicable cost analysis. In *VMCAI 2011*, 2011.

[2] Ralph-Johan Back, Anna Mikhajlova, and Joakim von Wright. Class refinement as semantics of correct object substitutability. *Formal Aspects of Computing*, 12:18–40, 2000. Also as TUCS Technical Report 333.

[3] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

[4] Adam Bakewell and Colin Runciman. A space semantics for core Haskell. In *2000 ACM SIGPLAN Haskell Workshop*, September 2000.

[5] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[6] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer, 1990.

[7] Rutger M. Dijkstra. Relational calculus and relational program semantics. Computer Science Report CS-R9408, Department of Computing Science, University of Groningen, The Nethelands, 1994.

[8] Jörgen Gustavsson and David Sands. Possibilities and limitations of call-by-need space improvement. In *Proceeding of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 265–276. ACM Press, September 2001.

[9] Walter Guttmann. Imperative abstractions for functional actions. *The Journal of Logic and Algebraic Programming*, 79:768–793, 2010.

[10] Walter Guttmann. Lazy UTP. In *Second International Symposium on Unifying Theories of Programming*, volume 5713 of *Lecture Notes in Computer Science*, pages 82–101. Springer-Verlag, 2010.

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

[12] Eric C. R. Hehner. Termination is timing. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 36–47, Groningen, The Netherlands, June 1989. Springer.

[13] Eric C. R. Hehner. A practical theory of programming. *Science of Computer Programming*, 14(2,3):133–158, October 1990.

[14] Eric C. R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer, 1993.

[15] Eric C. R. Hehner. *A Practical Theory of Programming*. `http://www.cs.toronto.edu/~hehner/aPToP/`, 2002–2011.

[16] Eric C. R. Hehner, Lorene E. Gupta, and Andrew J. Malton. Predicative methodology. *Acta Informatica*, 23(5):487–505, September 1986.

[17] Eric C. R. Hehner and Andrew J. Malton. Termination conventions and comparative semantics. *Acta Informatica*, 25(1):1–14, January 1988.

[18] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall International Series in Computer Science. Prentice Hall, 1998.

[19] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *POPL 2011*, 2011.

[20] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[21] Albert Y. C. Lai. Lazy evaluation of Haskell. `http://www.vex.net/~trebla/haskell/lazy.xhtml`, 2011.

[22] John Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, 1993.

[23] M. Douglas McIlroy. Enumerating the strings of regular languages. *Journal of Functional Programming*, 14(5):503–518, 2004.

[24] Ali Mili, Jules Desharnais, and Fatma Mili. *Computer Program Construction*. Oxford University Press, 1994.

[25] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[26] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[27] Thomas Nordin and Andrew Tolmach. Modular lazy search for constraint satisfaction problems. *Journal of Functional Programming*, 11(5):557–587, 2001.

[28] David Lorge Parnas. A generalized control structure and its formal definition. *Communications of the ACM*, 26(8):572–581, August 1983.

[29] Peter Naur (editor) et al. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1–17, January 1963.

[30] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–129, 1975.

[31] David Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, Department of Computing, Imperial College, University of London, September 1990.

[32] David Sands. Complexity analysis for a lazy higher-order language. In *Proceedings of the Third European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 361–376. Springer-Verlag, 1990.

[33] D. S. Scott. Continuous lattices, toposes, algebraic geometry and logic. In *Proc. 1971 Dalhousie Conference*, volume 274 of *Lecture Notes in Mathematics*, pages 97–136. Springer-Verlag, 1972.

[34] François-Régis Sinot. Complete laziness: a natural semantics. *Electronic Notes in Theoretical Computer Science*, 204:129–145, April 2008.

[35] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–783, November 1982.

[36] Philip Wadler. Strictness analysis aids time analysis. In *15'th ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[37] Philip Wadler and R. J. M. Hughes. Projections for strictness analysis. In *3'rd International Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, USA, September 1987.

[38] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

[39] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, 1996.

[40] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS 2011*, 2011.