

FoolProof: A Component Toolkit for Abstract Syntax with Variable Bindings

Kees Hemerik
Dept. of Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
c.hemerik@tue.nl

June 2, 2008

Abstract

FOOLPROOF is intended as a component toolkit for implementation of formal languages with binding structures. It provides a coherent collection of components for many common language processing tasks, in particular those related to binding structures. FOOLPROOF consists of: a meta-language for specifying signatures with variable bindings; a signature editor for constructing well-formed signatures; a small collection of interfaces for manipulating syntax trees and binding structures at various levels of detail; a set of generic components for processing syntax trees with binding structures, in particular for: copying, substitution, editing, matching, unification and rewriting; a generator which maps signature specifications to signature-specific classes. FOOLPROOF is being implemented in Object Pascal and will eventually take the form of a component library for the Delphi environment.

Keywords

abstract syntax, variable bindings, signatures, structure editors, API generators, component library, Delphi.

foolproof (*adj*): *so simple, plain or reliable as to leave no opportunity for error, misuse, or failure*
Webster's New Collegiate Dictionary

1 Introduction

Many computing systems manipulate syntactic objects like programs, formulas, types, and proofs. Well-known examples of such systems are compilers, interpreters, structure editors, static analyzers, rewriting systems, theorem provers,

and proof assistants. A common characteristic of the syntactic objects manipulated by these systems is, that they contain variable bindings like declarations, definitions, parameters, and quantifications. Usually the syntactic objects are specified by some kind of signature and represented by some form of abstract syntax trees (AST). The AST is a central data structure in the system. Usually it is built up by a parser or structure editor and subsequently traversed, annotated and transformed by other parts of the system. A large part of the necessary code can be generated automatically from the signature specification.

The methods for specifying and implementing binding structures are less developed and standardized, however:

- Context-dependent properties of programs are often expressed by means of attribute grammars, sometimes extended with extra facilities for expressing bindings, such as references to binding positions [15]. This works well for static syntax trees, but is inadequate for systems involving rewriting, such as theorem provers.
- In type theory, binding structure is usually handled in combination with typing and expressed by means of a deduction system (see e.g. [3]). This is a powerful and general technique, well suited for meta-theoretical work, but it provides little support for constructing implementations.
- In general, little support (in the form of formalisms and tools) is available for specifying and implementing binding structures. As a consequence, in implementations of formal languages with binding structures many essential but error-prone operations - such as copying, substitution, matching, unification and rewriting - are still hand-coded.

FOOLPROOF is intended as a component toolkit for implementation of formal languages with binding structures, in particular small but intricate formalisms like typed lambda calculi, logics, and languages that integrate specifications and programs. With FOOLPROOF we aim to facilitate the specification and implementation of such systems in several ways:

- By extending the notions of signature and abstract syntax tree with facilities for expressing binding structures.
- By designing a meta-language for specifying signatures with binding structures;
- By providing a signature designer (actually a kind of structure editor for signatures), which supports the design of well-formed signatures.
- By providing software components that implement the essential operations involving binding structures, such as copying, substitution, rewriting, matching, and structure editing. In addition, FOOLPROOF also provides more traditional components such as various parsers and a formatter. The components are implemented in Object Pascal and will eventually take the form of a component library ('package') for the Delphi environment.

In this paper we concentrate on some of the practical aspects of FOOLPROOF, such as the ways to specify signatures with variable bindings, the structure of the resulting class hierarchy, and some of the techniques for handling binding structures at a signature independent level. The underlying theory will be presented elsewhere.

The structure of this paper is as follows:

We provide a stepwise introduction to FOOLPROOF's signature notions in sections 2 and 3.

Section 2 does not deal with binding structures yet, but illustrates FOOLPROOF's approach to handling abstract syntax trees at various levels of abstraction.

In section 3 we introduce a new abstraction level, which is geared towards handling all aspects of binding structures. This level provides *items*, which correspond to defining occurrences of names and their properties, and *references*, which correspond to applied occurrences. We show how operations like substitution and copying can be handled in a uniform and signature independent way. As a concrete example we present a signature for Pure Type Systems (PTSs), a family of typed lambda calculi [3, 4].

We conclude by discussing the current status of FOOLPROOF in section 4 and related work in section 5.

2 Abstract Syntax

In this section we first consider signatures without binding structures and the way they are handled within FOOLPROOF. In particular we show how FOOLPROOF deals with syntax trees at two different levels of abstraction: a signature independent abstract level dealing with general tree properties only, and a signature dependent level where different node kinds and their structure can be distinguished. This two-level approach paves the way for the treatment of binding structures in section 3, which is based on a *three*-level approach.

The abstract syntax of a programming language or logic language is usually defined as a *term algebra* over a *signature*. A signature consists of a finite set of *sorts* and a finite set of *operators*, each of which has a (possibly empty) sequence of argument sorts and a result sort. E.g. for a programming language with while-statements we would have sorts like *Expr* and *Stat* and an operator $while : Expr \times Stat \rightarrow Stat$. For easier reference to the sub-terms usually labels are provided, resulting in something like $while : [guard : Expr, body : Stat] \rightarrow Stat$.

It is well-known that a signature of this kind can be mapped to a class hierarchy in a class-based object-oriented language such as Java or Object Pascal. Every sort S is mapped to a class C_S , and each operator $o : [f_1 : S_1, \dots, f_n : S_n] \rightarrow S$ is mapped to a subclass of the class C_S , which has members with names derived from f_1, \dots, f_n which are references to objects of classes C_{S_1}, \dots, C_{S_n} . Well-known examples of systems employing such a mapping are [1, 2, 13, 16, 18]. Some of these systems make use of an elaborate architecture of traversal routines

or visitor classes, which are also generated from the signature.

In FOOLPROOF we take a different approach. We start from an abstract base class `T_Node` which just provides some declarations (but no implementations) of methods for general tree operations, such as giving the number of sons of a node or getting or setting the *i*-th son. In a slightly simplified form the class header for `T_Node` is as follows:

```
type
  T_Node =
    class(TObject)
    public
      function NodeCount: Integer; virtual; abstract;
      function GetNode(I: Integer): T_Node; virtual; abstract;
      procedure SetNode(I: Integer; ANode: T_Node); virtual; abstract;
      class function SortCode: T_SortCode ; virtual; abstract;
    end;
```

For each sort *S* of the signature a subclass `T_S` is constructed. For instance, for our example sort *Stat* we have:

```
type
  T_Stat =
    class(T_Node)
      // fields and methods relevant for all statements
    end;
```

The scheme for an operator $o : [f_1 : S_1, \dots, f_n : S_n] \rightarrow S$ is somewhat more involved. It yields a class which descends from the class corresponding to *S* and it does provide members for accessing and setting each of the sub-terms. But in addition it also overrides the methods of the ancestor class `T_Node`. Thus, the class is provided with two interfaces, a signature dependent one and a signature independent one. For instance, for the signature element

```
while: [guard: Expr, body: Stat] < Stat
```

the following class header is generated:

```
type
  T_While =
    class(T_Stat)
    protected
      F_guard: T_Expr;
      F_body: T_Stat;

      procedure Set_guard(A_guard: T_Expr); virtual;
      procedure Set_body(A_body: T_Stat); virtual;
    public
      constructor Create(A_guard: T_Expr; A_body: T_Stat);

      function NodeCount: Integer; override;
      function GetNode(I: Integer): T_Node; override;
      procedure SetNode(I: Integer; ANode: T_Node); override;
```

```

class function SortCode: T_SortCode ; override;

property _guard: TExpr read F_guard write Set_guard;
property _body : TStat read F_body  write Set_Body;
end;

```

The properties `_guard` and `_body` provide access to the sub-terms with types that are specific to the while-construct. The functions `NodeCount`, `GetNode` and `SetNode` on the other hand provide uniform access to the sub-terms with types that are signature independent. The following code is generated for them:

```

function T_While.NodeCount: Integer;
begin
  Result := 2;
end;

function T_While.GetNode(I: Integer): T_Node;
begin
  case I of
    0: Result := F_guard;
    1: Result := F_body
  else
    raise ENodeException.Create('T_While.GetNode Index error');
  end;
end;

procedure T_While.SetNode(I: Integer; ANode: T_Node);
begin
  case I of
    0: F_guard := ANode as T_Expr; // 'as' is a checked downcast
    1: F_body  := ANode as T_Stat;
  else
    raise ENodeException.Create('T_While.SetNode Index error');
  end;
end;

```

The following example shows how the general and language-specific levels can be used together. It is a simple procedure to traverse an AST in pre-order and do something for each while node:

```

procedure PreOrderWhile(ANode: T_Node);
var
  I: Integer;
begin
  if ANode is T_While then
    with ANode as T_While do
      // ... do something with _guard and _body ...

      for I := 0 to ANode.NodeCount - 1 do
        PreorderWhile(ANode.GetNode(I));
      end;
  end;
end;

```

3 Incorporating Binding Structures

In this section we extend the signatures of section 2 with facilities for variable bindings. By means of examples we identify various binding structures underlying common constructs in programming languages and logics. We point out the need for a uniform treatment of binding structures and illustrate the way this is achieved in FOOLPROOF by the introduction of a new abstraction level. We illustrate the concepts by means of examples drawn from Pure Type Systems (PTSs), a family of typed lambda calculi [3, 4].

3.1 Binding structures in programming languages and logics

Most programming languages offer facilities for declaring or defining constants, variables, procedures, functions, types or classes. Similarly, in many logics one can introduce constants, hypotheses, theorems, and proofs, and associate names with them. Although the notations used for these language constructs are often ad hoc and differ widely from one language to another (and even within one language), the constructs themselves have much in common, viz. the introduction of a *binder*, i.e. a name with certain properties and a certain scope. Within the scope of the binder it may be referred to by its name. In programming parlance derived from ALGOL 68 [29] these two different uses of a name are usually called *defining occurrences* and *applied occurrences* respectively. Let us illustrate the commonalities by considering some familiar notions from programming languages and logics, writing them in a form which stresses the common elements:

constant definition Associates a name with a manifest value and its type, e.g.:

- $N = 3 : nat$
- $pair = (3, true) : nat \times bool$

variable declaration Associates a name with a type of modifiable values, e.g.:

- $x : \mathbf{var}[real]$

type definition Associates a name with a type expression, e.g.:

- $gridpoint = nat \times nat : \mathbf{type}$

function definition Associates a name with a function expression and its type, e.g.:

- $idnat = (\lambda x : nat.x) : nat \rightarrow nat$
- $polyid = (\lambda \alpha : *. (\lambda x : \alpha.x)) : (\Pi \alpha : *. \alpha \rightarrow \alpha)$

named hypothesis Associates a name with an assumed proposition, e.g.:

- $H : P \Rightarrow Q$

theorem Associates a name with a proposition and a (lambda term) proof of that proposition, e.g.:

- $thm1 = (\lambda P : *. (\lambda x : P.x)) : (\forall P : *. P \Rightarrow P)$

3.2 Extending Signatures with Binding Structures

When implementing a language with variable bindings, a lot of effort goes into the coding of operations like identification (i.e. relating applied occurrences to defining occurrences), substitution, copying, α -equivalence, β -reduction, matching, etc. . Section 5 mentions some of the approaches encountered in the literature.

In FOOLPROOF we take a different and novel approach. We introduce a new abstraction level, which is still independent of a particular signature, and which is geared towards handling all aspects of binding structures. This abstraction level provides *terms*, which correspond more or less to the usual notion, *items*, which correspond to defining occurrences of names and their properties, and *references*, which correspond to applied occurrences. This level is organized in such a way that all operations involving binding structures can be handled in a uniform and signature independent way. More precisely, we distinguish the following three levels

Level 0 Like before, this is the level which provides general tree operations by means of the abstract class `T_Node`.

Level 1 This level provides the notions of terms, items, and references. It is realized by means of two mutually recursive abstract classes, viz. `T_Term` and `T_Item`:

- The class `T_Term` is the abstract base class for terms. It consists of
 - A number (possibly 0) of *items*, which correspond to the binders of the term
 - A number (possibly 0) of *terms*, which are the subterms of the term
 - A number (possibly 0) of *references* to items (i.e. pointers from applied occurrences to defining occurrences)
 - A number (possibly 0) of *data* elements (meant for storing simple data like the digit sequence of a number)
- The class `T_Item` is the abstract base class for binders. It consists of
 - A name, which is the name of the binder
 - A number (possibly 0) of terms, which correspond to the properties of the binder

Level 2 This level is signature specific. It contains all the classes generated from a given signature

At first sight this three-level organization may seem somewhat strange, but it is the key to separating the general aspects of binding structures from signature dependent aspects. All the general aspects can be handled completely with level 1 operations. Also, one might wonder whether the structure for terms is not too general: most likely, a term node is either an internal node with subterms or a leaf with no subterms and with a single reference to an item. The answer is, that this extra generality does no harm, but actually makes algorithms simpler and more uniform.

Here are the class headers of the abstract classes `T_Term` and `T_Item`:

```

type
  T_Term =
  class(T_Node)
  public
    // override inherited T_Node methods
    function NodeCount: Integer; override;
    function GetNode(I: Integer): T_Node; override;
    procedure SetNode(I: Integer; ANode: T_Node); override;

    // local items
    function ItemCount: Integer; virtual;
    function GetItem(I: Integer): T_Item; virtual;
    procedure SetItem(I: Integer; AItem: T_Item); virtual;

    // subterms
    function TermCount: Integer; virtual;
    function GetTerm(I: Integer): T_Term; virtual;
    procedure SetTerm(I: Integer; ATerm: T_Term); virtual;

    // references
    function RefCount: Integer; virtual;
    function GetRef(I: Integer): T_Item; virtual;
    procedure SetRef(I: Integer; ARef: T_Item); virtual;

    // data fields
    function DataCount: Integer; virtual;
    function GetData(I: Integer): String; virtual;
    procedure SetData(I: Integer; AData: String); virtual;
  end;

  T_Item =
  class(T_Ctxt)
  protected
    FAux: T_Item; // scratch field, used for copying, alpha equality, etc.
    FName: string; // name of bound variable
    procedure SetName(AName: string); virtual;
  public
    constructor Create(AName: String);

    // override inherited T_Node methods
    function NodeCount: Integer; override;
    function GetNode(I: Integer): T_Node; override;
    procedure SetNode(I: Integer; ANode: T_Node); override;

    // subterms
    function GetTerm(I: Integer): T_Term; virtual;

```

```

procedure SetTerm(I: Integer; ATerm: T_Term); virtual;
function TermCount: Integer; virtual;

property Name: string read FName write SetName;
end;

```

To make things more concrete, we show in subsection 3.3 how a particular language with variable bindings can be encoded in a signature with binding structures and how this signature can be mapped to language specific classes. In subsection 3.4 we show as an example how a copy operation can be coded in terms of level 1 operations exclusively.

3.3 Example: Pure Type Systems (PTSs)

In this subsection we show how a particular language with variable bindings can be encoded as a signature. The language we consider is that of *Pure Type Systems*. Pure Type Systems (PTSs) are a parameterized family of typed lambda calculi. They provide a common framework which captures the essence of many notions from both functional programming languages (such as data types, functions, polymorphism, type constructors, inductive types, abstract datatypes, etc.) and logic (such as propositions, predicates, quantification, hypotheses, proofs, and theorems). These can all be described in a small kernel. The expressiveness of a PTS is determined by its parameters. For more information on PTSs we refer to [3, 4].

In publications from type theory the syntax of PTSs is usually given in the following style:

$T ::= S$	<i>sorts</i>
V	<i>variables</i>
$(T T)$	<i>application</i>
$(\lambda V : T.T)$	<i>λ-abstraction</i>
$(\Pi V : T.T)$	<i>Π-abstraction</i>
$(\delta V = T : T.T)$	<i>local definition</i>
$(T \rightarrow T)$	<i>\rightarrow - types</i>

where S is a set of *sorts* and V is a set of *variables*, disjoint from S .

We will not go into the exact meaning of the syntactic constructions, but just explain them with reference to the following example term:

$$(\delta polyid = (\lambda \alpha : *. (\lambda x : \alpha. x)) : (\Pi \alpha. \alpha \rightarrow \alpha)). \\
polyid \text{ nat } 3)$$

This term contains a local definition of a name *polyid*, which is associated with the term $(\lambda \alpha : *. (\lambda x : \alpha. x))$, and which has type $(\Pi \alpha. \alpha \rightarrow \alpha)$. The term $(\lambda \alpha : *. (\lambda x : \alpha. x))$ is a polymorphic function, which takes a type parameter α and returns the identity function $(\lambda x : \alpha. x)$ on that type. *polyid* is first applied to the type argument *nat* and subsequently to the term 3. By δ -conversion

(“unfolding the definition” of *polyid*) and β -reduction the application will first reduce to $(\lambda x : \text{nat. } x) 3$, and by another β -reduction to 3.

In order to encode PTS terms in FOOLPROOF, we first note that both the λ -term and the Π -term have a binder of the form $V : T$ (a “declaration”), and that the δ -term has a binder of the form $V = T : T$ (a “definition”). Since applied occurrences of names may refer both to names introduced by declarations and to names introduced by definitions, we introduce an item subsort `Binder` and two subsorts `Dec` and `Def`. `Dec` extends `Binder` with one term `T[ty:Term]`. `Def` extends `Binder` with two terms `T[te:Term, ty:Term]`. The λ -term `Lambda` extends the term sort `Term` with a declaration item `I[dec:Dec]` and a subterm `T[body:Term!]`. The exclamation mark indicates that the body is in scope of the item. The encodings of the Π -term and λ -term are similar. Finally, the applied occurrence of a name is encoded by the term subsort `Var`, which extends the term sort `Term` with a single reference `R[rf:Binder]` to a binder.

The complete encoding is shown in the panel on the left in the following screenshot from the class generator. The panel on the right shows the class header generated for the `Lambda` term.

```

// Hierarchy -----
_Node = _Data | _Term | _Item
_Data = Sort
_Term = Term
_Term = Const | Var | App | Lambda | Pi | Delta
_Item = Binder
_Binder = Dec | Def

// Definitions -----
Sort < _Data
Binder < _Item
Term < _Term
Dec = T[ty:Term] < Binder
Def = T[te:Term, ty:Term] < Binder
Const = D[val:Sort] < Term
Var = R[rf:Binder] < Term
App = T[func:Term, arg:Term] < Term
Lambda = I[dec:Dec] T[body:Term!] < Term
Pi = I[dec:Dec] T[body:Term!] < Term
Delta = I[def:Def] T[body:Term!] < Term
Arrow = T[left:Term, right:Term] < Term

TLambda =
class(TTerm)
protected
  Fdec: TDec;
  Fbody: TTerm;
public
  constructor Create(
    Adec: TDec;
    Abody: TTerm);
  destructor Destroy; override;

  class function Kind: T_NodeKind; override;
  class function SortCode: Integer; override;
  class function SortLab: T_SortLabel; override;

  class function ItemCount: Integer; override;
  function GetItem(
    I: Integer): _Item; override;
  procedure SetItem(
    I: Integer;
    AItem: _Item); override;

  class function TermCount: Integer; override;
  function GetTerm(
    I: Integer): _Term; override;
  procedure SetTerm(
    I: Integer;
    ATerm: _Term); override;

  property dec: TDec read Fdec write Fdec;
  property body: TTerm read Fbody write Fbody;
end;

```

The code generated for the methods `ItemCount`, `GetItem` and `SetItem` is as follows:

```
class function TLambda.ItemCount: Integer;
```

```

begin
  Result := 1;
end{ItemCount};

function TLambda.GetItem(
  I: Integer): _Item;
begin
  case I of
    0:
      Result := Fdec;
    else
      raise Exception.Create('Index error in TLambda.GetItem: ' + IntToStr(I))
      end;
end{GetItem};

procedure TLambda.SetItem(
  I: Integer;
  AItem: _Item);
begin
  case I of
    0:
      Fdec := AItem as TDec;
    else
      raise Exception.Create('Index error in TLambda.SetItem: ' + IntToStr(I))
      end;
end{SetItem};

```

The code generated for the methods `TermCount`, `GetTerm` and `SetTerm` is similar.

3.4 Coding operations in a signature independent way

In this section we show by means of an example how operations involving binding structures can be coded in terms of level 1 operations exclusively, i.e. independent of a particular signature. The operation we consider is that of copying a term. Since the notions of terms and items are defined by mutual recursion, we construct two functions: `CopyTerm` and `CopyItem`. These functions proceed by recursion on terms and items respectively. Special care has to be taken when encountering items in the copying process. These have to be copied as well, but we should remember that references to such items in the original should be replaced by references to copies of those items in the result. At first it seems that we have to maintain a list of pairs of (reference to original item, reference to copy) which has to be searched each time we encounter a reference. By a small coding trick this list can be avoided. Each item has a scratch reference field `FAux`, which is temporarily filled with a reference to a newly created copy of the item, and which is followed when encountering a reference to the original. Using this trick, the copying process can be performed in linear time. The code follows.

```

function CopyTerm(ATerm: T_Term): T_Term;
var
  I: Integer;

```

```

VOldItem, VNewItem: TItem;
VOldRef, VNewRef: TItem;
begin
  // create a result node of the same class as ATerm
  Result := ATerm.ClassType.Create; // virtual constructor

  // copy the items of ATerm to Result
  for I := 0 to ATerm.ItemCount - 1 do
  begin
    VOldItem := ATerm.GetItem(I);
    VNewItem := CopyItem(VOldItem);
    Result.SetItem(I, VNewItem);
    VOldItem.FAux := VNewItem; // temporarily link VOldItem to its copy VNewItem
  end;

  // copy the terms of ATerm to Result;
  for I := 0 to ATerm.TermCount - 1 do
    Result.SetTerm(I, CopyTerm(ATerm.GetTerm(I)));

  // copy the data elements of ATerm to Result
  for I := 0 to ATerm.DataCount - 1 do
    Result.SetData(I, ATerm.GetData(I));

  // copy the refs of ATerm to result, adjusting where necessary
  for I := 0 to ATerm.RefCount - 1 do
  begin
    VOldRef := ATerm.GetRef(I);
    if VOldRef.FAux <> nil // temporarily link to copy of VOldRef exists
    then VNewRef := VOldRef.FAux // use it
    else VNewRef := VOldRef;
    Result.SetRef(I, VNewRef);
  end;

  // finally, clear the temporary FAux links in the items of ATerm
  for I := 0 to ATerm.ItemCount - 1 do
    ATerm.GetItem(I).FAux := nil;
  end;

function CopyItem(AItem: T_Item): T_Item;
var
  I: Integer;
begin
  // create a result node of the same class as AItem
  Result := AItem.ClassType.Create; // virtual constructor

  // copy the name of AItem to Result
  Result.Name := AItem.Name;

  // copy the terms of AItem to Result
  for I := 0 to AItem.TermCount - 1 do
    Result.SetTerm(I, AItem.GetTerm(I));
  end;
end;

```

4 Current Status and Future Work

Phrased in contemporary terms: FOOLPROOF is currently undergoing a complete make-over. Various parts, such as the signature designer, the class generator, the structure editor and the scanner and parser generators have been developed as separate applications, which has resulted in several incompatibilities. Currently a redesign is going on, where the aforementioned tools are all based on a common set of components. Moreover, the tools will take the form of component library for the Delphi environment. This should make it possible to compose systems in a RAD (rapid application development) style. In this development style, applications are constructed in an integrated development environment (IDE) by dragging components from a palette, dropping them on a form, and setting their properties using property editors or component editors. The following example may give an idea of the intended development style.

Example

Assume we want to build a simple structure editor for, say, lambda terms. Given the FOOLPROOF toolkit and the Delphi IDE, we would take the following steps:

1. Use the signature designer to create a well-formed signature *LamSig* for lambda terms;
2. Use the class generator to generate from *LamSig* a collection of *LamSig*-specific classes and a factory class *LamFac*;
3. Define a textual representation *LamRep* for lambda terms;
4. Drop a structure editor component *StrEd* on a form, and:
 - Set its Signature property to *LamSig*;
 - Set its Factory property to *LamFac*;
5. Drop a structured term view component (e.g. nested text panels) *STV* on the form, connect it to the Term property of *StrEd*, and set its Representation property to *LamRep*;
6. Drop a context view component *CV* on the form, connect it to the Context property of *StrEd*, and set its Representation property to *LamRep*;

5 Related work

The notion of abstract syntax was introduced by McCarthy in the classic paper [22]. Abstract syntax specifications in the form of a signature are often used in work on algebraic specifications.

Various systems take some sort of syntax specification (usually a mixture of abstract and concrete syntax elements) as input and generate a set of type or

class definitions for representing abstract syntax trees and elements of syntax-directed programming environments. Some well-known examples are ASF+SDF [1], ASDL [7, 30], SableCC [13] and the JJTree component of the JavaCC compiler [17]. The problem of implementing both generic and specific syntax tree operations is handled in different ways, e.g. by generating signature-specific visitor classes or visitor combinators [18, 2], by using a single visitor and reflection [25], or - in a functional setting - by second-order polymorphism [21]. Some systems intended for generation of programming environments also offer facilities for specifying context dependencies by means of attribute grammars. A well-known example is the Cornell Synthesizer generator [26]. Another recent example is the JAstAdd system [16].

Several formal techniques exist for expressing binding structures. In [5] De Bruijn introduced the technique of *nameless dummies*, now usually referred to as *De Bruijn indices*. This technique is used in many type-theory based proof assistants, such as Coq [9] and LEGO [20]. A well-known technique for handling abstract syntax with binding operators is Higher-Order Abstract Syntax, described by Pfenning and Elliott in [24], but in essence already present in the work of Church [6]. A good description is given by Miller [23]. Gabbay, Pitts and Shinwell have developed a theory and a metalanguage (FreshML) for programming with bound names [14, 12]. In [27, 28] Talcott has developed a formal framework for formal systems with binding structures and rewriting, based on syntax trees with back-references from variable uses to their definitions. Back-references are expressed by means of access paths in the syntax tree. In [19] Lang et al. propose a different way of handling references in syntax trees and rewriting. The Cocktail program and proof assistant [10, 11, 8] implements abstract syntax trees with binding structures by means of separate Java classes for tree nodes, items (binders) and context. Back-references are implemented as object references to item objects. Generalizations of this technique have been applied in FOOLPROOF.

References

- [1] *ASF+SDF*, URL: <http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment>
- [2] Attali, Isabelle, a.o., *SmartTools: a Development Environment Generator based on XML Technologies*, LNCS 2027, p.355.
- [3] Barendregt, Henk, *Lambda Calculi with Types*, In S. Abramsky, Dov M. Gabbay and T.S.E. Maibaum (eds.), *Handbook of Logic in Computer Science*, Clarendon Press, 1992, pp. 117-309.
- [4] Barendregt, Henk, and Kees Hemerik, *Types in Lambda Calculi and Programming Languages*, Proceedings of ESOP'90 (European Symposium on Programming), Copenhagen, 1990.

- [5] Bruijn, N.G. de, *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem*. Indag. Math., 34(5):381-392, 1972.
- [6] Church, Alonzo, *A formulation of the simple theory of types*. Journal of Symbolic Logic, 5:56-68, 1940.
- [7] Christ-Neumann, M.-L. and H.-W. Schmidt. *ASDL—an object-oriented specification language for syntax-directed environments*. In European Software Eng. Conf. (ESEC '87), pages 77-85. Strasbourg, September 1987.
- [8] Cocktail, URL: <http://www.win.tue.nl/~michaelf/cocktail.php>
- [9] Coq, URL: <http://coq.inria.fr/>
- [10] Franssen, M., *Cocktail: A Tool for Deriving Correct Programs*, Proceedings of the 6th Workshop on Automated Reasoning, Bridging the Gap between Theory and Practice 6th-9th April, 1999, Edinburgh College of Art & Division of Informatics, University of Edinburgh.
- [11] Franssen, M.G.J., *Cocktail: A Tool for Deriving Correct Programs*, Ph.D. dissertation, Faculty of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, December 2000.
- [12] FreshML, URL: <http://www.cl.cam.ac.uk/~amp12/freshml/>
- [13] Gagnon, Etienne, *SableCC, An Object-oriented Compiler Framework*, School of Computer Science, McGill University, Montreal , March 1998.
- [14] Gabbay, M. J., and A.M. Pitts, *A New Approach to Abstract Syntax with Variable Binding*, Formal Aspects of Computing 13(2002)341-363, special issue in honour of Rod Burstall.
- [15] Hedin, G., *Reference attribute grammars*. In 2nd International Workshop on Attribute Grammars and their Applications, 1999.
- [16] Hedin, Görel, and Eva Magnusson, *JastAdd—a Java-based system for implementing front ends*, Electronic Notes in Theoretical Computer Science, Vol. 44 (2) (2001)
- [17] JavaCC compiler, URL: <https://javacc.dev.java.net/>
- [18] Kuipers, Tobias, and Joost M.W. Visser, *Object-oriented Tree Traversal with JJForester*, CWI-report SEN-R0041, ISSN 1386-369X.
- [19] Lang, Frédéric, a.o., *Addressed Term rewriting Systems*, res.rapp. RR 1999-30, INRIA, 1999.
- [20] LEGO, URL: <http://www.dcs.ed.ac.uk/home/lego/>

- [21] Lämmel, Ralf, and Simon Peyton-Jones, *Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming*, Procs. of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, 2003, pp. 26-37.
- [22] McCarthy, J., *Towards a Mathematical Science of Computation*, Proc. IFIP Cong. 1962, North Holland Pub. Co., Amsterdam, pp. 21-28.
- [23] Miller, Dale, *Abstract Syntax for Variable Binders: An overview*, Procs. of the First Intern. Conf. on Computational Logic, pp. 239-253, 2000, ISBN:3-540-67797-6.
- [24] Pfenning, F., and C. Elliott, *Higher-order abstract syntax*. In Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, pages 199–208. ACM Press, 1988.
- [25] Palsberg, Jens, and C. Barry Jay, *The Essence of the Visitor Pattern*. Procs of COMPSAC'98, 22nd Annual International Computer Software and Applications Conference. Vienna, Austria, August 1998.
- [26] Reps, T. and T. Teitelbaum, *The Synthesizer Generator*, Springer, 1989.
- [27] Talcott, C. L., *Binding structures*. In Vladimir Lifschitz, editor, Artificial Intelligence and Mathematical Theory of Computation. Academic Press, 1991. 11.
- [28] Talcott, C.L., *A Theory of Binding Structures and Applications to Rewriting*, TCS 112,1993 (short version in: Second International Conference on Algebraic Methodology and Software Technology, AMAST91, LNCS, Springer, 1991).
- [29] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck and C.H.A. Koster (eds.), *Report on the Algorithmic language ALGOL 68*, Numerische Mathematik, 14, 79-218 (1969); Springer-Verlag.
- [30] Wang, Daniel C. , Andrew W. Appel, Jeff L. Korn, and Chris S. Serra. *The Zephyr Abstract Syntax Description Language*, USENIX Conference on Domain-Specific Languages, Santa Barbara, October 15-17, 1997.