Assignment Calculus: A Pure Imperative Language

Marc Bender and Jeffery Zucker*

Department of Computing and Software, McMaster University,
Hamilton, Ont. L8S 4L8, Canada +1 905 525-9140
bendermm@mcmaster.ca, zucker@mcmaster.ca

Abstract. We undertake a study of *imperative computation*. Beginning with a philosophical analysis of the distinction between imperative and functional language features, we define a (pure) *imperative language* as one whose constructs are (inherently) *referentially opaque*. We then give a definition of a *computation language* by identifying desirable properties for such a language.
We present a new pure imperative computation language, *Assignment Calculus* ***AC***. The main idea behind ***AC*** is based on the insight of T. Janssen that Montague's modal operators of *intension* and *extension*, developed for the study of natural language semantics, are also useful for the semantics of programming language features such as assignments and pointers. ***AC*** consists of only four basic constructs, *assignment* '$\boldsymbol{X}$ **:= t**', *sequence* '**t ; u**', *procedure formation* '**ᵢt**' and *procedure invocation* '**!t**'. Two interpretations are given for ***AC***: an *operational semantics* and a *term-rewriting* system; these interpretations turn out to be equivalent.

1 Introduction

What is a pure imperative language? This paper attempts to answer this question by pursuing one possible definition: a pure imperative language is one whose operators are fundamentally *referentially opaque*.

In Section 2 we give a discussion of this approach, involving *referential transparency* and *opacity*, the *substitutivity* principle and *intensionality*, with natural-language examples. We show that these problems are also present in imperative programming languages, and introduce our imperative computation language Assignment Calculus, ***AC***.

Section 3 presents the syntax and operational semantics of ***AC***, with a running example which will be used throughout the paper. We examine an important aspect of ***AC***: *state backtracking*.

In Section 4 a *term rewriting system* for ***AC*** is presented, and used with our running example. We state the equivalence of the operational semantics and rewriting system, and (without proof) a confluence theorem for this system. We

* Research for this paper was supported by the Natural Sciences and Engineering Research Council of Canada and by the McMaster Center for Software Certification.

conclude in Section 5 with a summary of the significance of the ideas presented here, a brief examination of similar work, and possible lines of research.

A comment on notation: syntax is represented by bold text; the equivalence of two syntactic entities is indicated by '$\equiv$'.

This paper is based on the thesis [Ben10], which contains full proofs of most of the results stated here. The thesis also provides, among other things, a full denotational semantics for ***AC***, along with related results including the equivalence of operational, denotational, and rewriting semantics. This paper extends our work by presenting a stronger syntax-directed proof of equivalence between the operational semantics and term rewriting system.

2 Imperative Computation

To program a computer, we must provide it with instructions to arrive at the desired result. There are two types of instructions: *commands* (or statements) and *goals*. The latter provide a "specification," and the computer (or compiler) must figure out how to arrive at a solution. Goals are generally written as *declarations*, often presented in a "functional" style [Bac78].

This distinction between types of instructions forms the basis of the two types of programming language: a language that is based on commands is called *imperative*, and one that is based on goals is called *declarative*.

This distinction can be traced back to two pioneers in computation theory, Alan Turing and Alonzo Church. Turing's analysis of computation [Tur36] is via a machine that executes tasks sequentially, reading from and writing to a storage device (the "tape"). Church's system [Chu41] is more abstract, taking the mathematical notion of *function* as basic, and employing only the operations of (functional) *abstraction* and *application* to express computational goals. Turing proved that their systems have the same computational power, and argued convincingly that they are *universal* (the *Church-Turing Thesis*).

The benefit of Turing's approach is its suggestion of a real computer; a variant, due to von Neumann, lies at the core of virtually every computing device in use today. Since computers are basically built on Turing's idea, so are imperative languages. Thus any imperative language contains operations for sequencing ('**;**') and reading from and writing to memory (assignments). Imperative languages are thus closely connected to practice. However, dealing with imperative languages' semantics can be tricky, and the Turing machine can be a clumsy theoretical tool.

Church's λ-calculus stems from research into the foundations of mathematics. Its language of functional abstraction and application is small, elegant, and powerful, and makes it immediately amenable to theoretical study [Bar84]. Many "functional languages" have been designed around the λ-calculus. However, a major drawback to functional languages is their lack of "machine intuition", which can make them difficult to implement.

Real computers are based on the imperative model, so compilers for functional languages are needed to translate Church-style computation into Tur-

ing's model. Conversely, for those working in denotational semantics, giving a mathematical meaning to imperative programming languages means interpreting Turing-style computation in a Church-style language.

Can we "short-circuit" these interpretations in either direction? In other words, can we either (a) build a computer on Church's notion, or (b) design a formal language that embodies Turing's conception? In this paper we focus on question (b). In this regard, it is somewhat surprising that there is no accepted canonical *theoretical computation language* that is fundamentally imperative in character. Our goal is to present such a language. It is not presented as "the" basic language for imperative computation, but simply as a potential candidate. First, however, we must answer a pressing question: what, exactly, is "imperative computation"?

2.1 What is Imperative Computation?

A first attempt to define imperative computation could be made from the point of view of *machine behaviour*. If functional features are seen as "high-level," then we should remain close to a machine-based intuition.

The problem is that this does not sufficiently *restrict* our definition: there are many different machine architectures, instruction sets, abstractions from machine to assembly languages, and implementations of control structures; in short, there is too much freedom when working with machine intuition alone.

So we want an imperative language which is (a) as simple as possible, (b) Turing complete, and (c) "pure". How do we understand (c)? A good approach is to take the concepts of *pure imperative* and *pure functional* as "orthogonal" in some sense.

We begin by asking: what is *functional purity*?

Purely functional languages are *referentially transparent* [FH88]. This allows for *call-by-name* or *lazy* evaluation, as in Haskell [HHJW07]. Referential transparency as a concept goes back to Quine [Qui60, §30] and essentially embodies Leibniz's *principle of substitutivity of equals*:

$$\boldsymbol{e}_1 = \boldsymbol{e}_2 \implies \boldsymbol{e}(\cdots \boldsymbol{e}_1 \cdots) = \boldsymbol{e}(\cdots \boldsymbol{e}_2 \cdots).$$

The benefit of this is clear: "computation" proceeds by substituting expressions for variables. This idea is taken to an extreme in the pure λ-calculus, which reduces functional computation to its smallest possible core. Its (only!) operators are λ-*abstraction* and *application*; its only atoms are *variables*. Syntactically, application of a function to an argument is accomplished by substitution, taking advantage of referential transparency.

In fact we take the λ-calculus as the paragon of theoretical computation languages: (a) it is small, elegant and intuitive; (b) its operators represent well-understood, fundamental concepts; (c) we can rewrite its terms using simple rules; and (d) it has multiple equivalent forms of semantics. Our aim is to develop a formal language for *imperative* computation that has as many of the above virtues as possible.

Going back to our question: what is a pure imperative language? We now propose: it is a language whose features are fundamentally non-transparent, or *opaque*, i.e., substitutivity is the exception rather than the rule.

2.2 Referential Opacity

We begin with an example in natural language. Consider the sentence:

The temperature is twenty degrees and rising.

Formalizing this gives us

$$\boldsymbol{twenty}(\boldsymbol{temp}) \wedge \boldsymbol{rising}(\boldsymbol{temp}) \tag{1}$$

Suppose the temperature is $\mathbf{20°}$. Substituting this for $\boldsymbol{temp}$ in (1) (using substitutivity of equals) yields '$\boldsymbol{twenty}(\mathbf{20°})$' for the first conjunct, which is fine, but '$\boldsymbol{rising}(\mathbf{20°})$' for the second, which is not even false, but nonsense: "temperature" here means not its current value, but its value over time.

The problem is that although the predicate '$\boldsymbol{twenty}$' introduces a *transparent context*, the predicate '$\boldsymbol{rising}$' creates an *opaque context*.

Such intensional phenomena abound in natural language, and have been studied by philosophers and linguists for some time [Fre92]. They can be recognized by apparent violations of substitutivity as above. This is also the case for imperative programming languages, to which we now turn.

Consider the assignment statement $\boldsymbol{X} := \boldsymbol{X} + \mathbf{1}$. Clearly, we can substitute the current value of $\boldsymbol{X}$ for '$\boldsymbol{X}$' on the right-hand side, but not on the left-hand side; attempting the latter gives a meaningless expression. Janssen [JvEB77,Jan86] noticed that these are simple examples of *transparent* and *opaque* contexts in programming languages; and he was able to develop a compositional semantics for programming languages, dealing with assignments and pointers, comparable to what Montague had done for natural languages. This penetrating insight of Janssen's was the starting point for a line of investigation continued in [Hun90,HZ91], and further in [Ben10] and the present paper.

In fact, it turns out that opaque contexts are inherent in *all* the fundamental imperative operations.

2.3 Intensions

Frege [Fre92] analyzed the substitutivity problem. He distinguished between two kinds of meaning: *sense* (*Sinn*) and *denotation* (*Bedeutung*). He showed that, in cases where substitutivity does not hold in terms of the *denotations* of expressions, it can be *restored* if we consider, not the *denotation* of the expression, but its *sense*, which can be understood as a function from "contexts," "states," "settings," or (as in example (1)) "time instants" to *values*.

A formal system implementing Frege's ideas was developed by Church [Chu51], and a semantic interpretation by Carnap [Car47], who introduced the terms *intension* and *extension* for sense and denotation. Kripke [Kri59] rounded out the

semantic treatment by providing the setting of *possible worlds* for modal logic. By combining his work with Carnap's, we can treat intension as a function from possible worlds to values.

The next major step was accomplished by Montague [Mon70,Mon73], who developed a system ***IL*** of *intensional logic* for the mathematical treatment of natural language. Next, Janssen and van Emde Boas [JvEB77,Jan86] applied Montague's techniques to imperative programming languages. By identifying possible worlds with *machine states*, they provide a strikingly elegant treatment of assignment statements and pointers.

A significant extension was accomplished by Hung and Zucker [Hun90,HZ91], who provided compositional denotational semantics for quite intricate language features such as *blocks* with local identifiers; *procedure parameters* passed by *value*, by *reference* and by *name*; and *pointers* which can be dereferenced anywhere, including the left-hand side of assignment statements. Janssen's system — specifically, Hung's version — is the genesis of Assignment Calculus ***AC***, the language to be presented in this paper.

2.4 Intentions

We begin with the observation that

the intension operator generalizes *the (parameterless) procedure.* (2)

A procedure is a function from states to *states*, and an intension is a function from states to *values*. If we include states in the set of values, then the generalization is clear. The reason that it was not noticed by Janssen or Hung is that, in Montague's systems, *states cannot be treated as values.*

Another observation is that we can allow "storage" of intensions in the state. Stored procedures are a well-known but difficult feature of imperative languages [Sco70,SS71]. They allow general recursion, and also a generalization of the treatment of pointers given by Janssen and Hung.

The resulting system is sufficiently interesting to be studied in "isolation", removing all the functional and logical components in ***DIL***, such as variables, abstraction and application. The resulting language ***AC*** is a *case study in pure imperative computation.*

3 Assignment Calculus *AC*

In attempting to create a pure imperative computation language, it is important to remain as close as possible to existing imperative languages.

An imperative programming language can perhaps be defined as a language ***L*** with the following characteristics:

1. The interpretation of ***L*** depends on some form of computer memory (state) through which data can be stored and retrieved.

2. The state consists of contents of discrete memory locations that can be read from or written to independently.
3. An $\boldsymbol{L}$-program consists of a *sequence* of explicit *commands* or instructions that fundamentally depend on (and often change) the state.
4. $\boldsymbol{L}$ contains some form of looping, branching or recursion mechanism to allow for repeated execution of program parts.

The first three characteristics are common to all imperative languages. Therefore ***AC*** includes them directly.

Characteristic 4 can be embodied in many ways: conditional branching, "goto" statements; looping constructs ("while" etc.); and recursion.

AC's approach to 4 is to employ Montague's intension operator as a *generalization of parameterless procedure*, in accordance with our observation above. In ***AC***, intensions are treated as *first-class values*: they can be defined anonymously, assigned to locations, and invoked freely.

The goals of ***AC*** are twofold: first to continue the line of research initiated by Janssen and continued by Hung and Zucker in applying Montague's work to programming languages, and secondly to attempt to provide a small, elegant, useful *core language* for imperative-style computation—a *pure imperative computation language* as defined in Section 2.

3.1 Introduction to *AC*

Term is the set of ***AC*** terms $\mathbf{t}$, $\mathbf{u}$, ... Before defining ***Term*** formally, we go over the basic constructs of ***AC*** and their intuitive meanings.

1. *Locations*: $\boldsymbol{X}$, $\boldsymbol{Y}$, ... correspond to *memory locations*. The collection of all locations is the *store*.
2. *Assignment*: $\boldsymbol{X} := \mathbf{t}$. Overwrites the contents of location $\boldsymbol{X}$ with whatever $\mathbf{t}$ computes. This operation computes the *store* that results from such an update.
3. *Sequence*: $\mathbf{t}\,;\mathbf{u}$. Interpreted as follows: first compute $\mathbf{t}$, which returns a store, then compute $\mathbf{u}$ in this new context.
4. *Intension*: $\mathbf{i}\mathbf{t}$. This is *procedure formation*. It "stops" evaluation of $\mathbf{t}$ so that $\mathbf{i}\mathbf{t}$ is the procedure that, when invoked, returns whatever $\mathbf{t}$ computes.
5. *Extension*: $\mathbf{!t}$. This is *procedure invocation*, that is, it "runs" the procedure computed by $\mathbf{t}$.

An easy way to remember our notation (different from Montague's) for intension and extension is that '$\mathbf{i}$' looks like a lowercase 'i', for intension, and that '$\mathbf{!}$', an *ex*clamation mark, is used for extension.

This constitutes *pure* ***AC***. For convenience, we can add *supplementary operators*: *numerals* $\mathbf{n}, \ldots$, *booleans* $\mathbf{b}, \ldots$ and their standard operations.

We will use the following as a running example of an ***AC*** term:

$$\boldsymbol{P} := \mathbf{i}\boldsymbol{X};\ \boldsymbol{X} := \mathbf{1};\ \mathbf{!}\boldsymbol{P}. \tag{3}$$

This term sets $\boldsymbol{P}$ to the procedure that returns $\boldsymbol{X}$, then sets $\boldsymbol{X}$ to $\mathbf{1}$. Finally, $\boldsymbol{P}$ is invoked thus returning the current value of $\boldsymbol{X}$ which is $\mathbf{1}$.

3.2 Types and Syntax of *AC*

AC is a simply typed language in the sense of Church [Chu40].

Definition 3.1 (Types). *The set* ***Type*** *of types* $\tau, \ldots$*, is generated by:*

$$\tau \ ::= \ \mathsf{B} \mid \mathsf{N} \mid \mathsf{S} \mid \mathsf{S} \to \tau,$$

where B, N *and* S *are the types of booleans, naturals and stores respectively, and* $\mathsf{S} \to \tau$ *is that of* intensions *(procedures) which return values of type* τ*. (The base types* B *and* N *are added only to handle the supplementary operators described above; for pure* ***AC****, they can be removed.)*

Now we define the set of terms ***Term*** of ***AC***. For each type τ, the sets $\boldsymbol{Term}^{\tau}$ of terms of type τ are defined by mutual recursion:

Definition 3.2 (Syntax of *AC*).

1. $X \in \boldsymbol{Loc}^{\tau} \Rightarrow X \in \boldsymbol{Term}^{\tau}$
2. $\mathsf{t} \in \boldsymbol{Term}^{\tau} \Rightarrow \mathsf{i}\mathsf{t} \in \boldsymbol{Term}^{\mathsf{S}\to\tau}$
3. $\mathsf{t} \in \boldsymbol{Term}^{\mathsf{S}\to\tau} \Rightarrow \mathsf{!}\mathsf{t} \in \boldsymbol{Term}^{\tau}$
4. $X \in \boldsymbol{Loc}^{\tau}$, $\mathsf{t} \in \boldsymbol{Term}^{\tau} \Rightarrow X^{\tau} \coloneqq \mathsf{t}^{\tau} \in \boldsymbol{Term}^{\mathsf{S}}$
5. $\mathsf{t} \in \boldsymbol{Term}^{\mathsf{S}}$, $\mathsf{u} \in \boldsymbol{Term}^{\tau} \Rightarrow \mathsf{t}\,;\mathsf{u} \in \boldsymbol{Term}^{\tau}$

Supplementary operators are defined in a standard way [Ben10].

Notice that the intension of a term t of type τ is of type $\mathsf{S} \to \tau$, and the extension of an (intensional) term t of type $\mathsf{S} \to \tau$ is of type τ. The assignment construct is (always) of type S. Most interestingly, the sequence operator allows terms of type other than S to the right of assignments; for example we have the term $X^{\tau} \coloneqq X + 1\,; X$ of type N. This is discussed further in §3.5.

Assignments are of type S due to the fact that they return stores, and the type of the location (on the left-hand side) and the assigned term must be in agreement. We do not allow locations of type S, but we *do* allow locations of type $\mathsf{S} \to \tau$ for any τ. This amounts to storage of intensions in the store, which accounts for much of ***AC***'s expressive power.

As an example, think of $\mathsf{i}\mathsf{t}$ as representing the "text" of t, which, in an actual computer, is the way that procedures and programs are stored. Now consider the term

$$X \coloneqq \mathsf{i}\mathsf{!}X,$$

which is read "store in X the procedure that invokes X". Once this action is performed, an invocation of X

$$X \coloneqq \mathsf{i}\mathsf{!}X\mathsf{;}\ \mathsf{!}X$$

will proceed to invoke X again and again, leading to divergence.

3.3 Operational Semantics of AC

We access the contents of $\boldsymbol{X}$ in a store ς by function application $\varsigma(\boldsymbol{X})$.

We will use the standard notion of *function variant* to update the contents of a location, where the variant $f[x/d]$ of a function f at x for d is defined by: $(f[x/d])(x) = d$ and $(f[x/d])(y) = f(y)$ for $y \neq x$.

We employ "big-step" operational semantics [Win93]. The rules define the *computation relation* $\Downarrow \subset ((\boldsymbol{Term} \times \boldsymbol{Store}) \times (\boldsymbol{Term} \cup \boldsymbol{Store}))$. Really, since a term can compute *either* a store (if the term is of type S) *or* another term (if it is of any type other than S), the computation relation can be broken into two disjoint relations $\Downarrow_c \subset ((\boldsymbol{Term}^{\mathsf{S}} \times \boldsymbol{Store}) \times \boldsymbol{Store})$ and $\Downarrow_v \subset ((\boldsymbol{Term}^{\tau} \times \boldsymbol{Store}) \times \boldsymbol{Term})$. However, since the rules are so simple, we choose instead to use the metavariable d to range over *both* terms and stores, and give the rules as follows:

Definition 3.3 (Operational semantics of AC). *First, the rules for locations, assignment and sequence are standard [Win93]:*

$$\frac{\varsigma(\boldsymbol{X}) = \mathsf{t}}{\boldsymbol{X},\ \varsigma\ \Downarrow\ \mathsf{t}} \qquad \frac{\mathsf{t},\ \varsigma\ \Downarrow\ \mathsf{u}}{\boldsymbol{X} := \mathsf{t},\ \varsigma\ \Downarrow\ \varsigma[\boldsymbol{X}/\mathsf{u}]} \qquad \frac{\mathsf{t},\ \varsigma\ \Downarrow\ \varsigma' \qquad \mathsf{u},\ \varsigma'\ \Downarrow\ \mathsf{d}}{\mathsf{t};\ \mathsf{u},\ \varsigma\ \Downarrow\ \mathsf{d}}$$

The interesting new rules are those for intension and extension:

$$\frac{}{\mathsf{i}\mathsf{t},\ \varsigma\ \Downarrow\ \mathsf{i}\mathsf{t}} \qquad \frac{\mathsf{t},\ \varsigma\ \Downarrow\ \mathsf{i}\mathsf{u} \qquad \mathsf{u},\ \varsigma\ \Downarrow\ \mathsf{d}}{!\mathsf{t},\ \varsigma\ \Downarrow\ \mathsf{d}}$$

The rules for supplementary operators are standard, and omitted here.

The intuition for the rules for intension and extension are as follows:

- intension "holds back" the computation of a term,
- extension "induces" computation.

Note that there are *no side-effects in* $\boldsymbol{AC}$; if a term is of type $\tau \neq \mathsf{S}$ then it only results in a value. This is further discussed in §3.5.

Lemma 3.4. *Computation is* unique and deterministic*, i.e., there is at most one derivation for any* t, ς.

We return to our running example (3). Its operational interpretation is as follows. For any ς, $\varsigma' = \varsigma[\boldsymbol{P}/\mathsf{i}\boldsymbol{X}]$ and $\varsigma'' = \varsigma[\boldsymbol{P}/\mathsf{i}\boldsymbol{X}][\boldsymbol{X}/\mathbf{1}]$,

$$\dfrac{\dfrac{\mathsf{i}\boldsymbol{X},\ \varsigma\ \Downarrow\ \mathsf{i}\boldsymbol{X}}{\boldsymbol{P} := \mathsf{i}\boldsymbol{X},\ \varsigma\ \Downarrow\ \varsigma'} \qquad \dfrac{\dfrac{\mathbf{1},\ \varsigma'\ \Downarrow\ \mathbf{1}}{\boldsymbol{X} := \mathbf{1},\ \varsigma'\ \Downarrow\ \varsigma''} \qquad \dfrac{\dfrac{\varsigma''(\boldsymbol{P}) = \mathsf{i}\boldsymbol{X}}{\boldsymbol{P},\ \varsigma''\ \Downarrow\ \mathsf{i}\boldsymbol{X}} \qquad \dfrac{\varsigma''(\boldsymbol{X}) = \mathbf{1}}{\boldsymbol{X},\ \varsigma''\ \Downarrow\ \mathbf{1}}}{!\boldsymbol{P},\varsigma''\ \Downarrow\ \mathbf{1}}}{\boldsymbol{X} := \mathbf{1};\ !\boldsymbol{P},\ \varsigma''\ \Downarrow\ \mathbf{1}}}{\boldsymbol{P} := \mathsf{i}\boldsymbol{X};\ \boldsymbol{X} := \mathbf{1};\ !\boldsymbol{P},\ \varsigma\ \Downarrow\ \mathbf{1}}$$

3.4 Rigidity, Modal Closedness, and Canonicity

Definition 3.5 (Operational rigidity). *A term* t *is called* operationally rigid *iff there is a* u *s.t. all stores* ς *give* $\mathsf{t}, \varsigma \Downarrow \mathsf{u}$. *A term for which this is not the case is called operationally* non-rigid.

To provide a *syntactic* approximation to rigidity, we follow Montague and define the set of *modally closed* terms as follows:

Definition 3.6 (Modal Closedness). *The set of modally closed terms* $\boldsymbol{MC}$, *ranged over by* mc, *is generated by*

$$\mathsf{mc} \ ::= \ \mathsf{b} \mid \mathsf{n} \mid \mathsf{i}\,\mathsf{t} \mid \mathsf{mc}_1 + \mathsf{mc}_2 \mid \ldots$$

with similar clauses for other arithmetic, comparison and boolean operators.

Lemma 3.7. $\mathsf{t} \in \boldsymbol{MC} \Longrightarrow \mathsf{t}$ *is operationally rigid.*

Modal closedness captures the intuitive notion of a completed imperative computation, but it can leave arithmetic and boolean computations "unfinished". Terms in which all of these computations are also complete are called *canonical* and are defined by:

Definition 3.8 (Canonical terms). *The set of canonical terms* $\boldsymbol{Can}$, *ranged over by* c, *is generated by:*

$$\mathsf{c} \ ::= \ \mathsf{b} \mid \mathsf{n} \mid \mathsf{i}\,\mathsf{t}$$

Clearly, $\boldsymbol{Can} \subseteq \boldsymbol{MC}$. Hence:

$$\mathsf{t} \in \boldsymbol{Can} \ \Rightarrow \ \mathsf{t} \in \boldsymbol{MC} \ \Rightarrow \ \mathsf{t} \text{ is operationally rigid.}$$

Definition 3.9 (Properness of Stores). *A store* ς *is* proper *if it maps locations only to canonical terms.*

The main result of this section is that the operational semantics is well defined, i.e., it only produces canonical terms or proper stores:

Theorem 3.10 (Properness of Operational Semantics). *If* ς *is proper, then for any* t *where* t, ς *converges:*

1. *If* t *is of type* S *then there is a proper store* ς' *s.t.* $\mathsf{t}, \varsigma \Downarrow \varsigma'$;
2. *Otherwise, there is a (canonical)* c *s.t.* $\mathsf{t}, \varsigma \Downarrow \mathsf{c}$.

Proof. By induction on derivations. Details in [Ben10].

3.5 State Backtracking

A significant difference between $\boldsymbol{AC}$ and traditional imperative languages is that there are *no side-effects*. In $\boldsymbol{AC}$, terms represent either *stores* (effects), if they are of type S, or *values* if they are of some other type.

In this respect, note that a language based on side-effects can easily be translated into $\boldsymbol{AC}$. Further, the absence of side-effects in $\boldsymbol{AC}$ leads to an interesting phenomenon: *state backtracking*.

State backtracking, or *non-persistent* or *local state update*, occurs when we have terms of the form

$$\mathsf{t}\,;\mathsf{u}^{\tau} \quad \text{where} \quad \tau \neq \mathsf{S} \tag{4}$$

because state changes caused by t are "lost," "localized" or "unrolled" when the value of u is returned. Consider the following example:

$$\boldsymbol{Y} := (\boldsymbol{X} := \mathbf{1};\ \boldsymbol{X}).$$

Changes to $\boldsymbol{X}$ are *local to the computation of* $\boldsymbol{Y}$, so in fact the above term is equivalent to '$\boldsymbol{Y} := \mathbf{1}$', a result which can be confirmed easily by the reader (for details, see [Ben10]).

The inclusion of terms of the form (4), makes possible a clean *rewriting system* for $\boldsymbol{AC}$: it gives us a way to "push assignments into terms". We discuss this further in §4; for now, consider this example: $\boldsymbol{X} := \mathbf{1};\ \boldsymbol{X} := (\boldsymbol{X} + \boldsymbol{X})$. By intuition and the operational semantics, we know that this term is equivalent to $\boldsymbol{X} := \mathbf{2}$. But how can it be possible, without overly complicated rules, to *rewrite* the former to the latter? By admitting terms like (4), we can express *intermediate steps* of the computation that could not otherwise be written. Using the rewriting rules (Definition 4.1),

$$\begin{aligned} \boldsymbol{X} := \mathbf{1};\ \boldsymbol{X} := (\boldsymbol{X} + \boldsymbol{X}) &\Rrightarrow \boldsymbol{X} := (\boldsymbol{X} := \mathbf{1};\ (\boldsymbol{X} + \boldsymbol{X})) \\ &\Rrightarrow \boldsymbol{X} := ((\boldsymbol{X} := \mathbf{1};\ \boldsymbol{X}) + (\boldsymbol{X} := \mathbf{1};\ \boldsymbol{X})) \\ &\Rrightarrow \boldsymbol{X} := (\mathbf{1} + \mathbf{1}) \Rrightarrow \boldsymbol{X} := \mathbf{2} \end{aligned}$$

Thus, based on its usefulness and the simplicity of the resulting rewriting rules, we believe that

State backtracking is a natural part of imperative computation.

4 Term Rewriting

In this section we explore $\boldsymbol{AC}$ by examining meaning-preserving transformations of terms. What we will develop is essentially the "calculus" part of Assignment Calculus—a term rewriting system whose rules are meant to capture and exploit its essential equivalences.[1]

[1] In developing these rules, we find significant guidance in Janssen's [Jan86] and Hung's [Hun90] work on *state-switcher reductions*.

4.1 Rewrite rules and properties

In order to make the rewriting definitions simpler, we adopt the convention that terms are syntactically identical regardless of parenthesization of the sequencing operator; to wit, $(\mathbf{t}\,;\mathbf{u})\,;\mathbf{v} \;\equiv\; \mathbf{t}\,;(\mathbf{u}\,;\mathbf{v})$ This convention makes it much easier to express rewriting rules that govern the interaction of assignment operators.

The heart of the rewriting system is the rewriting function $\Rrightarrow : \boldsymbol{Term} \to \boldsymbol{Term}$. Recall the definition (3.6) of modally closed terms $\mathbf{mc}$.

Definition 4.1. *The rewriting function* $\Rrightarrow$ *is given by*

1. $!\,\mathbf{i}\mathbf{t} \;\Rrightarrow\; \mathbf{t}$
2. $\boldsymbol{X} := \mathbf{mc}_1;\ \mathbf{mc}_2 \;\Rrightarrow\; \mathbf{mc}_2$
3. $\boldsymbol{X} := \mathbf{t};\ \boldsymbol{X} \;\Rrightarrow\; \mathbf{t}$
4. $\boldsymbol{X} := \mathbf{mc};\ \boldsymbol{Y} \;\Rrightarrow\; \boldsymbol{Y}$
5. $\boldsymbol{X} := \mathbf{mc};\ \boldsymbol{X} := \mathbf{u} \;\Rrightarrow\; \boldsymbol{X} := (\boldsymbol{X} := \mathbf{mc};\ \mathbf{u})$
6. $\boldsymbol{X} := \mathbf{mc};\ \boldsymbol{Y} := \mathbf{t} \;\Rrightarrow\; \boldsymbol{Y} := (\boldsymbol{X} := \mathbf{mc};\ \mathbf{t});\ \boldsymbol{X} := \mathbf{mc}$
7. $\boldsymbol{X} := \mathbf{mc};\ !\,\mathbf{t} \;\Rrightarrow\; \boldsymbol{X} := \mathbf{mc};\ !(\boldsymbol{X} := \mathbf{mc};\ \mathbf{t})$

Definition 4.2. *The rewrite relation* $\Rrightarrow \;\subset (\boldsymbol{Term} \times \boldsymbol{Term})$ *is defined by:* $\mathbf{t} \Rrightarrow \mathbf{u}$ *iff* $\mathbf{u}$ *results from applying* $\Rrightarrow$ *to a subterm of* $\mathbf{t}$. *If* $\mathbf{t} \Rrightarrow \cdots \Rrightarrow \mathbf{u}$ *(including if* $\mathbf{t} \equiv \mathbf{u}$*), then we write* $\mathbf{t} \Rrightarrow\!\!\!\!\Rrightarrow \mathbf{u}$*; that is,* $\Rrightarrow\!\!\!\!\Rrightarrow$ *is the reflexive-transitive closure of* $\Rrightarrow$.

Some brief remarks are in order to explain the rewrite rules. Rule 1 expresses a basic property of Montague's intension and extension operators. In our setting, it embodies the *execution of a procedure*. Rule 7 is very important: it is the *recursion rule*. It may be difficult to see immediately why we identify this rule with recursion; the following special case, which combines the use of rules 7, 3 and 1, illustrates the concept more clearly:

$$\boldsymbol{X} := \mathbf{i}\mathbf{t}\,;!\boldsymbol{X} \;\Rrightarrow\!\!\!\!\Rrightarrow\; \boldsymbol{X} := \mathbf{i}\mathbf{t}\,;\mathbf{t}.$$

This amounts to simple substitution of a procedure body for its identifier, while "keeping a copy" of the procedure body available for further substitutions.

Our first order of business is to show that the rewrite function does not change the meaning of a term.

Theorem 4.3 (Validity of rewrite rules). $\mathbf{t} \Rrightarrow \mathbf{u} \;\Longrightarrow\; (\mathbf{t}, \varsigma \Downarrow \mathbf{d} \;\Leftrightarrow\; \mathbf{u}, \varsigma \Downarrow \mathbf{d})$

Proof. By cases on the rewrite rules. This proof also provides constructive "rules of replacement" for parts of operational derivation trees: each rewrite rule corresponds to a transformation on derivation trees. □

We can gain some valuable insight into how to use the rewriting rules by using them to interpret our running example (3):

$$\begin{aligned}
\boldsymbol{P} := \mathbf{i}\boldsymbol{X};\ \boldsymbol{X} := 1;\ !\boldsymbol{P} &\Rrightarrow\!\!\!\!\Rrightarrow\ \boldsymbol{X} := 1;\ \boldsymbol{P} := \mathbf{i}\boldsymbol{X};\ !\boldsymbol{P} \\
&\Rrightarrow\!\!\!\!\Rrightarrow\ \boldsymbol{X} := 1;\ \boldsymbol{P} := \mathbf{i}\boldsymbol{X};\ !(\boldsymbol{P} := \mathbf{i}\boldsymbol{X};\ \boldsymbol{P}) \\
&\Rrightarrow\!\!\!\!\Rrightarrow\ \boldsymbol{X} := 1;\ \boldsymbol{P} := \mathbf{i}\boldsymbol{X};\ !\mathbf{i}\boldsymbol{X} \\
&\Rrightarrow\!\!\!\!\Rrightarrow\ \boldsymbol{X} := 1;\ \boldsymbol{P} := \mathbf{i}\boldsymbol{X};\ \boldsymbol{X} \;\Rrightarrow\!\!\!\!\Rrightarrow\; 1
\end{aligned}$$

4.2 Equivalence of Interpretations

In this subsection we demonstrate that the rewriting system provided by the $\Rrightarrow$ relation is equivalent to the operational interpretation. Before stating this theorem, however, we need to address some technicalities.

In order to use the rewriting rules to arrive at the same result as the operational semantics, we need to take into account the *store*. That means that we need a way to take the required information from the store and *actualize* it as a term. For example, take the term $\mathbf{t} \equiv \boldsymbol{X} + \boldsymbol{X}$ and a store ς that maps $\boldsymbol{X}$ to $\mathbf{1}$; then, $\mathbf{t}, \varsigma \Downarrow \mathbf{2}$. We can accomplish this in rewriting by *prepending* $\mathbf{t}$ with $\boldsymbol{X} := \mathbf{1}$, which gives $\boldsymbol{X} := \mathbf{1};\ (\boldsymbol{X} + \boldsymbol{X}) \Rrightarrow \mathbf{2}$ as needed. For technical convenience, we take the set of locations $\boldsymbol{Loc}$ to be finite ([Ben10] extends the treatment to a countable set).

Definition 4.4 (Store terms, store actualization and store abstraction). *The store-actualization of ς, assuming $\boldsymbol{Loc} = \{\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n\}$, is defined as*

$$\langle \varsigma \rangle \underset{\text{def}}{=} \boldsymbol{X}_1 := \varsigma(\boldsymbol{X}_1)\ ;\ \ldots\ ;\ \boldsymbol{X}_n := \varsigma(\boldsymbol{X}_n).$$

Terms like those above we call store-terms. The set of store-terms $\boldsymbol{STerm}$, ranged over by $\mathbf{s}$, is the set of terms of the form

$$\boldsymbol{X}_1 := \mathbf{c}_1\ ;\ \ldots\ ;\ \boldsymbol{X}_n := \mathbf{c}_n$$

We also define an inverse to the store-actualization operator, store abstraction, which takes a store term and returns the corresponding store, such that $[\langle \varsigma \rangle] = \varsigma$ and $\langle [\mathbf{s}] \rangle$ is $\mathbf{s}$ with its assignment statements reordered into some arbitrary canonical ordering.

A convenient lemma shows the operational soundness of the above notions.

Lemma 4.5. $\mathbf{t}, \varsigma \Downarrow \mathbf{d} \iff \forall \varsigma' \cdot (\langle \varsigma \rangle ; \mathbf{t}), \varsigma' \Downarrow \mathbf{d}$. □

We now present the first part of the proof of equivalence between the operational semantics and term rewriting.

Theorem 4.6 (Operational adequacy).

1. *If $\mathbf{t}, \varsigma \Downarrow \mathbf{c}$, then $\langle \varsigma \rangle ; \mathbf{t} \Rrightarrow \mathbf{c}$;*
2. *If $\mathbf{t}, \varsigma \Downarrow \varsigma'$, then $\langle \varsigma \rangle ; \mathbf{t} \Rrightarrow \langle \varsigma' \rangle$.*

Proof. By induction on derivations. Details can be found in [Ben10]. □

The above theorem only works in one direction, in that it only shows that the rewriting system is at least as strong as the operational semantics. In fact it is (syntactically speaking) even stronger; as a simple demonstration of this, consider the term $\mathbf{i}(\mathbf{1} + \mathbf{1})$. Operationally, it is inert: it returns itself. However, the rewrite rules allow us to "reach inside" the intension and rewrite it to $\mathbf{i}\mathbf{2}$.[2]

[2] This was not an issue in [Ben10] because the equivalence proof there was stated in terms of denotational semantics; here we strive for a stronger result about the syntax itself.

The operational semantics and rewriting system are therefore syntactically equivalent only *up to rewriting of unevaluated intensions.* The following theorem completes the equivalence result.

Theorem 4.7 (Rewriting adequacy).

1. *If* $\mathbf{t} \Rrightarrow \mathbf{c}$*, then there is a* $\mathbf{c'}$ *s.t.* $\mathbf{c} \Rrightarrow \mathbf{c'}$ *and for any* ς'*,* $\mathbf{t}, \varsigma' \Downarrow \mathbf{c'}$*.*
2. *If* $\mathbf{t} \Rrightarrow \mathbf{s}$*, then there is a* $\mathbf{s'}$ *s.t.* $\mathbf{s} \Rrightarrow \mathbf{s'}$ *and for any* ς''*,* $\mathbf{t}, \varsigma'' \Downarrow [\mathbf{s'}]$*.*

Proof. By induction on the length of the rewrite sequence $\mathbf{t} \Rrightarrow \mathbf{c}$ or $\mathbf{t} \Rrightarrow \mathbf{s}$. The inductive step consists roughly of identifying the affected parts of the operational derivation tree and substituting a suitably modified tree (using Theorem 4.3); or, if the rewrite step affects only an uninterpreted (i.e., bound by intension) part of the term, to add the rewrite to $\mathbf{c} \Rrightarrow \mathbf{c'}$ or $\mathbf{s} \Rrightarrow \mathbf{s'}$ as applicable. □

By combining Theorems 4.6 and 4.7, using Lemma 4.5, we obtain our desired equivalence result.

We conclude this section by mentioning that we have also attained a confluence result for our term rewriting system [Ben10, App. B]. It will be discussed in detail in a forthcoming publication.

Theorem 4.8 (Confluence). *If* $\mathbf{t} \Rrightarrow \mathbf{u}$ *and* $\mathbf{t} \Rrightarrow \mathbf{v}$*, then there is a* $\mathbf{w}$ *s.t.* $\mathbf{u} \Rrightarrow \mathbf{w}$ *and* $\mathbf{v} \Rrightarrow \mathbf{w}$*.*

5 Conclusion

We hope to have convinced the reader, in the last four sections, that ***AC*** does indeed possess the desirable properties listed in §2.1. (a) It is *small*, *elegant* and (hopefully) *intuitive*: as discussed in §2, ***AC***'s set of four basic operators is simple and understandable; (b) its operators represent well-understood, fundamental concepts: by taking only assignment, sequence, procedure formation and procedure invocation as basic, we remain close to practical imperative languages; (c) we can *rewrite* its terms using simple rules: Definition 4.1 demonstrates this; and (d) it has multiple forms of semantics that are equivalent: in this paper we demonstrated this equivalence for operational semantics and term rewriting, [Ben10] extends this equivalence to denotational semantics as well.

We believe that the above points show that Assignment Calculus is a realization of our goal to develop a true *imperative computation language.*

5.1 Related Work

The final step in our presentation is to explore related and otherwise relevant work. First, note that there has been no other attempt, as far as the authors are aware, to define a pure imperative computation language as we have done. Therefore all of the other work that we will examine is only indirectly related to our aims; nevertheless there are certainly interesting connections to be explored.

The first and perhaps most striking language of interest that can be found in the literature is (unfortunately!) nameless; it is defined in the seminal report of Strachey and Scott [SS71, §5]. Here we find a language that has features that closely resemble those of ***AC***: there are operators for referencing and dereferencing, and operators for treating a procedure as an expression and an expression as a procedure. This language does not appear to have been revisited in subsequent work.

Insofar as *rewriting systems for imperative-style languages*, the prime example is the work of Felleisen [FH92]. He adds facilities for handling state and control operators to Plotkin's call-by-value λ-calculus, which results in a quite elegant system. There could be a good deal of interesting work in comparing our work with Felleisen's; the main problem that we immediately encounter is that his work depends fundamentally on the λ-calculus, which is precisely what we have tried to avoid incorporating into ***AC***.

5.2 Future Work

We now discuss interesting avenues of further research. First, we should like to continue exploring, expanding, and improving the rewriting system of §4. Our present goal was to arrive at a small set of rules that was sufficient to achieve our equivalence proof; however, it would be useful to develop more powerful rules that might be more intuitive in terms of real-world use. In fact we have already made significant progress in this direction while developing the proof of confluence in [Ben10, App. B]; the work will be elaborated in forthcoming work.

It would be interesting to examine more carefully the concept of *state backtracking* in ***AC***. As mentioned in §3.5, we believe that state backtracking is a fundamental part of imperative computation; therefore, we would like to provide an improved analysis of what it comprises and how it takes part in and affects imperative computation. Along these lines, it is important to explore connections with Separation Logic [Rey02], particularly its interesting store model, and with Felleisen's work as mentioned above.

The aim of this paper was to provide an analysis of imperative computation. We have done this on multiple levels: from a philosophical dissection of the concept of imperative computation in §2, to developing the actual types and syntax of ***AC***, and finally to ***AC***'s operational and rewriting-based meanings. We believe that this broad approach leaves ***AC*** well-prepared for further investigations, and we hope that it will stimulate future work in what we consider to be an exciting new approach to an old paradigm.

References

[Bac78] John Backus. Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[Bar84] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984.

[Ben10] Marc Bender. *Assignment Calculus: A Pure Imperative Reasoning Language*. PhD thesis, McMaster University, 2010.

[Car47] Rudolf Carnap. *Meaning and Necessity*. University of Chicago, 1947.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton, 1941.

[Chu51] Alonzo Church. A formulation of the logic of sense and denotation. In Paul Henle, editor, *Structure, Method and Meaning: Essays in Honor of Henry M. Sheffer*, pages 3–24. Liberal Arts Press, 1951.

[FH88] A. J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, July 1988.

[FH92] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[Fre92] Gottlob Frege. Über Sinn und Bedeutung. *Zeitschrift für Philosophie und philosophische Kritik*, 100:25–50, 1892.

[HHJW07] P. Hudak, J. Hughes, S.P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *HOPL III: Proc. 3rd ACM SIGPLAN Conf. on History of Prog. Languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[Hun90] Hing-Kai Hung. *Compositional Semantics and Program Correctness for Procedures with Parameters*. PhD thesis, SUNY-Buffalo, 1990.

[HZ91] Hing-Kai Hung and Jeffery Zucker. Semantics of pointers, referencing and dereferencing with intensional logic. *Proc 6th Annual IEEE Symposium on Logic in Computer Science*, pages 127–136, 1991.

[Jan86] Theo M.V. Janssen. *Foundations and Applications of Montague Grammar: Part 1: Philosophy, Framework, Computer Science*. Centrum voor Wiskunde en Informatica, Amsterdam, 1986.

[JvEB77] T.M.V. Janssen and P. van Emde Boas. On the proper treatment or referencing, dereferencing and assignment. In *ICALP*, volume 52 of *Lecture Notes in Computer Science*, pages 282–300. Springer, 1977.

[Kri59] Saul A. Kripke. A completeness theorem in modal logic. *Journal of Symbolic Logic*, 24:1–14, 1959.

[Mon70] Richard Montague. Universal grammar. *Theoria*, 36:373–398, 1970.

[Mon73] Richard Montague. The proper treatment of quantification in ordinary English. In K.J.J. Hintikka, J.M.E. Moravcsik, and P. Suppes, editors, *Approaches to Natural Language*, pages 221–242. Reidel, 1973.

[Qui60] W. V. Quine. *Word and Object*. MIT Press, Cambridge, MA, 1960.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[Sco70] Dana S. Scott. Outline of a mathematical theory of computation. Tech. Monograph PRG–2, Oxford University Computing Laboratory, 1970.

[SS71] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In J. Fox, editor, *Proc. Symp. on Computers and Automata*, volume XXI, pages 19–46, Brooklyn, N.Y., April 1971. Polytechnic Press.

[Tur36] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.