Abstract Identifiers and Textual Reference

Stuart Frazier Allen*

November 2002

Abstract

Here are three proposals concerning the structure and maintenance of formal, inter-referential, digitally stored texts: (1) include abstract atomic identifiers in texts, (2) identify these identifiers with references to text objects, and (3) keep among the texts records of computationally substantiated claims about those texts. We use "formal" in a narrow sense approximating computer-checkable.

We are informed by informal symbolic practices used in mathematical text and program source text, which we hope to enhance and exploit explicitly; the basic management problem is how to alter texts *rather freely* without ruining the bases for claims depending upon them, which becomes an issue of accounting for various dependencies between texts. We are *not* here proposing the use of abstract structured text; nonetheless, experience using it in Nuprl4 has led us to appreciate the benefits of distinguishing abstract form from concrete presentation, and also has shown us the cognitive and practical *un*importance of just which identifiers occur in abstract structured texts when texts are mediated by a system that realizes concrete presentation.

Abstract treatment of identifiers involves concrete realization during communications between "text servers" and their clients. The benefit of treating identifiers abstractly is a radical avoidance of name collision, even at runtime, and is important for claims about texts that are based upon program execution. The notion of text collections and equivalence of text collections modulo change of identifiers is made precise by the second proposal. The complete identification of abstract identifiers with reference values is discussed, addressing the issues of dangling pointers, the association of ordinary symbolic identifiers with meaningful defining texts, the "flatness" of the pointer space, and the perhaps counterintuitive collapse of two abstract name spaces into one.

The notion of "certification system" is introduced as a formalization of generic computationally defined claims about texts, emphasizing the diversity of clients who may not agree on a common "logic". The notion of a certificate whose computational meaning, in the context of the texts it refers to, is completely specified (although perhaps non-deterministic), and the notion of a certificate further being deterministic, are introduced and elaborated with regard to their epistemic value. What it means to give certificate texts the force of factual records, and mechanisms to accomplish this, are discussed. Scenarios for practically exploiting identifier abstractness and fully deterministic certificates are considered, involving the combination of partially independently developed texts and the experimental modification of texts in a collection. The importance of implementing multiple certification systems is articulated.

*Cornell University Computer Science Department, www.cs.cornell.edu/home/sfa. Supported in part by ONR Grant #N00014-01-1-0765.

Contents

1 Introduction

Here are three proposals concerning the structure and maintenance of formal, inter-referential, digitally stored texts, such as formal mathematical proofs and source code for programs, which are often intended both for human understanding and for use as data by computer processes. Large collections of formal texts can be difficult to manage, with regard to their meaning and correctness, even for an individual, and it becomes more difficult when one attempts to combine work of distinct individuals developed independently or partially independently. These proposals are made with the intention of liberating developers and consumers of such material from various management problems involving naming (such as name collision when combining materials and other name-changing situations), as well as to provide a basis for managing intertext dependencies by making it possible to stipulate criteria for when established claims about a body of texts should be called into doubt as a result of changing or deleting a text. The first proposal is to syntactically include "abstract" atomic identifiers directly in the texts, either supplanting or supplementing concrete identifiers in most cases; the second is to identify these abstract identifiers with references to texts, both as they occur in referring texts and as "runtime" values during computations; the third is to keep, among the texts, records of computationally substantiated claims.

These proposals have in fact been adopted by the PRL group [6, 1] for the purpose of designing and implementing software for storing and managing "logical libraries" grounded in formal texts. The management of logical dependency is paramount since the designers have adopted a neutral stance with regard to various logical systems, presupposing no one criterion for correctness of proofs, and they aim to incorporate disparate kinds of formal texts, developed with external tools. The coexistence of such multifarious texts entails the need to avoid accidental dependency upon untrusted elements, while simultaneously supporting users with different criteria of trust, which is essential to encouraging the sharing of data between diverse clients.

The relevant sense of "formal" here is having precise meaning or objective criteria of correctness, ideally computer verifiable, based simply upon syntactic form. This is the sense of the word most relevant to our endeavor; this usage, which has some currency, stands in contrast to other common meanings such as being rigid, ceremonious, solemn, customary or not casual. It is closer to meaning exact, methodical or orderly, but for the purposes of supporting precise understanding or procedures that can be performed by machines. Computer checkable proofs and computer executable programs are paradigmatic. Of course, if these proposals are adopted for formal texts, they can also be exploited in informal texts, such as explanations or discussions, that piggyback on the formal texts and references, and it seems likely that such informal texts are essential to making practical use of formal texts, but the driving principle is to facilitate management of (and accounting for) the formal.

2 Abstract Identifiers and Intertextual Reference

Here we make the first two proposals concerning what might be considered "data" as opposed to "fact", but upon which our proposal for recording facts will depend. Indeed, it is the intention of recording facts that ultimately motivates the proposals to treat identifiers abstractly and to formalize intertextual reference.

2.1 Informal Textual Practices

Let us consider some pertinent informal practices. In ordinary mathematical text, specific symbols or names are introduced both for notation of mathematical entities and for reference to objects occurring in the discourse itself, such as definitions, theorems and proofs. Examples might be: "$+$", "0", "$\exists$", "$\in$", "prime number", "Fermat's last theorem", "Lemma 2.1 above". When such material is organized as data for computer manipulation there is plainly some benefit to explicitly recognizing occurrences of "identifiers" used for these purposes. Obvious possibilities include readers following references by mouse-clicking, automated theorem verification, search based on symbolic content, and automated dependency tracking. For these purposes we would break the whole text into smaller texts suitable for such reference. In informal texts, once a notation is introduced it must be used without explicit reference to the notation-defining text, which causes the reader to scan back looking for the defining text when need be. This is an obvious opportunity for enhancing a formalized notation by building into the formal syntax for the notation an explicit reference to the defining text, although the presentation of the text for a human reader would normally suppress this reference while keeping it available for use on demand.

Reference between separate informal texts, such as papers or books, is achieved by bibliographical description, often mediated throughout most of a text by abbreviatory references to such bibliographical descriptions. Naturally if the referenced text is represented in the same "space" as the referring text, then a direct reference is possible in lieu of bibliographical description, which would then become annotation of the referenced text rather than the mechanism of reference.

Source texts for programs, the commonest formal texts, are replete with identifiers intended to refer to independently defined globals such as procedures, types, classes, modules or assignable variables, and often use macros or other programmer definable syntactic extensions (which play the role that notationally eliminable definitions play in mathematical text). It often happens that particular meanings are assumed for these symbols which may not be expressed in the program text itself, but rather are implicit in the collection of several such source texts coordinated for purposes of execution. While this treatment of all source text identifiers as essentially uninterpreted can surely be justified when attempting to treat the source text as re-interpretable at each of these symbols, nonetheless, often the programmer is *not* thinking of them as having variable meaning but intends them to be determined by other particular extant source texts. This intention could be formalized by building into the identifier a reference to the intended source text. Similarly the intended sources for notational definitions (macros) could be incorporated into source text references in the manner suggested above for formal mathematical notations.

Even if one did not include a source text reference with every identifier in a program source text P, one could still explicitly make such an association in another text by combining reference to P and a pairing of identifiers in P with appropriate references to source texts for their intended meanings. The point is that one can achieve a flexible explicit control over specifying sources to be coordinated for execution, which is essential when one uses such code to maintain specific and checkable claims about collections of texts, such as formal arguments. Similar management techniques have been implemented with the VESTA system [13].

Naturally, once text references are admitted into program source text, one can also refer in program source text to other arbitrary text for use as runtime data, not just as source code text for building an executable.

It is worth noting that the formal digitally manipulated counterparts of these two paradigmatic domains, formal verification and programming, are not independent. In some formal proof development systems (especially "tactic" based systems [5]) program source text is included as part of the proof to indicate how the proof is to be verified, and such proof verification programs often make explicit reference to proofs as data (as lemmas to be cited, for example). And of course, when a formal argument is about particular program texts, those program texts will be cited in the assertions of the proof. The correctness of claims based on collections of formal texts obviously will often depend upon what those texts contained and how they referenced one another when the claims were established.

The basic management problem for formal texts is how to extend or otherwise alter them *rather freely* without accidentally ruining the bases for claims depending upon them; this becomes an issue of accounting for various dependencies between texts. A collection of proof texts might be ruined, for example, if the proof of some lemma were replaced by a new proof using a new axiom not accepted by the reader. Or a program text might be ruined by alteration of a critical subprogram that makes the program as a whole useless.

2.2 Conventional Runtime References

Since we propose to use identifiers as references, let us digress and consider familiar computational devices for reference. Presumably there is some operation on a class of reference values implementing the "content of" or "referent of" reference values, which is typically provided as an implicit argument to the execution of programs. The nature of the reference value might be that it is structured from parts in some referentially relevant way: for example it might be a sequence of units specifying a "path" to the referent, perhaps a URL for web lookup, or a file system path, or it might be a RAM address depending on a subtler relation between its numeric structure and its referential significance. On the other hand the reference value might be referentially atomic, meaning that it has no particular internal structural relation to any other reference values; maybe it is used as an index in an association list.

Reference values might also be either concrete or abstract; character strings and numbers would be examples of concrete values, whether treated as referentially atomic or not, whereas pointer values as in SIMULA, say, would be abstract[1], since they can be used computationally only in comparisons

[1] The notion of abstract reference values seems to have arisen independently with SIMULA [14] and the ancestors of PASCAL, namely EULER [20] and ALGOL-W [19]. Little seems to have been made of it by the authors, perhaps because of its minor significance relative to the other innovations. Conjecture: when one incorporates ram addresses into an Algol60-like semantic method, as with both the above lines of development, they tend to become abstract.

to other pointer values, or to look up their referents, or as whole data passed between computations, and the only way to get a pointer value (other than nil) initially is by allocation of a referent for one. In contrast, concrete values have some "external" identity independent of this or that execution of a program. It would also be possible to base a reference function on a mixed mode, such as values being structured complexes of abstract atoms. Below, our second proposal will be to *identify* identifiers with references to texts and to treat them as both abstract and atomic.

2.3 Constraints on Structure of Individual Texts

Now we consider the structure of individual texts. We shall be rather careful in our parameterizations at this point, assuming the reader has a rough idea of the concept of text we aim at. We assume there is a class $\mathrm{Text}(D)$ of possible texts where D is whatever type of values will play the part of our distinguished kind of identifiers in texts. That is, we shall parameterize our concept of text by the class of identifiers; as will be seen, this is not simply rhetorical – quantification over suitable types for D will be key to our treatment of identifiers as abstract.

Before considering plausible candidates for $\mathrm{Text}(D)$, let us present the properties of $\mathrm{Text}(D)$ actually important for the proposals we shall make. Basically we presuppose the concept of identifier occurrences within a text and the concept of replacing those occurrences by other identifiers, as well as the derivative concept "$t/{:=}g$" of uniformly replacing identifiers throughout a text t by applying function g to them. We shall be more precise about this here, which the reader may regard as either pedantic or as salubrious demonstration of definiteness.

We assume that there is a precise notion of identifier occurrences in texts, for the distinguished sort of identifiers we're trying to introduce. Let the class $\mathrm{Occ}(t)$ index the identifier occurrences within $t \in \mathrm{Text}(D)$, of which we assume there are finitely many, indexing them independently of both D and which particular identifiers occur in t; we might think of these as places in the text where identifiers can go. We shall presuppose functions $\mathrm{id}(t)@(x)$, computing the identifier for each place x in t, and t/f, which assigns an identifier to each place according to an assignment function f, characterized by:

$$\forall t{:}\mathrm{Text}(D).\ \mathrm{Occ}(t) \text{ finite}$$

$$\forall t{:}\mathrm{Text}(D'),\ f{:}(\mathrm{Occ}(t) \to D).\ t/f \in \mathrm{Text}(D)$$

$$\forall t{:}\mathrm{Text}(D'),\ f{:}(\mathrm{Occ}(t) \to D).\ \mathrm{Occ}(t/f) = \mathrm{Occ}(t)$$

$$\forall t{:}\mathrm{Text}(D''),\ g{:}(\mathrm{Occ}(t) \to D'),\ f{:}(\mathrm{Occ}(t) \to D).\ t/g/f = t/f$$

$$\forall t{:}\mathrm{Text}(D).\ \mathrm{id}(t)@ \in \mathrm{Occ}(t) \to D$$

$$\forall t{:}\mathrm{Text}(D),\ f{:}(\mathrm{Occ}(t) \to D).\ t = t/f \Leftrightarrow f = \mathrm{id}(t)@$$

$$\mathrm{Text}(D) \text{ is monotonic in } D$$

$$\forall t{:}\mathrm{Text}(D).\ t \in \mathrm{Text}(\mathrm{Ids}(t)), \text{ where } \mathrm{Ids}(t) \equiv_{\mathrm{def}} \{\ \mathrm{id}(t)@(x){:}D \mid x \in \mathrm{Occ}(t)\ \}$$

and consequently

$$\forall t{:}\mathrm{Text}(D).\ t = t/\mathrm{id}(t)@$$

$$\forall t{:}\mathrm{Text}(D'),\ f{:}(\mathrm{Occ}(t) \to D).\ \mathrm{id}(t/f)@ = f \in \mathrm{Occ}(t) \to D.$$

Then $t/{:=}g \equiv_{\mathrm{def}} t/(g \circ \mathrm{id}(t)@)$ is the uniform replacement of each identifier x in t by $g(x)$.

We further assume that identity on $\mathrm{Text}(D)$ is effectively decidable when identity on D is. This is all we shall require of the relation between texts and their constituent identifiers for the purpose of our three proposals. Still we shall discuss some plausible candidates for $\mathrm{Text}(D)$ in order to anticipate its applications.

2.4 Candidates for Text

Thus far we have referred to the notion of text both as a rough familiar concept and "axiomatically" by precise stipulation of what our proposals presuppose. Now we take the complementary approach of describing more precisely how Text(D) might be realized. This should further convey the intent of the proposals as well as facilitate the imagination of how one might exploit the proposed methods.

We focus on a couple of extremes: (1) extended character strings and (2) a more complex recursive type. If one conceives of the *a priori* textual medium as essentially character strings variously interpreted, usually parsed, then one might adopt as Text(D) the finite sequences each of whose units is either a character or else an "identifier", i.e. a member of D. A similar alternative would be extending the concept of already-tokenized strings rather than character strings, making insertion of our new identifiers a bit more natural since they would then fill places likely to be occupied by identifier-like tokens of the preexisting kind of text. Let us use "string text" to mean either of these sequential structures.

Of course, for most formal purposes string text is likely to be parsed into some recursive abstract structure as an intermediate form, mediating in particular the formal meaning attributed to the string text. Indeed, one might have conceived ones texts to actually have had these more directly meaningful structures in the first place, and considered the strings seen when viewing these structured texts as mere (yet visually necessary) presentations of the underlying structured texts. This is a view encouraged by structure editors such as were pioneered by Teitelbaum and Reps [17, 16], and is the view explicitly adopted in the late 1980s as the basis of texts in the Nuprl systems [12].

Here is a rather simple kind of recursive text structure: a text consists of a finite (possibly empty) sequence of immediately constituent texts as well as a finite sequence of "pro-textual" constituent values, where these pro-textual values are taken to be independent of the text structure. These pro-textual constituents might be values such as integers or character strings, for example, or our identifiers from D. Alternative organizations of structured text might stipulate some other organization instead of sequences for the constituents, perhaps assignments to labels, or might involve some restrictions on which combinations are permitted, such as adherence to particular grammars. Further refinements might involve stipulation of a notion of variable-binding (in the sense that quantifier notation is considered to be variable-binding).

Both string texts and structured texts obviously induce simple interpretations for $\mathrm{Occ}(t)$, $\mathrm{id}(t)@$ and t/f satisfying the constraints stipulated above. Further, string texts will be effectively discrete if identifiers are, and structured texts will be effectively discrete if all their pro-textual constituents are.

2.5 Structured Text and Abstract Syntax

Although our proposals are independent of whether text is structured or string text, experience with abstract structured text informs our expectations for exploiting and managing abstract identifiers. Indeed, making identifiers abstract can be seen as the completion of a movement away from concrete syntax, which in the Nuprl research program began with the adoption of pure structured texts rather than string text as the fundamental data structure, and which may be seen as well in the more recent widespread adoption of XML on the worldwide web.

In the Nuprl4 systems [12], although all pro-textual constituents are concrete values, the texts are deemed otherwise independent of any string representation. Indeed Nuprl4 was in use long before any conventions for standard string representations were specified (for communication with external processes). How texts are presented for viewing or editing by a user is determined by how the user's session is configured, especially what collections of "display forms" have been activated. These independently loadable display forms can be altered by the user at will, permitting different users to use their preferred notations for the underlying structured texts, or letting a developer proceed with content development without first committing to notations. Further, it is possible to manage notational ambiguity when the browsing or editing tools directly access the underlying unambiguous structured text, and support incremental disambiguation on demand.

The purpose of making content independent of notation was to encourage the use of formal text free of notational vagaries and disputes, simultaneously acknowledging the profound cognitive significance of concrete notations to users and their irrelevance to ordinary formal issues. Concrete notation may be so important to the user that it should not be needlessly dictated by the different needs of others, including authors of formal texts.

The use of structured texts not only liberated the development of formal material from many notational concerns, but actually simplified the programs used to analyze and modify texts both for formal purposes, such as proof verification, and also for utilities for presenting, editing and managing texts. While the simplification of formal textual analysis by programs is dependent on the fact that the text is structured, and therefore cannot be expected to provide models for runtime manipulation of atomic identifiers, the abstractness of textual structure does give us some insight into how one lives concretely with abstract features of syntax.

Let us use "operator" loosely to mean a structured text with some unspecified constituents that are filled in to form instances of the operator; the constituent identifiers held constant across operator instances may be thought of as identifying the operator.

Here are a few practical observations gleaned from experience using abstractly structured text, albeit with concrete identifiers. First, since the user's access to the structured texts is mediated by an interface that realizes (user alterable) notations for the abstract text structures, the user ordinarily does not need to know all the details of the underlying text structure, such as what order the various constituents really occur in, which may differ from the order in which they are displayed. More pertinently, although the meanings of abstractly structured texts are keyed off of constituent identifiers, like "all" or "implies", exactly which concrete identifiers are so used can typically be ignored by the user because the display of instances typically uses some other suggestive notation not involving those identifiers. For example, the text displayed as "$(\exists x{:}\mathbb{Z}.\ x = x) \Rightarrow P$" in the standard Nuprl system setup is actually a complex including the concrete identifiers "implies", "exists", "equal", "int" and "variable", none of which is apparent from the notation or relevant to the understanding or editing of the underlying structured text.

These identifiers do not need to be known to the user because instances of the operators of interest can be used directly; for example, one can find the definition of an operator (or that it is an undefined primitive) directly from a textual instance unmediated, as far as the user need know, by the constituent identifiers. When the user wants to create an instance of an operator the identifier need not be known because it can be built by cut-and-paste from another text, or by clicking in a menu, or by the user entering a command which is bound to the operator. And just as notations for structured text can be controlled by the user, so can these command-bindings, which are similarly independent of the constituent identifiers; for example the command bindings that happened to be used when creating the example text just above were "some", "imp" and "=", rather then "exists", "implies" and "equal".

Further, since program source texts in Nuprl are themselves structured, other text can be included in literals (in quotation) in the source program, and so even the programmer need not be explicitly aware of what the constituent identifiers are in sample texts.

These observations suggest the possibility that the concreteness of identifiers may well be inessential to a person's practical access to structured text when mediated by an appropriate interface that realizes any concrete symbolic "bindings" to abstract text. Below we shall emphasize definite liabilities of concrete identifiers in formal texts, which ought to encourage us to exploit their formal dispensability – by dispensing with them.

2.6 The First Proposal

Use abstract identifiers in formal text. Now, how might one do that? Obviously the intent is to tightly restrict what operations may be performed upon them. If we were using a single programming language with its runtime environment we might borrow some method from it. If we were simply building texts as runtime data operated on by programs of a conventional language that had endlessly allocatable atomic values already, we'd be in business. Some languages (such as module or class based languages) have explicit methods for introducing abstract types, where the programmer can stipulate the creation and access methods. Some have abstract pointer values which would be quite appropriate for implementing abstract identifiers as data especially if the second proposal, identifying abstract identifiers with references to texts, is adopted as well. And even in programming languages without primitive notions of abstract types, programming disciplines for treating values abstractly are common. But we do not want to presuppose such a monolithic programming environment, so we need to expose the nature of abstract identifiers rather than simply taking them as a convenient, but perhaps mysterious, primitive notion.

A more analytical explanation of abstract identifiers would be to say that they are *treated* abstractly rather than somehow *being* abstract in any relevant sense. Procedures that treat identifiers abstractly as constituents of some class of data values have equivalent computational behavior for equivalent data, where equivalence of two data values consists in their being identical modulo differences as to which identifiers occur in them, but restricted to there being a one-one correspondence between "matching" identifiers. In the case of text as data the machinery introduced above is sufficient for expressing this equivalence precisely: $t \in \mathrm{Text}(D)$ and $t' \in \mathrm{Text}(D')$ are equivalent when there is a one-one function $g \in D \to D'$ such that $t' = (t/{:=}g)$.

A "text server" would be a plausible alternative to a monolithic programming environment for allowing various parties to access and share texts having abstract components. Falling back on the familiar idea (fiction?) that abstract values somehow exist only as an aspect of the process implementing them, the abstract identifiers of texts could be thought of as internal to the text server, and so cannot be communicated to external clients. Thus an essential element of establishing a connection with the text server involves agreement upon how to communicate concrete values instead of abstract identifiers occurring in texts to be served up; part of the state of a connection to a text server might be a table internal to the text server uniquely pairing a concrete value with each abstract identifier that has occurred in any of the texts served to the client since the connection was established.

If the external client wants to build and communicate to the text server a new text involving new identifiers, then since the client can build only concrete values externally which will be converted to abstract identifiers by the server for its internal use, the client has to get hold of a value which will be associated in the text server connection with a heretofore unused abstract identifier; the obvious way is to implement a text server utility for serving up a new text comprising freshly allocated abstract identifiers, which are then automatically paired by the server with concrete values to be communicated in their steads.

Since external clients are provided concrete values instead of abstract identifiers, and their programming environments are not under control of the text server, they cannot be compelled to write their programs to treat such values abstractly. However, we can encourage such abstract treatment by adopting the following policy: when a text is provided by the server, the choice of which concrete identifier is associated with which abstract identifier is made by the server and is only guaranteed to remain stable within a single connection to the text server. If two independent connections are established then they may be expected to realize different concretizations for abstract identifiers. For example, the text server might simply sequentially number from one the abstract identifiers as they are served to the client during a particular connection.

The discussion above about programming with abstract types implicitly suggests that such programming methods are run-of-the-mill and should not be considered burdensome. The more problematic issue is the cognitive fact that persons depend upon concrete values. This issue was addressed in the earlier discussion of lessons gleaned from using abstract syntax. There it was suggested that for many purposes any connections between abstract identifiers and concrete ones should be part of the user's medium of access rather than inherent in the text. Note that the text server's connection-specific association of abstract identifiers with concrete values does *not* serve this purpose as the text server is making up the association. Any user associations would themselves be stored as texts available from the server and would be taken as parameters by the browsing and editing utilities connecting to the text server. The complexity of implementing this proposal requires some benefits to justify the effort.

2.7 Radical Freedom from Name Collision

So far, our discussions of treating identifiers abstractly have been to the effect that it is feasible rather than that it is beneficial. The benefits of making syntactic *structure* of text abstract seem fairly direct since it makes texts convenient to manipulate for formal purposes as well as allowing the realization of variant presentations and removing notational disputes from the formal text to a more tractable realm. In contrast, the abstractness of identifiers makes no contribution to such notational liberty because there is no "hidden" structure in an identifier to expose (parse); and because the very devices for presenting concrete values in place of abstract identifiers could have been applied to concrete identifiers as well to translate them to other concrete identifiers that the user prefers; and because, in the case of structured texts, the methods alluded to earlier concerning notation and editing tend to make the choice of identifiers arbitrary and largely irrelevant to the user anyway.

We have built into our concept of text, Text(D), the presupposition that identifiers have "places" where they occur within texts, so we can locate and change occurrences, giving a precise sense to the notion of changing identifiers throughout a text. But fixing a (concrete) type D would leave this feature intact, so whatever benefits might accrue to us from abstract treatment of identifiers must, tautologically, derive from the fact that they can be freely replaced (one-one) without significantly altering the use of the texts.

When several formal texts, developed independently or partially independently, are to be combined, there is a serious possibility of name collision between identifiers if they are concrete. Perhaps two text developers have used the same (concrete) name for different mathematical concepts or for two different procedures (defined by different program source texts). Plainly, combination requires us to consistently rename the problematic identifiers. This problem becomes acute because the desire to combine formal texts is irresistible and because these identifiers become data operated on by programs such as proof checkers or other programs establishing formal claims, sometimes at great expense.

Given our presupposition of Text(D) with its places for identifiers, renaming occurrences within some texts to avoid "static" name collision with others is non-problematic, even if D were not treated abstractly. The problem is that program execution might depend upon these identifiers as runtime values, and thus claims established by means of executing these programs might not survive renaming of identifiers. Claims about specific texts based upon programs that treat identifiers abstractly remain intact, however, and that is the purpose of abstract identifiers.

Now let us consider the connection between program source text and identifier renaming. After the second proposal is made, viz. identifying text references with abstract identifiers, we shall reconsider how programs may be about texts, but we leave it vague at this point and emphasize other aspects. When a program is constructed for execution from one or several source texts, texts that may contain abstract identifiers both as literal data for the program to operate on and as program source operator- or name-syntax, then these source texts themselves may be subjected to identifier renaming. When this program is executed, if it treats identifiers abstractly, then any identifiers it operates on must either come from the source text for the program, or from the texts it is about, or be otherwise provided to it as part of its state. The program text itself cannot "hide" identifiers that it then creates at runtime; it cannot create identifiers "from nothing" since they are not concrete. In contrast, had identifiers been concrete, say integers or character strings, there would doubtless be sufficient computationally expressive resources in the programming language to generate values not occurring in the source text (nor in the other texts nor other runtime state).

Thus, claims about a collection of texts that are based upon execution of programs whose sources are also among those texts, remain intact if identifiers are uniformly renamed throughout those texts, so long as the programs treat identifiers abstractly and the state provided to those programs for execution also (somehow) renames all the identifiers in the same way (which would be the case if the state were derived from the text collection in a way that treated identifiers abstractly).

The second proposal shall greatly increase the scope, and amplify the effect, of this radical freedom from name collision, and form the basis of accounting for intertextual dependencies of computationally based claims about texts.

2.8 The Second Proposal: Identifying Abstract Identifiers with References to Texts

Use $D \rightarrow \mathrm{Text}(D)$ as the type of text reference, i.e. make the "content of" or "referent of" function map every identifier, whatever the class of identifiers happens to be, to texts whose identifiers are drawn from that same class. We shall call such functions "closed maps" because every identifier that can occur in a text refers to a text, i.e. there are no dangling pointers.

Closed maps then will serve as inter-referential text collections (considering the choice of domain as part of the map's identity). Claims about (inter-referential) texts will be interpreted with respect to closed maps $\langle D, f\rangle$. Claims that are abstract with regard to identifiers will be about appropriate equivalence classes of closed maps. It will be desirable to make the claims themselves come under management by recording them as texts referring to the texts they are about. Imperative programs that operate on inter-referential texts will have a closed map, call it the "current map", as part of their state and can apply this closed map for text lookup, can modify constituent text values (i.e. change the assignment of texts to identifiers), and can extend the domain to include new identifiers. Because of our esteem for formal correctness and our hope for free combination of formal texts, we shall be especially interested in claims that are abstract with regard to identifiers and are based on execution of programs operating on closed maps.

Some topics we are now bound to elaborate are closed map equivalence, operations on closed maps that are important for our purposes, why atomic pointers, and why no dangling pointers. We shall indicate equivalence between closed maps by $\langle D, f\rangle \sim \langle D', f'\rangle$ which can be characterized by

$\forall f{:}(D \rightarrow \mathrm{Text}(D)),\ f'{:}(D' \rightarrow \mathrm{Text}(D'))$.
$\langle D, f\rangle \sim \langle D', f'\rangle \Leftrightarrow (\exists g{:}(D \rightarrow D').\ g{:}D \rightarrow D' \text{ renames } f \text{ to } f'\ )$

where

$g{:}D \rightarrow D' \text{ renames } f \text{ to } f' \ \equiv_{\mathrm{def}}\ g{:}D \rightarrow D' \text{ invertible } \&\ g \text{ carries } f \text{ into } f'$

$g \text{ carries } f \text{ into } f' \ \equiv_{\mathrm{def}}\ \forall i{:}D.\ (f(i)/{:=}g) = f'(g(i)) \in \mathrm{Text}(D')$

The notion of "carrying" $f \in D \rightarrow \mathrm{Text}(D)$ into $f' \in D' \rightarrow \mathrm{Text}(D')$ is that of embedding the texts and intertextual references modulo consistent change of identifiers, but further allowing for the identification ("collapsing") of some identifiers, which can be useful independently of just renaming. Renaming is carrying without collapsing distinct identifiers by requiring the identifier translation to realize a one-to-one (i.e. invertible) correspondence between the identifier types, hence when $D = D'$ renaming is simply permuting all the identifiers.

It will be convenient to avail ourselves of a vernacular of "objects" and "pointers" in discussing closed maps. The object-vernacular works by pretending that the current closed map relation between identifiers and texts is mediated by some "objects" which have the identifiers as names or pointers and the texts as content; these are concepts we are all accustomed to using informally, and which may help to indicate the practical purposes of this proposal. This pretense allows us to think of changing the contents of an object or its name independently. Sometimes it will be convenient to identify objects with identifiers, and talk about changing contents of, deleting, creating or copying objects. Other times it will be convenient to consider the identifiers as arbitrarily assigned, and freely reassignable, object names. But always our vernacular here shall allude to facts about closed maps *per se*.

Returning to closed maps, it may help to be precise about a few concepts.

2.9 Closed Submaps and Indirect Reference.

The concept of closed map equivalence $M \sim M'$ (along with the cognate concepts of "renaming" the identifiers of a map and "carrying" one map into another) was introduced immediately with the second proposal because respecting this equivalence is the principal stance towards text collections motivating it. Here are some other useful concepts made explicit, along with some glosses in the object/pointer vernacular, pertaining to indirect reference and to closed submaps.

The concept of a text referring directly to an object is obvious but doesn't require the reference relation as an explicit parameter, since it is just the occurrence of a pointer value in the text:

$$t{:}\mathrm{Text}(D) \text{ refers directly to } j \equiv_{\mathrm{def}} \exists p{:}\mathrm{Occ}(t).\ j = \mathrm{id}(t)@(p) \in D$$

$$\mathrm{Ids}(t) = \{i{:}D \mid t{:}\mathrm{Text}(D) \text{ refers directly to } i\ \}$$

Possibly indirect reference, of course, depends on the map:

$$t{:}\mathrm{Text}(D) \text{ refers to } j \text{ via } f \equiv_{\mathrm{def}} \exists n{:}\mathbb{N}.\ t{:}\mathrm{Text}(D) \text{ reaches } j \text{ via } f \text{ within } n \text{ steps}$$

which means that the pointers are followed through the map. Composing this relation with the map itself naturally extends it to inter-object reference, for example i refers (perhaps indirectly) to j when $f(i){:}\mathrm{Text}(D)$ refers to j via f. We extend this to a reflexive relation between objects:

$$i{:}D \text{ reaches } j \text{ via } f \equiv_{\mathrm{def}} j = i \in D \vee f(i){:}\mathrm{Text}(D) \text{ refers to } j \text{ via } f$$

A (closed) submap of a closed map is simply a restriction of the identifier class, *given* that the map remains closed. Here is a characterization of the closed submap relation $M \subseteq M'$.

$$\forall f{:}(D \to \mathrm{Text}(D)),\ f'{:}(D' \to \mathrm{Text}(D')).$$
$$\langle D, f\rangle \subseteq \langle D', f'\rangle \Leftrightarrow (D \subseteq D')\ \&\ (\forall i{:}D.\ (\mathrm{Ids}(f'(i)) \subseteq D)\ \&\ f(i) = f'(i) \in \mathrm{Text}(D))$$

Note that there is no renaming involved in this simple notion of submap. Now let us consider some important ways of finding submaps. Although we cannot restrict a closed map to just any subclass X of its identifiers, we can restrict it to the smallest submap comprising X, which we shall call "contracting around" X:

$$\forall f{:}(D \to \mathrm{Text}(D)).$$
$$(X \subseteq D) \Rightarrow \text{Contract } \langle D, f\rangle \text{ around } X = \langle \{j{:}D \mid \exists i{:}X.\ i{:}D \text{ reaches } j \text{ via } f\ \}, f\rangle$$

This fundamental operation allows us to select "coherent" subparts of a closed map by stipulating what part we are primarily interested in and retaining everything that part depends on. A cognate operation is "focusing" on the part relevant to identifier subclass X, which is contracting around all the identifiers that can reach X:

$\forall f{:}(D \to \mathrm{Text}(D))$.
$(X \subseteq D) \Rightarrow$ Focus $\langle D, f\rangle$ on X = Contract $\langle D, f\rangle$ around $\{i{:}D \mid \exists y{:}X.\ i{:}D$ reaches y via f $\}$

In the other direction, to discard several objects X requires discarding all that point to them as well, i.e. contracting around the type of identifiers neither in X nor pointing even indirectly to any of X:

$\forall f{:}(D \to \mathrm{Text}(D))$.
$(X \subseteq D) \Rightarrow$
Delete X from $\langle D, f\rangle$ = Contract $\langle D, f\rangle$ around $\{i{:}D \mid \neg(\exists y{:}X.\ i{:}D$ reaches y via $f)$ $\}$

The significance of a submap is not only that it is a natural unit of coherency for an object collection, but that when claims are "localized" in a sense to be discussed below they are about texts only of a particular submap, and if the localized claim is recorded as a text in the map itself, then any submap or supermap containing that record preserves the claim. This is a matter to which we shall return after discussing the acceptability of *identifying* pointers with abstract identifiers in the second proposal.

2.10 Acceptability of the Identification

The Second Proposal (section 2.8), to identify abstract identifiers with pointers, is not obviously acceptable. Once one accepts the first proposal and admits the occurrence of abstract values in texts, it is a small step to permit the occurrence of object reference values as well, thus the issue of making every object reference an abstract identifier becomes that of making them atomic and abstract in the first place and that of collapsing what could be two independent abstract name spaces into one. The other direction of the identification engenders the question of why every abstract identifier should point to a text object, this being especially acute in light of the stipulation that every pointer-value points, as expressed in the formulation of the reference map being assigned the type $D \to \mathrm{Text}(D)$, which is a type of *total* functions.

Let us first justify this "no dangling pointer" characterization of closed maps. Observe first that this is not a substantial property of maps since there is an obvious translation between closed maps and maps whose domains may omit some identifiers - simply distinguish some sequence of $\mathrm{EMPTY}(k) \in \mathrm{Text}(D)$ for $k \in \mathbb{N}$, then for any function F from a (decidable) subclass of D, say D', into $\mathrm{Text}(D)$ define $G(x)$ as

EMPTY(0)	if $x \in D$ is not in D',
EMPTY(k+1)	if $x \in D'$ and $F(x)$ is EMPTY(k), and
$F(x)$	if $x \in D'$ and $F(x)$ is not one of the EMPTY(?).

Then $G \in D \to \mathrm{Text}(D)$ and F can be easily recovered from it.Thus we *could* use possibly non-closed maps as our basic type and invoke this one-one correspondence if desired. So we base our formulation on closed maps as a convenience; it enables a certain simplicity of explanation. As is easily foreseen, we aim especially at supporting the certification and recording of claims about texts that survive addition of new texts and alteration or deletion of texts upon which they do not apparently depend. The operations of monotonic extension of closed maps and of taking closed submaps leave intact the reference of any pointers used by these privileged kinds of claims, whereas taking a non-closed submap can amount to changing the content of "deleted" texts and making a monotonic extension of a non-closed map can amount to changing the texts associated with dangling pointers, making these operations on non-closed maps of little significance to us. So the *convenience* of exploiting the familiar concepts of submap and map extension for the purposes of these proposals depends on their restriction to closed maps.

In the worst case, the policy of making abstract identifiers refer to texts might lead to assigning "empty" texts to some abstract identifiers, but realistically there are more useful texts one might assign to identifiers not created principally as pointers. When an identifier is introduced to serve as a new operator in mathematical text (as either defined or primitive) it seems most natural to associate with this identifier the text that constitutes its definition or its declaration as a primitive. Then as recommended in Informal Practices (section 2.1) this definition or declaration is automatically referenced at any textual occurrence of the operator. Similarly, an identifier used in program source text, such as is meant stand for a global procedure or data type, could be made to refer to the source text defining the value to be bound to that global at runtime. If one introduced an abstract identifier for a special, perhaps informal, purpose then it would be natural to associate an informal remark as to this purpose.

Returning to the other direction of the identification, the principal benefit of making pointers abstract and atomic is that it enables one to formulate a definite criterion for when one object refers to another which serves as a basic criterion for dependency between objects. As was pointed out in Runtime References (section 2.2), the use of abstract atomic reference values at runtime is a familiar practice, but there are other benefits of concrete and complex identifiers which we must avoid losing in the move to atomic abstract ones. The issues surrounding the move from concrete to abstract pointer values are the same as those for identifiers addressed above. However references to objects often have a complex structure that serves to indicate some important connection between the objects that is "external" to the objects, that is, a relation between the objects that is not to be inferred simply from which objects refer to which. For example, one common scenario when browsing the web is the attempt to find some context for a page one jumped to, via a search engine perhaps, when the page itself is deficient in this regard - a rather successful maneuver is to follow the URL gotten by chopping the last leg off the current URL. Sometimes theorems, definitions, lemmas and axioms developed in computerized proof systems are organized into "theories" that exploit theory-relative addressing in inter-object references; similarly collections of program texts are often arranged into "modules" (the MetaPRL system under development at Cal Tech does both [8, 7]).

Under the proposals offered here, where these inter-object references are instead atomic, if such multi-textual structure is desired it must be (and can be) represented by explicit textual description that effects the relations between the objects. This is analogous to externally establishing connections between abstract and concrete identifiers or between abstract syntax and concrete presentation, and has corresponding benefits. If a formal claim can be established based upon the intertextual references without recourse to the module or theory structure then those objects can be "appropriated" from those original organizations imposed upon them; indeed other organizations could then be built around these objects. This possibility of co-existing alternative organizations of overlapping object collections can be effected both for formal purposes and for informal explanatory purposes, probably involving the addition of informal explanatory texts referring to formal objects, giving rise to different "readings" of formal text collections directed at different audiences.

The only issue raised at the beginning of this section as yet unaddressed is the "collapse" of what could reasonably be made independent abstract value spaces into a single space; after all one might choose to distinguish abstract pointer values from abstract non-pointer identifiers (albeit losing the ability to have a "defined" identifier point to its defining text). There is a simplicity of formulation that follows from having the single abstract value space, but might we be losing something? I think not, because practically the chances for interference do not appear likely, but even theoretically the implementation of independent abstract value spaces could be easily effected by reduction to the one space as follows; each new space would be identified by a newly allocated identifier from the basic space, and paired with values from the basic space, thus allocating a new value in one of the new spaces is simply allocating a value from the basic space and pairing it with the appropriate already allocated space-identifying value.

3 Keeping Records of Fact

The rest of this article concerns the formalization of recording facts, which motivates the first two proposals for managing data in a manner that will support record keeping.

3.1 Certification Systems

The notion of "formal" explained in the Introduction (section 1) plays a heavy part, the part of an anchor, in these proposals; and it is now time to drop that anchor. The potential for maintaining various "claims" about various inter-referential text collections has been raised in earlier sections to motivate our proposals for structuring text; for example, it was suggested that the concept of a closed submap gets part of its significance that of a "localized" claim, a localized claim being, with respect to a closed map, one that depends only upon the values assigned to identifiers reachable from the objects explicitly mentioned in it. This rather vague statement gels rather suddenly when

> (1) the content of a claim is expressed as a text (2) whose interpretation consists of there being a computationally established fact, (3) such computation being executed by a prescribed method (4) that is implicitly parameterized by the current closed map, (5) this text being made the content of an object distinguished as a record, a "certificate", of that computationally established fact.

The phrases of this suggestion are enumerated as a convenient indication of its aspects. We shall return to the matter of localized claims and submaps after considering this suggestion of how to treat, indeed formalize, claims about texts.

Let us use the term "certification system" in reference to systems that realize and record claims in the manner suggested. Here we shall describe a simple certification system as a concrete paradigm in order to elucidate the concept and its complications; the system presented here will be too simple to adopt practically, for reasons to be explained, but should be adequate to indicate the potential of our three proposals. We shall suppose that this certification system is to be built into a text-server of the kind described above in the First Proposal (section 2.6).

A computerized system is *competent* to establish a claim only by effecting a computation, which is also the ideal for establishing *formal* claims. If the intended clients of the certification system were interested only in whether purported proofs, say, were in accord with the rules of some particular fixed logic, then one might simply prescribe computations that check purported proof texts for conformance to those rules. If a certification system were to be designed solely to verify when texts were created and by which clients (for purposes of accountability or priority), then reliable methods for determining dates and client identity would be sufficient computational means for establishing claims and would be integrated into the object creation methods of the text-server. But more generally, we would anticipate radically greater diversity of claims, by clients who have not agreed to common logics, or even to what one normally thinks of as a logic at all. Diverse parties *can* come to a common understanding of ways to execute programs.

We shall suppose our paradigm certification system to include a general purpose programming language, and suppose it to support claims of the textual form Certified($Prog$; $Result$) meaning that some execution of program text $Prog$ had the text $Result$ as its principal result value (further side effects are possible). We say "some" execution to allow for both the possibility that execution is nondeterministic and the possibility that execution depends upon some data, such as a generic runtime state, that cannot be determined from the program text or the other texts it refers to.

The current closed map will be supplied as a parameter to the execution of program texts and will serve as the mechanism for reference to text objects. There is a distinction to be made as to how the current closed map enters into computations. While we consider the domain of a closed map to be part of the map, it will be of some importance whether the domain is made available for computation (as a list for example) as opposed to the map being provided merely as a content lookup mechanism. One might contrast the use of the current closed map (as a whole) as an ontic value that a program can mention versus its use simply as a semantic device for referring to text objects via pointers one has in hand. Sometimes this distinction of semantic roles of a data value is expressed in the jargon of computing by applying or withholding the epithet “first class”. Let us designate programs that mention the current closed map as being “about the current closed map”, and programs for which the current closed map is merely a semantic device as being “about texts individually”. Execution of the latter under a given closed map will have the same results as if performed under any supermap of the submap reachable from the program text together with any abstract identifiers supplied further in the runtime state - almost. The result can only be guaranteed to be an *equivalent* closed map, and even that only under the assumption that the program treats identifiers abstractly; after all if the program execution creates new objects those new identifier values are not likely to be determined by the reachable submap.

Here we have briefly treated certification system aspects 1 through 4, which might be considered semantic, and we shall elaborate them a bit more before discussing aspect 5, which is a matter of assertoric force.

3.2 Certificate Semantics - external significance and openness

We continue the discussion of the semantical aspects of certificates and their exploitation. The “internal” significance of certificates, the significance that a certification system implementor must guarantee, is stipulated simply in terms of program execution but, of course, persons will attach some “external” significance to such program execution, and different persons may attach different significance to the same computational fact. For example, a program may be deemed by some to recognize certain forms of independently prescribed formal inferences for which the program was designed, whereas others may consider the program erroneous in that regard; and among those persons who agree as to which inferences a program realizes there may yet be disagreement as to whether those inferences are valid, and hence as to whether the certificates based upon that program can be attributed the significance of actually verifying the claims expressed in the logic. Plainly there is not just one external significance that may be attributed to a computational fact.

Different communities of clients sharing a large text collection may be expected to attend to different, although hopefully overlapping, sets of certificates distinguished by the programs those certificates employ. Such programs can typically be expected to formalize some externally interesting concepts and many certificates about those concepts will share a common program component. If, for example, $P(x)$ is a program that is widely understood to check whether texts are proofs according to some consistent logic, say, or is widely accepted as a user identification protocol, then there will be many certificates of form $\mathrm{Certified}(P(z); r)$ for various z and r. Clients who don't accept those claims about program P will probably be less interested in certificates of this form, and will instead invest their trust and attention in certificates based upon other programs. To summarize, the generic nature of the certification system simply provides a computational framework upon which externally significant certification methods are built and which are then employed by a potentially diverse community of clients, typically embodying their needs for certification in methods that are then used repeatedly to establish claims about texts. Note that it would be impractical for clients to constantly examine complex programs occurring in certificates, which means that they can be expected to develop compact easily recognizable programs whose properties they can become familiar with, and which can be easily recognized by utilities for browsing texts, including certificates. Compactness of certificate programs is achieved, naturally, by use of indirection with pointers.

That certification is based upon prescribed methods of computation inherent to the certification system should *not* be regarded as a requirement for some grand universally attractive programming language in which all clients must directly encode their formal concepts about texts. On the contrary, it is expected that the programs upon which certification is based can establish communication with, or create, external processes which can then execute programs according to their own methods; the certification system simply provides program texts to those external processes and records the results. The external significance of such a certificate is then largely understood in terms of the method of external program execution which was chosen presumably as one more suitable, in terms of cost or intellectual clarity perhaps, than would be execution directly by the certification system. The cost of going external, of course, includes any doubts one may have about the integrity of external communication and establishment of processes beyond direct control of the certification system; but those doubts are *accounted for* by the fact that any dependency upon external execution is explicit in the program of the certificate.

There is another dimension in which the certification system is open to new programs, namely the fact that a new kind of certificate can be added if need be, which requires extending the implementation of the certification system. In order to maintain clients' trust that whatever significance they attach to already-created certificates will not be undermined by supposed improvements to the certification system, it is essential that once a kind of certificate is implemented then it is not altered; thus extensions (improvements) to certificate semantics must be made by adding new kinds of certificates to be interpreted. This is a situation appropriate for employing abstract identifiers in service of the certification system itself. Suppose that the textual structure of our paradigm form Certified($Prog$; $Result$) is that it is text with two places for the immediately constituent texts, and one place for an abstract identifier which has been concretized here as "Certified"; that this abstract identifier identifies certificate expressions will be inherent to the certification system. Then to extend the certification system just pick a new abstract identifier and include it as well in a list of certificate identifiers for the system; because it is abstract no objects already existing among the texts can be mistaken for these new certificates.

3.3 Certificate Programs as Texts - locality and stability

Certificate texts themselves may be quite small even if they produce large effects by means of large programs because the certification is always done in the context of the current closed map. The program text to be recorded in the certificate may refer indirectly to many texts for both program source text and data; the result text recorded in the certificate can refer to other objects including ones newly created by executing the certificate program. (For those readers who might be concerned that we have somehow built into our designs the assumption that programs are to be executed purely by direct "interpretation" of the source texts, let it be noted that by including "bit file contents" as pro-textual values it becomes possible to build certificates that work by compiling source texts and recording the connection between the source texts and the resulting executables or load modules, thus making it possible to account for execution of source texts via their compilations. The VESTA system is one which maintains such connections to source texts explicitly [13].)

A very interesting consequence of the fact that certificates can refer to other texts is that they may refer to *other certificates*. One can design certificate programs that depend on the existence and content of other certificates, and when this dependence is well-founded this inter-certificate structure can be viewed as a rather generic proof structure, each certificate representing an inference from prior certified facts. This formal structure could be employed to realize proofs of a more conventional logic by associating a certificate with each subproof, or it could be used, as was implicit in the introductory remarks (section 3.1) on certification systems, as the formal face of a non-computer based notion of proof involving recording human attestations to various facts that may refer to other certificates as well as to ordinary texts.

Let us now consider an especially useful pair of concepts derived from the semantics of program execution, namely "localized" and "stable" code. The intent is first to distinguish program text having a *fixed* meaning dependent only upon the texts reachable from the program text itself. Let us call program text "localized" if (1) it treats identifiers abstractly, and (2) it is "about texts individually" in the sense explained above in Certification Systems (section 3.1), and (3) it gets no data from the runtime state that is not derivable from its own text and the texts it points to, and, (4) by leaving texts it depends upon intact, it avoids undermining its own meaning. We further distinguish localized programs that fully determine the result of execution (modulo closed map equivalence), which we shall designate "stable"; i.e. stability is locality plus a weak determinism. We shall not precisely define these concepts here since they are not *internally* significant to the certification system; there may be various methods implemented as certificate programs for recognizing usefully large classes of localized or stable program text, although no program can recognize exactly either the stable or the localized programs. Let us say a *certificate* is localized or stable according to whether its program text is localized or stable.

The locality of a certificate gives the certificate persistent significance since examination of the program text (perhaps following text pointers) tells one what was done to establish the certificate; in contrast if the certificate program depended upon aspects of state not determined by the current closed map then there might be little real external significance to the certificate. For example, suppose one were to write an inference-verifying program for some logic and were to use it to verify all the steps of a proof represented as texts. If the inference checking code were simply supplied temporarily via the runtime state, bound to some distinguished name, say, then once that state was gone there would remain no evidence that *that* program had been used to verify the proof rather than some other program *not* representing the intended logic. Certification according to non-localized code just doesn't count for much in the end. But that's not to say it isn't useful in the beginning and the middle.

There is a tension at this point: non-localized code does not "sustain" persistently meaningful claims, but localized code is often too inflexible for convenient and heuristic development. For example, suppose that as part of a proof development toolkit one implements programs to search the entirety of the current closed map for suitable lemmas when trying to certify proofs, perhaps modified by various user-settable parameters. Hence such programs are about the current map as a whole and not just about texts individually, and they may depend on data not derivable from the texts reachable from the program text. Later these parameters will be different and, *ex hypothesi*, this difference will not be reflected in the program text; the certificates based on this program text rapidly lose currency, becoming merely fragmentary historical records of underspecified facts. This tension entails the need for utilities that assist in the conversion of non-localized to localized certificate programs; we do not pursue this here, but suggest that in the case of a search based heuristic it is plausible, after the search returns its results, to replace program code that invokes the search by program code that refers to those results rather than finding them again, or at least by search code that compactly refers to a smaller search space that one is willing to retain permanently.

A stable certificate, in addition to having persistent meaning due to its locality, has the virtue that it can be checked for correctness by simply re-executing it since its locality determines how to execute it, and the uniqueness of correct result means we can confirm the result by reproducing it or, assuming it runs to completion, refute it by finding a different result. Thus, one value of stable certificates is that they can be *independently* rechecked either as a matter of course or should particular doubts about the correctness of the certification system arise.

Before continuing discussion of closed map operations and certificates, we consider the assertoric "force" of certificates.

3.4 Certificate Pragmatics - records and assertoric force

In ordinary language there is a distinction to be made between the semantic content of a proposition and the pragmatic significance of the act of asserting it. The same proposition, be it true or not, can appear as part of many larger propositions such as conditionals or disjunctions, making a uniform semantic contribution to those larger propositions. But it is by the act of assertion that one declares its truth, that one commits to its being true. See Dummett's chapter on Assertion in his "Frege: philosophy of language" [3] for extensive discussion. An apt analogy from Dummett is the gap between knowing all the possible moves of chess and knowing what counts as winning; or between knowing what positions are called "winning" and understanding that winning is what a player is supposed to aim at. We have been discussing the semantics of certificates in a way that alludes to an assertoric practice without making any explicit provision for representing assertion in the certification system. The fifth aspect of certificates mentioned in Certification Systems (section 3.1) is meant to do this; certificates are distinguished from non-certificates and their special significance derives from the fact that a certificate object is created only when its content has been established as true, e.g. in our paradigm system only after its program text has been executed and has returned the specified result text.

While it would be natural to use some indication external to the contents of text objects to distinguish certificates from non-certificates, it is convenient to constrain the certification system so that no non-certificate can be created whose content has the form of a certificate, and so simply distinguish certificates by textual form; we shall assume this method for our paradigm certification system. The existence of a certificate is not simply a suggestion or idea that a certain program was executed with a certain result, but actually constitutes a *record* of that fact. The sense of "record" used here is *not* that sometimes used in the computing community simply to mean some structure of several data items treated as a unit, but rather we use the more ordinary sense, which is also that used in records management and archival science. The content of a record is not essentially what makes it a record - a record is an attestation of some event or fact described by the content of the record. The special concern of the records keeper or archivist is to maintain the evidentiary significance and integrity of records, recognizing in particular the necessity of maintaining groups of interrelated, even interdependent, records (see [11, 4] for elaboration). The traditional archives receives the records of an organization that are no longer current, although there seems to be some expectation nowadays that archival science will move towards records management more generally (see [10, 18] concerning archives and their future). The fact that digital records often depend upon their use as data originally interpreted by particular hardware has the peculiar implication that integrity of records may further require the incorporation of some museum functions (see Rayward [15] on this topic).

A certificate to the effect that some program text was executed with some result would, of course, be of little value if the texts that it depends upon have been arbitrarily changed, or if the manner of executing the certificate program is not fixed. This would be most obvious in the case that both the program text and the result text of the certificate consisted simply of pointers to more substantive operative texts which got altered. It would be as if a person simply made up new words or special meanings for words then recorded facts based on them, but then kept no record of that special nomenclature. Some provision for stability in the meanings of certificates is necessary for a useful certification system, and for our paradigm system we stipulate (temporarily) that no certificate and no object referred to (even indirectly) by a certificate object can be altered, and that the method for executing programs for any existing certificate shall not be changed; changing an object forces deletion of dependent certificates. Thus, if the certificate is localized in the sense given above in Certificate Locality (section 3.3), then the basis for the certificate (excepting vagaries of external program execution) remains completely intact so long as the certificate exists.

In a certification system in which individuals and groups work to develop complex bodies of texts including certificates, the impossibility of altering texts referred to by certificates is not likely to be practically tolerable. For example, inserting formally irrelevant comments into text need not ruin certificates. Or more significantly, changing the proof of a theorem need not ruin the inferences citing it as a lemma, so long as the inferential basis (rules and axioms) upon which it depends is not extended. Part of designing a practical text certification system for development of texts, therefore, will entail designing methods whereby texts can be altered without necessarily removing certificates that depend upon them. A general method would be to associate with a certificate not only the code for creating it, but also code for "reconsidering" the certificate with the possibility of modifying it or leaving it intact. Thus, the significance of certificates would be more complex than in our paradigm system, but would still be explicit; protocols for reconsidering certificates would need designing.

More globally, a certification system must provide some sense of identity (modulo equivalence) of closed maps across sessions, such as keeping a "last" current closed map for each client which it presents (modulo equivalence) to the client at the next session. Concomitantly, in order to enable communication of texts between clients there must be some methods provided for allowing clients to provide, at least temporarily, external names which can be communicated outside the certification system and with which parts of their current closed maps may be retrieved by other clients.

These pragmatic systemic constraints are needed in addition to locality of certificates to give certificates the possibility of serving as usefully meaningful records, but the certification system itself is not competent to ascertain generally when a certificate is localized. Locality must be attributed "externally", perhaps by establishing other certificates to that effect.

3.5 Establishing Abstractness, Locality and Stability

It would be plausible to build into the certification system the abstract treatment of identifiers by the operations directly performed by the system itself but, as mentioned in the First Proposal (section 2.6), it is not plausible for abstract treatment by external program execution to be enforced by the certification system. The reason given there was that the interface between the text-server and the external client must be mediated by concrete values (temporarily associated with text objects); but further, it is only the ways of communicating with or creating external processes that are explained as part of internal program execution, and so it would be quite hopeless to attempt designing a certification system that depended upon effective tests for whether externally executed programs treat identifiers abstractly.

Similarly, enforcing locality and stability (concepts introduced in Certificate Locality (section 3.3) above) of external programs is beyond the reach of the certification system, but even worse, a practical certification system will not be restricted even internally to localized programs since the non-localized programs are too important as utilities and heuristics. There is also a valuable role for non-stable localized programs, namely in the coordinated use of multiple processes, an attractive example of which would be letting multiple proof engines race for a proof of a given goal then, after the first finishes, killing the others.

Determining generally that a program has one of these properties requires understanding how the program is executed, and is not effectively recognizable. There are basically two methods for establishing such a property. The first is *ad hoc* determination that a particular program has the desired property, or more usefully that any application of a common program to arbitrary arguments has it; formal proof systems might sometimes be incorporated to argue for such claims about a program. The second method is to specify a program that can be applied to program code in order to recognize usefully large classes of programs with the feature, these property-establishing programs being incorporated into certificates; again formal proof systems might sometimes be invoked as part of an automated test for the desired property. There can of course be disputes among clients about whether specific tests are correct.

Let's be a bit more concrete in order to make the intention here clear. A coarse partial test for locality might go *something* like this, presupposing a partial test for abstract treatment of identifiers. All the global names such as names for procedures or other constants occurring in a program text must refer to source texts that can be used to bind the name to a value for execution, and all those source texts must themselves satisfy our locality test (recursively); while code that updates globally assignable state variables is acceptable, code that reads such variables is not; while the operation of looking up a text content is acceptable, there must be no use of an operation for listing the whole domain, say, of the current closed map; and as with assignable state variables, any object whose content can be looked up in the process of executing the program must be writable by only that process; the program must also satisfy the presupposed test for abstract treatment of identifiers. Design of such tests is external to the design of the certification system, but the possibility is essential to the intended use of the system, even though, as with other matters of external significance, clients may disagree over whether particular such tests succeed.

Locality of certificates is conceived here as something we rely upon to give our records value as such, to give them a persistent meaning.[2] Hence exploiting locality would be hard to explain other than as how generally one exploits (externally) a reliable record keeping system. But there are interesting scenarios involving operations on closed maps for showing how one could exploit abstract treatment of identifiers and stability of certificates.

3.6 Exploiting Identifier Abstractness and Certificate Stability

The contribution of identifier abstractness and atomicity to establishing claims is to preclude the possibility of "hiding" identifiers in the execution by calculating concrete values "from nothing" or composing complex identifiers from simpler components. But there is a more obvious practical value discussed in Name Collision (section 2.7), namely the ability to freely combine texts without danger of accidental or intentional interference from name collisions.

Two closed maps can be easily united by uniformly renaming their identifiers to avoid collision; then localized certificates, by treating identifiers abstractly, retain their significance. The new larger combination of texts can then be further extended with texts that refer to text objects from both the original groups. More interestingly, should it happen that the two original closed maps have a nontrivial common closed submap (modulo renaming), then the two original closed maps can be joined along the common submap, each original closed map being equivalent to a submap of the combination; again all the localized claims will retain their significance.

Where might these maps we imagine combining come from? One probability is that multiple clients sharing a text-server have started from the same text collection and independently developed supermaps which they then later want to recombine. Another likely situation is that these developments were independent because they were developed by downloading the initial closed map onto different independent certification systems, implemented on laptops say, and so could not have shared a common text server for a while during "offline" development. We will briefly consider the multiplicity of certification systems later.

Our paradigm certification system was supposed to prohibit modification of certificates or any texts they (perhaps indirectly) refer to. That choice was made in order to emphasize the possibility of using certificate objects as reliable records, but in practice this precludes a lot of interesting and useful

[2] Abstractness of identifiers contributes to locality by eliminating one potential source of object references "hidden" in the execution of the program.

possibilities, so we shall now drop that assumption with the caveat that a protocol for "reconsidering" certificates, possibly modifying them or leaving them intact, would be a practical necessity which we shall here leave undeveloped. Certificates then still serve as records but with a more complex semantics involving not only how they are created, but also how they are reconsidered.

The purpose of reconsidering certificates is to allow modification of texts without wholesale destruction of hard-won knowledge, and involves accounting for dependencies in various ways. In such a system clients will attempt modifications to some texts and the effects on claims will ripple out through inter-text dependencies, sometimes leaving the claims essentially unchanged, sometimes altering or deleting them. Such a system encourages experimental alterations within text collections, such as changing a definition upon which other definitions and proofs depend. Such an experiment can be safely performed by making new copies, let's call these "clones", of all the objects one intends to change as well as clones of all the objects (or at least all we care about) that refer to them (even indirectly), but changing the clones to themselves point at clones instead of any originals that got cloned. That is, supposing A' and B' are clones of A and B, and C has no clone, then if A points to B and C, then A' points to B' and C. The experiment continues by modifying the contents of the clones and watching the effect ripple through the clones. If the result is satisfactory, the originals can be deleted if they are obsolete, or more interestingly, both branches can be retained and both referred to in combination by new objects. That is, the branches can be grown back together by adding one or more new objects that refer to objects in both branches.

The cloning operation, leaving localized certificates intact, is an exploitation of identifier abstractness. The experimental updating provides an opportunity for exploiting certificate stability. As stipulated above (section 3.3), *stable* code is not only localized but also fully determines its result. We understand how to execute the certificate because of its locality, since we can find all the code and data necessary for its execution by following object pointers, and the uniqueness of its result makes it confirmable since we can reproduce the result by re-executing it. This gives stable certificates a certain epistemic value beyond that of merely localized, perhaps non-deterministic, code, since they can be rechecked independently of the particular certification system either routinely or in case some particular doubt arises. But it is also possible to exploit stability for more heuristic purposes.

Let us take the reliability of the certification system for granted now. Certificate locality alone is enough to allow us to develop alternative branches without destroying established knowledge, which is certainly essential to the management of massive formal text collections, but the danger of losing established records is not the only obstacle to the successful management of formal texts in development. Since computations are typically vast compositional chains of subcomputations that branch on intermediate results, stable code is far more likely than is non-stable code to produce a useful result when re-executed later under slightly different closed maps. This is vital to the incremental development of texts under the "experimental" scenarios suggested above. It quite often happens that in the development of a large proof as a collection of texts, one encounters a dead end in a line of development and needs to alter some component, such as a definition, a lemma, a theorem formulation, or even some inference generating code, upon which numerous component texts have come to depend. After the alteration is effected and ripples out to the certificates, the losses will typically be less extensive if the certificates are built from stable program code since more subcomputations are left intact. Further, not only can stable code be re-executed in case of doubt, but it can be modified to expose intermediate data and so further explain the basis for the original result; for example, if some formal inference were "justified" by running code that (internally) decomposed that inference into many simpler intermediate inferences, it might become desirable on some later occasion to expose those temporarily computed inferences, perhaps because the large derived inference step is not plausible to a reader of the proof, who then desires more detailed explanation.

With these ideas in mind, it is convenient at this point to digress for a moment to consider an operation similar to the merging of closed maps described above, but one that should force reconsideration of certificates, which we shall call “folding” a closed map. Joining two maps requires specifying a one-one correspondence between some objects of those maps, then making sure that their contents match, considering the correspondence. Similarly one could “collapse” identifiers in a single closed map by specifying a set of identifiers to be *identified* with each other, and making sure that their contents match modulo the proposed collapse (actually one might specify several such classes each of which is to be collapsed). The resulting closed map will be smaller than the original and the result will not typically be a submap since there is more than renaming going on here. In a logical system such a collapse might be motivated by the observation that two formal concepts, either primitive or defined, are used identically. However, the possibility of identifying primitives is a dangerous prospect. Imagine that someone decided to identify universal and existential quantification for example; bizarrely, the same rules would then be applicable to both quantifiers, revealing that distinctness of primitives may be an important requirement for the correctness of some formal inferences. The various errors that can arise from aliasing symbols in programs are analogous. These considerations tell us that folding a closed map cannot be expected to preserve certificate significance, and so this operation must engender reconsideration.

3.7 Multiple Certification Systems

There are good reasons for implementing multiple certification systems that are similar enough to allow passing texts among them. The main formal requirement is that a certificate from one system cannot be adopted *as such* by another system. The best one can do is either to (1) “borrow” a foreign certificate by creating a native certificate attesting to the fact that text purporting to be a certificate was imported from a particular source, or (2) try to build a genuine native certificate by executing the program text of the foreign certificate; these strategies can co-exist. Modifying two certification systems to directly accept each others’ certificates would produce, not two cooperating certification systems but rather, at best, a single more complex certification system. Each certification system is a locus of trust (or perhaps doubt), and borrowed certificates should be accounted for, both for the epistemic purpose of assessing the trustworthiness of hybrid webs of certificates from disparate sources, and for the heuristic purpose of attempting local rehabilitation of claims from a foreign source that has fallen under doubt.

We do not say how certification systems should be implemented. Perhaps digital library scientists would succeed at an implementation distributed over the web, and find a way to make assurances of the sort we could expect from persons or institutions that would implement certification systems on hardware under their own control. As explained above in External Significance (section 3.2), we don’t expect control of client hardware or communication hardware (that would be absurd), because we implicitly account for dependencies on external communications within certificate code, which makes it an external issue. It is what is not accounted for that must be trusted.

Basic motives for creating a particular certification system instead of relying on an existing one can be expected to include: (1) Accessibility - one may simply be incapable or undesiring of access to an otherwise suitable certification system. Maybe access is unreliable. Maybe one wants to develop formal texts in isolation on a laptop for a while. Or maybe one must work in a more secure computational environment than is already provided. (2) Integrity - perhaps one has doubts about the integrity of an available certification system either because of particular doubts about it or because of non-specific doubts about the competency of the implementors or the security of their infrastructure. (3) Independence - the existence of independent compatible systems is a benefit to the entire community of clients because it provides some antidote to loss of accessibility or integrity of individual systems; the threat of dependency fosters a legitimate reluctance to invest in a commons of certification systems. Further, it is vital that the medium as a whole be beyond the control of interested parties; indeed, the evidentiary value, however scant, of mere publication derives from this kind of independence from authors and critics. (4) Limited Mission - a provider may decide to focus its effort on certain limited aspects of certification systems in order to guarantee integrity, or to control its limited resources. One organization might focus on library and archival functions, providing little direct support for text development, and thus relying on the existence of other development oriented systems for uploads. Another might emphasize development of formal texts by a small community, and therefore, although it has the functionality to implement archives, it refuses to commit to the storage of and public access to texts not pertinent to its own text development. (5) Experimental Design of Certification Systems.

3.8 The Third Proposal

The first proposal, giving identifiers a formal place in texts and treating them abstractly, and so disabling the hiding of identifiers and enabling the avoidance of name collision, has a certain appeal as a management device even on its own. The second proposal, to transfer these same benefits to the use of references to texts, also seems promising as a possible approach to managing intertextual dependencies, although it is quite vague as to how this could be of more than heuristic value. Hence, the third proposal: Take records seriously and implement certification systems to formalize them, taking special notice of the promises one must make to succeed.

Thanks

Thanks to Richard Eaton, Robert Constable and James Caldwell for numerous discussions with me on these topics. I am also grateful for remarks by Constable on a draft.

References

[1] Stuart Allen, Mark Bickford, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. *FDL: A Prototype Formal Digital Library.* The PRL Group at Cornell University, 2002. www.nuprl.org/html/FDLProject/02cucs-fdl.html.

[2] C.R. Cullen et al. *Authenticity in a Digital Environment,* CLIR Report pub92. Council on Library and Information Resources, Washington, DC, 2000. www.clir.org/pubs/abstract/pub92abst.html.

[3] Michael Dummett. *Frege: Philosophy of Language.* Harvard University Press, Cambridge, MA, 1991.

[4] Anne J. Gilliland-Swetland. *Enduring Paradigm, New Opportunities: The Value of the Archival Perspective in the Digital Environment,* CLIR Report pub89. Council on Library and Information Resources, Washington, DC, 2000. www.clir.org/pubs/abstract/pub89abst.html.

[5] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, NY, 1979.

[6] The PRL Group. PRL Project Home Page. www.nuprl.org.

[7] Jason J. Hickey. *The MetaPRL Logical Programming Environment.* PhD thesis, Cornell University, Ithaca, NY, January 2001. www.nuprl.org/documents/Hickey/Hickeythesis.html.

[8] Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL Home Page. metaprl.org.

[9] Edward Higgs, editor. *History and Electronic Artefacts.* Oxford University Press Inc., New York, 1998.

[10] Edward Higgs. *The Role of Tomorrow's Electronic Archives*, pages 184–194. In [9], 1998.

[11] Peter B. Hirtle. *Archival Authenticity in a Digital Age.* In [2], 2000. www.clir.org/pubs/reports/pub92/hirtle.html.

[12] Paul Jackson. *The Nuprl Proof Development System, Version 4.2 Reference Manual and User's Guide.* The PRL Group at Cornell University, 1996. www.nuprl.org/documents/Jackson/Nuprl4.2Manual.html.

[13] Roy Levin and Paul R. McJones. The Vesta approach to precise configuration of large software systems. Technical Report 105, Systems Research Center (Digital Equipment Corporation), 130 Lytton Avenue, Palo Alto, California 94301, 1993. citeseer.nj.nec.com/levin93vesta.html.

[14] Kristen Nygaard and Ole-Johan Dahl. The development of the SIMULA languages. In Richard L. Wexelblat, editor, *History of Programming Languages.* Academic Press, New York, 1981.

[15] W. Boyd Rayward. *Electronic Information and the Functional Integration of Libraries, Museums, and Archives*, pages 207–225. In Higgs [9], 1998.

[16] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual.* Springer-Verlag, New York, third edition, 1988.

[17] Tim Teitelbaum and Thomas Reps. The Cornell program synthesizer: a syntax–directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[18] Kenneth Thibodeau. Building the Archives of the Future: Advances in Preserving Electronic Records at the National Archives and Records Administration. *D-Lib Magazine*, 7, 2001. www.dlib.org/dlib/february01/thibodeau/02thibodeau.html.

[19] N. Wirth and C.A.R. Hoare. A contribution to the development of ALGOL. *Communications of the ACM*, 9:413–432, 1966.

[20] N. Wirth and Helmut Weber. EULER: a generalization of ALGOL, and its formal definition: Part II. *Communications of the ACM*, 9:89–99, 1966.