12

Towards a Theory of Document Structure

Bengt Nordström
Chalmers University of Technology and the University of Göteborg

Abstract

The structure of documents of various degree of formality, from scientific papers with layout information and programs with their documentation to completely formal proofs can be expressed by assigning a type to the abstract syntax tree of the document. By using dependent types – an idea from type theory – it is possible to express very strong syntactic criterion on wellformedness of documents. This structure can be used to automatically generate parsers, type checkers and structure oriented editors.

12.1 Introduction

We are interested to find a general framework for describing the structure of many kinds of documents, like

- books and articles
- "live" documents (like a web document with parts to be filled in)
- programs
- formal proofs

Are there any good reasons that we use different programs to edit and print articles, programs and formal proofs? A unified view on these kind of documents would make it possible to use only one structure oriented editor to build all of them, and it would be easier to combine documents of different kinds, for instance scientific papers, programs with their documentation, informal and formal proofs and simple web forms.

Such a view requires that we have a good framework to express syntactic wellformedness (from things like the absence of a title in a footnote to

correctness of a formal proof) and to express how the document should be edited and presented.

12.2 Examples

12.2.1 Proof editors and documented proofs

A proof editor is a program which helps a user to write a formal proof. In doing this, it can give suggestion of what proof rules to use and can check that the proof rules are correctly applied. There are also commands to delete part of the proof and to regret earlier steps.

Most proof editors are commmand-based; the user manipulates a proof state using a command language. But there are also WYSIWIG proof editors [15, 9, 3] where a proof is obtained by direct manipulation. The proof is presented on the screen to the user and edited by editing the presentation. The things which are edited are incomplete expressions, the missing parts are presented as placeholders. The basic editing step is to replace a placeholder with an (incomplete) expression or vice versa. In this article, we are mainly interested in these kind of editors.

Proof editors like Coq [5] and HOL Light [10] are becoming more and more mature. To become a tool which people can rely on, it it still necessary to develop a small and understood proof checker with a well-understood relationship between the proof checker and the logic. But this is not enough. The proofs must be readable by humans. The final product should be a document consisting of a mixture of formal proofs, programs whose values are formal proofs and informal mathematical text. The editor to produce this document is important.

12.2.2 Integrated Development Environments

An integrated development environment is a structure oriented editor for a programming language together with other tools to develop programs (debugging etc). One of the first programming environments based on a structure oriented editor was the MENTOR system [7] developed by Gilles Kahn et al in the 70's. In this system it was possible to mix different programming languages. The system was later generalized and simplified in the CENTAUR system [4]. An example of a more modern system is Eclipse [8] which is a framework for creating programming environments. New tools and languages can be added as plug-ins to the existing system.

A fundamental part of such a programming environment is a set of tools for documentation of programs. It is thus necessary to edit programs and documents with layout information.

12.2.3 Editable forms on the web

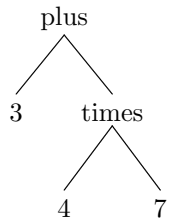
Documents on the web which contains forms to fill in are generally produced in an ad hoc manner. To create a form requires a lot of programming. For instance, a filled in date must be checked for correctness. These kind of interactive checks for syntactic wellformedness is something we recognize from proof editors. A proof editor (based on direct manipulation) is exactly doing this, it checks syntactic wellformedness and helps the user in other ways to build up a formal proof. Here we are looking at a document with slots to be filled in as an expression with holes in a WYSIWIG proof editor. If we have a type system which can express the syntactic restrictions of the placeholders, we could let the web browser use a builtin type checker to check that the forms are syntactically correct.

In this way, there would be no need to write ad hoc code checking the syntactic correctness, the expected types of the slots give enough information for a type checker to check this.

12.3 A first step

It is clear that documents (programs, proofs, etc) have a common syntactical structure. They are built up from parts, where each part is either simple or compound.

The simple parts could be an individual character (for running text), an integer or an empty list (for programs), 0 or π (for mathematical expressions). A compound part in a document could be a section (with a section heading and a body), in a program it could be a while-statement (with a boolean expression and a statement) and in a mathematical expression it could be an addition (with two operands). We will look at a simple part as a functional constant without arguments, while a compound part is a functional constant applied to its arguments (parts). For instance, the expression $3 + 4 * 7$ is built up in the following way:



We need a notation for these kind of trees, we will write the tree above as

$$+\ 3\ (*\ 4\ 7).$$

The functional constants $+$ and $*$ are written before their arguments.

We will use ideas from type theory to express the structure of documents. The syntactic wellformedness of a document will be expressed by giving a type to its abstract syntax.

The advantages of being explicit about this structure becomes evident when writing programs that manipulate documents. This is one of the reasons that people are going over from the untyped HTML to the weakly-typed XML for describing documents on the web.

A type system has also advantages for a structure oriented editor. The editor can be made more user-friendly since it will only suggest type-correct alternatives (completions become more meaningful if the number of suggestions is small).

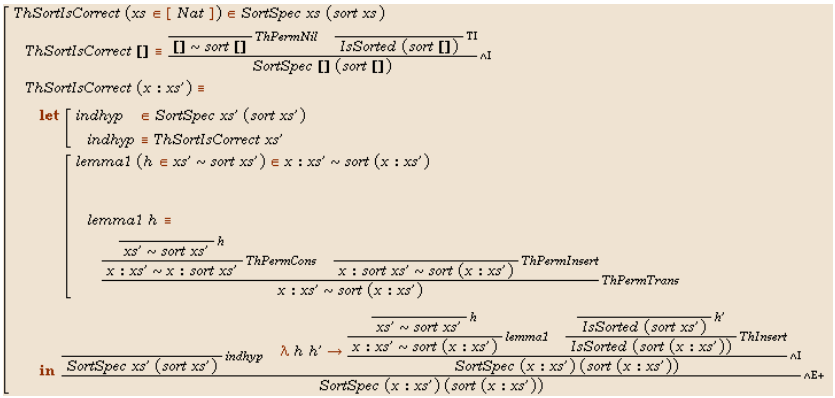
It will be easier to write a program taking a typed document as input. There is no need to check syntactic wellformedness, since this will be done by the type-checker. If we have a programming language in which it is possible to use the type system to express the structure of a document, then we can discover erroneous document-processing programs already when they are type checked (i.e. at compile time). In this way, we are sure that the resulting document is structurally well formed.

12.4 Abstract and concrete syntax

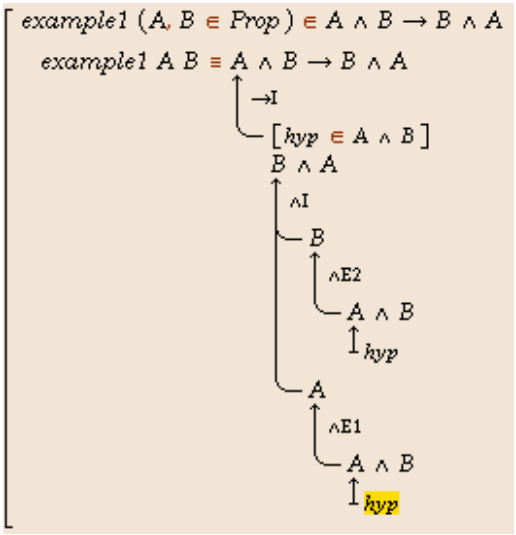
A document is represented by an *abstract syntax tree* which gives all important syntactic structure to it. This is a mathematical object explaining how a document is built up from its parts. The document (program, proof, etc) is presented by using a *linearization function*, which takes the abstract syntax to its *concrete syntax*. The concrete syntax is just a way to present the abstract object.

In general, it is possible to have many linearization functions, so that

the same document can be presented in different ways. A proof, for instance, could (like in the Alfa proof editor [9]) be presented as a proof term, a natural deduction proof tree



or a Fitch-like tree.



A web document would be presented in different ways on paper or screen (and different for screens of different sizes). A text in a natural language could be presented in English or Swedish by different linearization functions. This idea has been successfully used by Aarne Ranta in his GF project [20].

The inverse of the linearization function is *parsing*. There are two

consequences of this view of parsing. First, parsing is in general ambiguous, since there is no requirement that the linearization function is injective. There are many abstract syntax trees with the same concrete syntax. The second consequence is that parsing is not computable in general (to compute the abstract syntax tree from the concrete syntax may require knowledge about the world, this is the case for expressions in an unrestricted natural language), or not feasible (for instance if we present a proof of a mathematical proposition with the word true, then it would be necessary to find the proof in order to build its abstract syntax tree).

If the document is built using a WYSIWYG structure editor, one is tempted to think that there is little need for abstract syntax. Only the concrete syntax of the document is presented and the editing is done by changing the concrete syntax. However, the editing commands (like navigation, insertions and deletions) should reflect the abstract syntax. In fact, these commands can uniformly be generated from the abstract syntax. A better understanding of its structure is essential to give better designs of these commands [18].

In such an editor, it is necessary to present the document being edited in a concrete syntax which has a computable unambigous parser. The list of input characters from the input should uniquely determine the corresponding abstract syntax. It is an advantage if the parser can be automatically generated from the linearization function, and this is often the case (but the resulting parser is often ambiguous) [21].

One important concrete syntax is the *text syntax*, which is a parsable, unambiguous concrete syntax constructed in such a way that it is easy to use when editing the document in an unstructured text editor. For a programming language, there is usually no difference between the text syntax and other concrete syntaxes. Knuth's WEB system [14] and some other systems for literate programming are exceptions, where the concrete syntax of programs is slightly improved over the text syntax. For a document which is a combination of informal text with layout information and formal mathematical text, the classic example of text syntax is LaTeX without macros. Programming languages are in general infeasible to use for this purpose, since they use a quoting mechanism for text strings. On the other hand, mathematical expressions containing infix, postfix and mixfix operators must also be easy to input. This rules out XML and prefix languages as good text syntax for mathematical documents. Only an experienced Lisp programmer prefers to read and

write the expression $(+\ x\ (-\ (*\ y\ 3)\ 2))$ instead of the mathematical expression $x + y * 3 - 2$.

12.5 Expressing document structure

We will express the structure of a document by giving a type to the abstract syntax tree of the document. The abstract syntax tree is a mathematical object, and we will use ideas from Martin-Löfs type theory [16, 17, 1] to express the type. We assume that we have a syntactic category of identifiers, which either stand for variables or constants. We will use the notation i, x and c for identifiers, variables and constants.

The simple kind of abstract trees which was mentioned earlier were built up using application only. They had the shape $(c\ e_1\ \ldots\ e_n)$, where c is a constant and e_i are expressions. But in order to express variable binding, which is so common in mathematics and programming, we will also use expressions built up by abstracting n variables ($n \geq 1$) from an expression e. We will use the lambda notation $\lambda x_1 \ldots x_n.e$ for these kind of expressions.

So for instance, the mathematical expression $\int_0^1 e^x dx$ will be represented by

```
Int 0 1 \x.(power e x)
```

and $\sum_{i=1}^{100} 1/i$ will be represented by

```
sum 1 100 \i.(div 1 i).
```

We will start with a simple type system similar to a type system for functional programming languages. Then we will extend it to a system for dependent types. But before we do this, we have to explain how constants are declared.

12.5.1 Declaration of constants

There are two kinds of constants: primitive and defined. A defined constant gets its meaning by a definition, while a primitive constant gets its meaning outside the language. Our notion of computation is expansion of definitions and a computation terminates when there is no definition to expand. The type system must have the property that the type of an expression is preserved under computation.

Primitive constants. A constant is *primitive* when it has no definition, it can only be computed to itself. For instance, consider the following declarations:

$$\begin{aligned}
&\textbf{data} \quad \mathsf{N} \in \mathsf{Set} \quad \textbf{where} \\
&\qquad\qquad \mathsf{zero} \in \mathsf{N}; \\
&\qquad\qquad \mathsf{s} \in \mathsf{N} \to \mathsf{N} \\
&\textbf{data} \quad \mathsf{List}\ (A \in \mathsf{Set}) \in \mathsf{Set} \quad \textbf{where} \\
&\qquad\qquad \mathsf{nil} \in \mathsf{List}\ A\ \mathsf{zero} \\
&\qquad\qquad \mathsf{cons} \in (A\ (\mathsf{List}\ A)) \to \mathsf{List}\ A
\end{aligned}$$

This declares the primitive constants N, zero and s. It also expresses that N has the type Set (i.e. that it is a set), that the type of zero is N, and that s $x \in$ N whenever $x \in$ N. Furthermore, it expresses that this is an inductive definition of N, so there are no more primitive constants which produce objects in N. The only[1] way to introduce an object in Set is to use a declaration of this kind. The second declaration defines the set of lists in a similar way.

A set with no elements can be introduced by having an empty list of declaration after the keyword **where**.

Explicit definitions. A *defined* constant is a constant which has a definition, for instance

$$\begin{aligned}
&\mathsf{plus2} \in \mathsf{N} \to \mathsf{N} \\
&\mathsf{plus2} = \lambda x.\ \mathsf{s}\ (\mathsf{s}\ x)
\end{aligned}$$

is an example of an *explicitly defined constant*. This gives the meaning of a new functional constant by a right hand side. In order for this kind of definition to be correct it is necessary that the type is correct and that the right hand side is an object in that type.

Implicit definitions. An implicitly defined constant is defined by pattern matching (over primitive constants) and recursion. The standard example is:

$$\begin{aligned}
&\mathsf{add} \in \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\
&\mathsf{add}\ \mathsf{zero}\ y = e \\
&\mathsf{add}\ (\mathsf{s}\ x)\ y = \mathsf{s}\ (\mathsf{add}\ x\ y)
\end{aligned}$$

[1] It is however possible to give an object in Set a new name by using a definition.

This says that add is a constant which takes two objects in N as arguments and yields an object in N as result. It is defined by the defining equations in the declaration. In order for an implicit definition to be correct it is necessary that the patterns are defined for all arguments and that the definition is unambigous. We could for instance require that the patterns are disjoint and complete. It is also necessary that the expansion of definitional equality is terminating. Usually this is checked by finding some wellordering relation in the definition.

The distinction between implicit and explicit definitions is important because it separates the definitions according to how difficult it is to check them. There will always be a wish to extend the notion of a correct implicit definition, any fixed limitation of this concept will sometimes be a straitjacket.

12.6 A simple type system

In order to explain type theory, we will first describe a framework for a simple theory with nondependent types. This corresponds closely to the core of functional programming languages like ML and Haskell. The next step is to extend this to dependent types. The description will be very brief and no formal rules will be given.

Let us start with the system without dependent types. There is one basic type, the type Set whose objects are the sets. Then we have the following type-forming operations:

- El A, the type of elements in A (which must be an object in the type Set),
- $(A_1 \ \ldots \ A_n)$, the cartesian product,
- $A_1 \to A_2$, the function type,
- Record $(c_1 \in A_1) \ldots (c_n \in A_n)$, record type.

We will write El A as A, since it will always be clear from the context if A stands for an object in Set or the types of elements in A.

The type-forming operations are of course not minimal: records can be used to decode cartesian products, functions can be used to decode anything, natural numbers can decode anything. We want to present a system which can be used to express the abstract syntax of documents in a straightforward way, without any detours via complicated codings. For instance, we use cartesian products to express more than one argument of a function instead of currying since this is the most straightforward way of looking at a documents with several parts. Currying is some kind

of coding. Another advantage of this approach is that we want it to be clear from the outer form of an expression whether it is computed or not. The alternative of using records would require an explanation of not only how to express a tuple, but also patterns in implicit definitions.

The type system is defined by induction over the type structure. Assume that $A_1, \ldots, A_n$ are given types. We will follow the tradition in type theory and explain a new type-forming operation by explaining what it means to be an object in it. This should be followed by an explanation what it means for two objects to be equal in the type (each type comes together with an equality defined on it). The definition of equality will only be given in cases where it is not obvious.

12.6.1 Cartesian product

An object may consist of a finite number of well-formed objects.

The *cartesian product* $(A_1 \ \ldots \ A_n)$ is a type for $n \geq 0$. The elements of this type are tuples of the form $(a_1 \ \ldots \ a_n)$, where $a_i \in A_i$. Notice that $() \in ()$.

12.6.2 Functions

The type of functions $A_1 \rightarrow A_2$ is explained by saying that $f \in A_1 \rightarrow A_2$ means that $f\ a_1 \in A_2$ whenever $a_1 \in A_1$.

12.6.3 Record types

A record is an object which like a tuple has a finite set of components, the difference is that the components in records have names. These names can be used to project the component from the record, the expression $e.c_i$ projects the component with the name c_i from the record e. We are also using the names as locally defined constants inside the record.

The *record type* (or labeled cartesian product) $\mathsf{Record}\ (c_1 \in A_1) \ldots (c_n \in A_n)$ is a type, for $n \geq 0$. That we have an object e of this type means that $e.c_i \in A_i$, for $i \leq n$, and also that $e = \mathsf{record}\ (c_1 = e.c_1) \ldots (c_n = e.c_n) \in \mathsf{Record}\ (c_1 \in A_1) \ldots (c_n \in A_n)$

Two elements d and e in $\mathsf{Record}\ (c_1 \in A_1) \ldots (c_n \in A_n)$ are equal if $d.c_i = e.c_i \in A_i$ for $i \leq n$.

So, each record type has its own equality relation defined, and if the expressions d and e are of a record type, then they are equal in the empty record type, i.e. $d = e \in \mathsf{Record}$. There is an inherent notion

of sub-typing here, if $e \in \mathsf{Record}\ (c_1 \in A_1)(c_2 \in A_2)\ldots(c_n \in A_n)$ then $e \in \mathsf{Record}\ (c_1 \in A_1)(c_2 \in A_2)\ldots(c_j \in A_j)$ for all $j \leq n$.

We can use a record like a local definition in the sense of allowing the labels to occur in the definition of later fields. In an expression $\mathsf{record}\ (c_1 = e_1)\ldots(c_n = e_n)$ the label c_i may occur in the expression e_j, for $j > i$, and the above expression is defined by the following equalities (we use $e[c := a]$ for the expression obtained by substituting all free occurrences of the identifier c for the expression a in the expression e):

$$\begin{aligned} e.c_1 &= e_1 \\ e.c_2 &= e_2[c_1 := e.c_1] \\ &\ldots \\ e.c_n &= e_n[c_1 := e.c_1, \ldots c_{n-1} := e.c_{n-1}] \end{aligned}$$

Records are used in documents when, for instance part of the document is a database (or is computed from a database). A typical example is a database of literature references (like BibTeX). A watered down version is also used in HTML and XML for expressing arguments to some primitive constants (attributes). In these languages, the type of elements are restricted to be strings.

12.7 Dependent types

If we have a type system where a type can depend on an object in another type, then we can generalize the type forming operations to dependent types.

This is the case in the type system just outlined. For instance, if $A \in \mathsf{Set}$, then A is a type. And the type A depends trivially on the object A in the type Set. This opens the possibility to consider tuples of the form $(a\ b)$, where $a \in A$ and $b \in B(a)$, where the type $B(a)$ depends on the object $a \in A$. We can also have functions f, where the type of $f\ a$ depends on the value of the argument a.

A simple example is the following declaration of the type of vectors $(\mathsf{Vect}\ A\ n)$ of length n:

$$\begin{aligned} \textbf{data}\quad \mathsf{Vect}\ (A \in \mathsf{Set})(n \in \mathsf{N}) \in \mathsf{Set}\quad \textbf{where}& \\ \mathsf{vnil} &\in \mathsf{Vect}\ A\ \mathsf{zero} \\ \mathsf{vcons} &\in (a \in A)(as \in \mathsf{Vect}\ A\ n)\mathsf{Vect}\ A\ (\mathsf{s}\ n) \end{aligned}$$

The constructor vnil creates vectors of length 0, while the constructor vcons builds a vector of length $n+1$ from a vector of length n.

We can use this type to express tables of fixed size: the type Vect (Vect A n) m can be used when we want to express a table of the dimensions $n*m$. On the other hand, the type List (Vect A n) expresses a table with arbitrary number of rows of length n.

12.7.1 Dependent functions

The expression $(x \in A)B$ is a type if A is a type and $B\ x$ is a type for $x \in A$. This is the type of functions f, which when applied to an argument a in A yields an expression $f\ a\ \in B\ a$. Notice here that the type of the application depends on the argument of the function.

12.7.2 Dependent records

The expression Record $(c_1 \in A_1)(c_2 \in A_2)\ldots(c_n \in A_n)$ is a type. Each type expression A_i may depend on previous labels c_j, $j < i$, so for instance A_4 may depend on c_1, c_2 and c_3.

Here the requirements on an object e of the type are that $e.c_i \in A_i$, and we know that $e =$ record $(c_1 = a_1)(c_2 = a_2)\ldots(c_n = a_n)$ where $a_i = e.c_i$.

Example from mathematics. The disjoint union of a family of sets $\sum_{n\in\mathsf{N}} A^n$ is the type of pairs, the first being a natural number n and the second a vector $(a_1, ..., a_n)$ of length n.

Example from logic Dependent types are used to represent the universal and existential quantifiers (using the Curry-Howard-Kolmogorof isomorphism).

Example from semiformal language. A form which contain a part whose structure is dependent on the value of a previous part. For instance, when you fill in a form for your address, the structure of the address depends on the country you live in. The syntax of a telephone number depends on the area code of the number (for instance in Gothenburg we can only have 6 or 7 digits after the area code).

12.8 An application: a web browser with a built in type checker

Using these ideas it would be possible to integrate an interactive type checker (a proof editor[1]) in a web browser. When we want to present a form with slots to fill in, we just produce a document with typed place holders. Using dependent types it is possible to express very strong requirements on syntactic wellformedness, requirements which are currently expressed in an ad hoc way. For instance, if we had a placeholder (year, month, day) for a date, we can express that the type for this would be

```
record (year: Year)(month: Fin 12)(day: Fin (f year month))
```

where `Year` is a type of integers in some range around 2000, `Fin x` is the interval `[1,x]` and `f` is a function computing the maximal number of days in a specific year and month.

12.9 Related work

The main source of inspiration of this work comes from proof editors like Agda, Coq, HOL, Isabelle, Lego [1, 5, 13, 19] in the Types consortium [11, 12, 22] during the last 15 years. Many of these systems use dependent types to express mathematical propositions. The objects in such a type are the formal proofs of the proposition. So, a false proposition is represented by an empty type.

The view of the abstract syntax as the fundamental syntactical structure is persistent in computer science. It is a standard way of modularizing compilers and other program-manipulation programs. It is also implicit in denotational semantics, where the semantics is expressed by induction over the abstract syntax. This is probably the reason why the abstract syntax trees are sometimes called semantical (for instance, in linguistics and in the semantical web)

The idea of distinguishing between abstract and concrete syntax was introduced in linguistics by Curry [6] who used the words tectogrammatical and phenogrammatical structure. In Aarne Ranta's Grammatical Framework this is a major idea for obtaining multilinguality (the same

[1] The idea of identifying type checking with proof checking comes from the proposition-as-types principle in which you identify a proposition with the set of its proofs. To write a proof of a given proposition then becomes the same as writing a program with a given type. For instance, to write a proof of $A \wedge B \supset B \wedge A$ consists in writing a program which takes a proof of $A \wedge B$ to a proof of $B \wedge A$.

abstract syntax tree is presented by many concrete syntaxes, one for each language involved) and translation between different fragments of natural languages. Linguists in general tend to refuse to distinguish between abstract and concrete syntax.

The idea of using abstract syntax trees for documents is also present in the XML community.

12.10 Acknowledgements

I would like to thank the reviewers for many helpful comments.

It was during my visit to INRIA in the end of the 70's that I learned about the MENTOR project and learned about a more general view on programming environments. I will always be grateful to Gilles Kahn for his endless source of inspiration, his ability to ask good questions and suggest good solutions.

Bibliography

[1] Agda homepage. `unit.aist.go.jp/cvs/Agda/`.

[2] Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors. *Mathematical Knowledge Management, Third International Conference, MKM 2004, Bialowieza, Poland, September 19-21, 2004, Proceedings*, volume 3119 of *Lecture Notes in Computer Science*. Springer, 2004.

[3] Richard Bornat and Bernard Sufrin. Animating Formal Proof at the Surface: The Jape Proof Calculator. *Computer Journal*, 42(3):177 – 192, 1999.

[4] P. Borras, D. Clement, Th. Despeyrouz, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (PSDE)*, volume 24, pages 14–24, New York, NY, 1989. ACM Press.

[5] Coq homepage. `pauillac.inria.fr/coq/`, 1999.

[6] Haskell B. Curry. Some logical aspects of grammatical structure. In Roman O. Jakobson, editor, *Structure of Language in its Mathematical Aspects. Proceedings of the 12th Symposium in Applied Mathematics*, pages 56–68, 1961.

[7] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structured editors: The MENTOR experience, 1984.

[8] Eclipse homepage. `www.eclipse.org`.

[9] T. Hallgren. Alfa homepage. `www.cs.chalmers.se/~hallgren/Alfa/`, 1996-2000.

[10] J. Harrison. The HOL light theorem prover. `www.cl.cam.ac.uk/~jrh13/hol-light/`, 2006.

[11] G. Huet and G. Plotkin, editors. *Logical Frameworks: First International Workshop on Logical Frameworks, Antibes, May, 1990*. Cambridge Univ. Press, 1991.

[12] G. Huet and G. Plotkin, editors. *Logical Environments: Second International Workshop on Logical Frameworks, Edinburgh, May, 1991*. Cambridge Univ. Press, 1993.

[13] Isabelle Homepage. `www.cl.cam.ac.uk/Research/HVG/Isabelle/`, 2003.

[14] D. E. Knuth. *Literate Programming*. CSLI, 1992.

[15] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806 of *LNCS*, pages 213–237, Nijmegen, 1994. Springer-Verlag.

[16] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[17] B. Nordström, K. Petersson, and J. M. Smith. *Martin-Löf's Type Theory.*, chapter 1, pages 1 – 33. Oxford University Press, 2001.

[18] Luca Padovani and Riccardo Solmi. An investigation on the dynamics of direct-manipulation editors for mathematics. In Asperti et al. [2], pages 302–316.

[19] R. Pollack. The LEGO proof assistant. `www.dcs.ed.ac.uk/home/lego/`, 1997.

[20] A. Ranta. Grammatical Framework Homepage. `www.cs.chalmers.se/~aarne/GF/`, 1999–2005.

[21] A. Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *The Journal of Functional Programming*, 14(2):145–189, 2004.

[22] The Types Project Homepage. `www.cs.chalmers.se/Cs/Research/Logic/Types/`.

[23] Laurent Théry, Yves Bertot, and Gilles Kahn. Real theorem provers deserve real user-interfaces. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, pages 120–129, 1992.