Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Specification and generation of a $lambdacalculus environment

A. van Deursen

Computer Science/Department of Software Technology

CS-R9233 1992

Specification and Generation of a λ-calculus Environment

Arie van Deursen*

CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands.
Email: arie@cwi.nl

Abstract

An algebraic specification of the λ-calculus is described. The specification covers valid substitutions, α, β, and η conversions, left-most reductions, and let-constructs for λ-definitions. The complete specification of the λ-calculus is given. By deriving parsers from signatures, and term rewriting systems from equations, tools are generated automatically from the specification. Combining these tools by means of a user interface description formalism, an environment for experimenting with the λ-calculus has been generated. The environment obtained in this way is presented.

1991 CR Categories: D.2.1, D.3.1, F.3.2, F.4.1
1991 Mathematics Subject Classification: 68N15, 68Q42, 68Q55
Key Words & Phrases: Algebraic specifications, λ-calculus, programming language semantics, programming environments, program generation.

1 Introduction

In Amsterdam, the GIPE[1] group has performed a lot of research on the automatic generation of programming environments from algebraic specifications of programming languages. Thus far, this research has resulted in an algebraic specification formalism called ASF+SDF[2] [BHK89], and an environment generator called the ASF+SDF system [Kli91, Hen91], capable of deriving parsers, term rewrite machines and syntax-directed editors from ASF+SDF specifications.

While reading the (very pleasant) book *Programming Language Theory and Implementation* by Michael Gordon [Gor88], the idea to specify the λ-calculus came to mind. Gordon devotes one chapter to the description of the λ-calculus. He needs another chapter to cover the implementation of a λ-calculus environment, consisting of tools to experiment with conversions, left-most reductions, let-constructs, and so on. In this paper we give an algebraic specification of the λ-calculus (which, thanks to the free syntax allowed in the

*Partial support has been received from the European Communities under ESPRIT project 2177 (Generation of Interactive Programming Environments II - GIPE II) and from the Netherlands Organization for Scientific Research – NWO, project *Incremental Program Generators*.

[1]GIPE is an acronym for Generation of Interactive Programming Environments.

[2]ASF+SDF emerged by combining ASF (Algebraic Specification Formalism) and SDF (Syntax Definition Formalism), (see Section 2.1).

```
sorts BOOL
functions
  true                        -> BOOL
  false                       -> BOOL
  and      BOOL # BOOL        -> BOOL
  or       BOOL # BOOL        -> BOOL
variables p                   -> BOOL
equations
  [1]     and(p, true )   = p
  [2]     and(p, false)   = false
  [3]     or( p, true )   = true
  [4]     or( p, false)   = p
```

Figure 1: Algebraic Specification of the Booleans

ASF+SDF formalism, closely resembles the description given by Gordon). Moreover, we show how this specification can be used to obtain a λ-calculus environment *for free*. This environment supports syntax-directed editing of λ-expressions, performing one-step α, β, and η conversions, left-most reductions, and let-constructs for introducing λ-definitions. It can be used for teaching purposes, to play with λ-definitions, or to get acquainted with the λ-calculus.

The intent of this paper is to show, by giving a simple but nontrivial example, how several existing ideas on algebraic specifications and environment generation [BHK89, Kli91, Koo92a, Koo92b, Hen91, HKR90, Wal91] can be combined. The algebraic specification of the λ-calculus itself, which also can be regarded as a term rewriting system describing the λ-calculus, seems to be new. Similar experiments to generate a λ-calculus environment would be possible using other formalisms in other systems: in terms of denotational semantics in the Programming System Generator PSG [BS86], in terms of attribute grammars in the Synthesizer Generator [RT89] or the Gandalf system [HN86], or in terms of natural semantics using TYPOL in the CENTAUR system [BCD+89].

2 Algebraic Specifications in ASF+SDF

To familiarize the reader with algebraic specifications, consider the specification of the Boolean data type in Fig. 1. This specification consists of a *signature* (the `sorts` and `functions` declarations), and a set of *equations*. From the signature we derive *closed terms* (such as `true`, `and(true,false)`, `and(true,or(false,true))`, ...), and *open terms* in which variables are allowed (like `and(true,p)`, `and(p,or(false,q))`, ...). The equations relate (open) terms, and induce an equivalence (more precisely, a congruence) relation on the closed terms. For instance, the terms `true`, `true and true`, `false or true` ... all are contained in the same equivalence class. We will not go into detail concerning algebraic specifications in general, but rather refer to [Wir90].

In this section we will introduce the main features of the formalism we use in this paper, the ASF+SDF formalism. It supports modularization, user-definable syntax, associative lists, and conditional equations. We will illustrate these features by presenting some examples: modules that are also used in the algebraic specification of the λ-calculus. The formal "meaning" of an ASF+SDF specification is its so-called *initial algebra* [GTW78]. Again, we will not discuss the formal aspects, but refer to [BHK89, Hen91] for the details.

```
module Booleans
imports Layout
exports
  sorts BOOL
  context-free syntax
    true               -> BOOL
    false              -> BOOL
    BOOL and BOOL      -> BOOL  {left}
    BOOL or BOOL       -> BOOL  {left}
    "(" BOOL ")"       -> BOOL  {bracket}
  variables  p         -> BOOL
priorities  and > or
equations
  [1] p and true  = p
  [2] p and false = false
  [3] p  or true  = true
  [4] p  or false = p
```

Figure 2: Module Booleans with free syntax

2.1 Signatures as Grammars

In Fig. 1, the "form" of the terms derived from the signature was rather limited; it had to be something like `and(true,false)`, or `push(S,E)`. Preferably, we would like to have more freedom in the equations, allowing us to write `true and false`, or `push E on S`. This is possible by replacing the signature by a description covering the concrete representation of terms, in addition to their abstract representation.

This idea has been exploited in the ASF+SDF formalism. ASF+SDF resulted from combining ASF and SDF. ASF (Algebraic Specification Formalism), which came first, is a "pure" algebraic specification formalism [BHK89]. In ASF+SDF, ASF signatures have been replaced by SDF definitions. SDF (Syntax Definition Formalism) is a formalism to define lexical, concrete, and abstract syntax at the same time [HHKR89]. The main idea of SDF is that a declaration of the form `BOOL and BOOL -> BOOL` is on the one hand read as a declaration of the abstract function and : BOOL $\times$ BOOL $\rightarrow$ BOOL and on the other hand as the declaration of a context-free grammar production $\langle$BOOL$\rangle$ ::= $\langle$BOOL$\rangle$ "and" $\langle$BOOL$\rangle$. The SDF reference manual [HHKR89] describes how abstract functions are derived uniquely from a concrete syntax.

As Fig. 2 shows, this permits the use of all kinds of nice syntax in the definition of the equations[3]. There is, however, a price to pay: we have to take care that the syntax is not ambiguous. Thus, in order to decide whether `true and false or true` should be read as `(true and false) or true` or as `true and (false or true)`, we have to add a `priorities` declaration, which in Fig. 2 favors the first alternative. Likewise, we indicate whether `true and true and true` is to be read as `(true and true) and true` or as `true and (true and true)`, which we do by declaring `and` to be `left` associative. The import of module Layout (third line of module Booleans) is explained in Section 2.2

[3] We will `CAPITALIZE` sorts, and use uncapitalized words for function names introduced in signatures. Non-alphanumeric or capitalized function names should be quoted

```
module Identifiers
imports Layout
exports
  sorts ID
  lexical syntax
    [a-z][a-zA-Z0-9\-']*      -> ID
  context-free syntax
    prime(ID)                 -> ID
  variables    Chars          -> CHAR+
equations
  [1]  prime( id(Chars) )  =  id(Chars "'")
```

Figure 3: Module Identifiers

```
module Layout
exports
  lexical syntax
    [ \t\n]                 -> LAYOUT
    "%%" ~[\n]* "\n"        -> LAYOUT
```

Figure 4: Module Layout, defining white space

2.2 Lexical Syntax

A `lexical syntax` section can be used to define basic lexical words like numbers (consisting of a non-zero digit followed by zero or more digits) or identifiers (one or more alphanumerical characters, starting with a letter, possibly including hyphens or primes). For the λ-calculus specification we need variables, which we define as the Identifiers of Fig. 3. Besides defining the identifiers, the signature of module Identifiers (Fig. 3) introduces a function `prime`. This function will be used in the λ-calculus specification to represent new variables `prime(V)`, `prime(prime(V))`, ... The equation of module Identifiers states that the result of applying a `prime` function to an identifier is the same as appending a `'` character to it. (Details concerning the built-in `CHAR` sort can be found in [HHKR89].)

As the reader may have noticed, module Booleans of Fig. 2 imports module Layout. It is needed to define "white space" in the equations. Module Layout (Fig. 4) uses the predefined sort `LAYOUT` [HHKR89, Chapter 4]. It defines spaces, tabs, or newlines as white space, and comment as lines starting with two percent signs (`%%`).

Finally, variable declarations are treated as declarations of lexical syntax. For example, if we wanted to have several variables `p`, `p1`, `p2` `p01`, ... in module Booleans (Fig. 2) we could have written `variables p[0-9]*  -> BOOL`, defining all these variables in a single declaration.

2.3 Associative Lists

ASF+SDF supports list functions and variables. List functions have a varying number of arguments, and list variables may range over any number of arguments of a list function. An example can be found in module IdSets (Fig. 5), defining sets of Identifiers[4]. The line

[4]Alternatively one could have defined a module sets *parameterized* by the element sort. We omitted this for simplicity.

```
module IdSets
imports Booleans Identifiers
exports
  sorts ISET
  context-free syntax
    "[" {ID ","}* "]"                  -> ISET
    ISET "-" ID                        -> ISET
    ISET "U" ISET                      -> ISET  {left}
    "member-of?"(ID, ISET)             -> BOOL
  variables
    [XY] -> ID         Es[123]  -> {ID ","}*
    Set  -> ISET

equations
  [1]   [Es1, X, Es2, X, Es3]      = [Es1, Es2, X, Es3]
  [2]   [Es1] U [Es2]              = [Es1, Es2]

  [3]   [Es1, X, Es2] - X          = [Es1, Es2] - X

  [4]   member-of?(X, Set) = false
        ============================
            Set - X =  Set

  [5]   member-of?(X, []) = false
  [6]   member-of?(X, [X, Es1]) = true

  [7]              X != Y
        ===========================================
        member-of?(X, [Y,Es1]) = member-of?(X,[Es1])
```

Figure 5: Module IdSets, using built-in lists

`"[" {ID ","}* "]"  -> ISET` defines terms like `[]`, `[E1]`, `[E1, E2]`, ... to be sets of 0, 1, 2, ... elements. The asterisk `*` indicates zero or more elements, while the comma is the concrete representation for the separators (note that by definition they are separators rather than terminators, see [HHKR89, Chapter 5]). The list notation is an abbreviation for the declaration of infinitely many functions `[...]`, each with a different number of arguments. If appropriate, instead of an asterisk indicating "zero or more", the plus character can be used to indicate "one or more". Lists without separators can be defined by omitting the curly braces and the separator (e.g., `FOO+`). More details on lists can be found in [HHKR89, Hen91].

List variables are needed to define equations over list functions. Module IdSets defines the variables `Es1`, `Es2`, and `Es3` of Fig. 5 as ranging over zero or more elements separated by commas. Equation `[1]` states that duplicate elements in sets are irrelevant. Equation `[2]` joins two sets; equations `[3]` and `[4]` remove one element from a set. Equations `[5]` to `[7]` define the membership function on sets.

```
module Lambda-syntax
imports Identifiers
exports
  sorts L-EXP
  context-free syntax
    ID                                  -> L-EXP
    lambda ID+ "." L-EXP                -> L-EXP
    L-EXP L-EXP                         -> L-EXP  {left}
    "(" L-EXP ")"                       -> L-EXP  {bracket}
  variables
    E[0-9']*                            -> L-EXP
    V[0-9']*                            -> ID
    V[0-9']*"+"                         -> ID+
priorities
  { lambda ID+ "." L-EXP  -> L-EXP }  <  { L-EXP L-EXP -> L-EXP }

equations
  [1]  lambda V+ V . E = lambda V+ . lambda V . E
```

Figure 6: Module Lambda-syntax

2.4 Conditional Equations

To obtain more flexibility in algebraic specifications, *conditional equations* can be used. In module IdSets (Fig. 5) we have seen examples of the use of a *positive* condition `[4]`, and a *negative* condition `[7]`. The idea of conditions is that the consequence (below the bar) only holds if the sides of the conditions (above the bar) can be proved equal or unequal. Negative conditions should be used with care, since they destroy a desirable model theoretic property of algebraic specifications (namely, the unique initial model property) [Kap88]. We will be careful, and use negative conditions in the sense of [MS88]. In doing so, the use of conditions becomes merely an abbreviation mechanism allowing more succinct specifications. In [DK92] it is shown how in practice any specification using conditions can be translated to an equivalent unconditional specification.

To facilitate the description of equations having an if-then-else like character, ASF+SDF supports the `otherwise:`[5] construct. An `otherwise:` equation only applies if no other equation is applicable. Using the `otherwise:`, the `member-of?` function can be defined as follows:

```
[6']    member-of?(X, [Es1, X, Es2])   = true
[5'7']  otherwise: member-of?(X, Set)  = false
```

Again, the `otherwise:` construct is merely an abbreviation, since it can always be rewritten to a number of positive conditional equations. For a discussion of the consequences of otherwise-equations, we refer to [DK92].

[5]In the current version of ASF+SDF, "otherwise:" is called "default". We prefer the more intuitive "otherwise:".

3 Specification of the λ-calculus

The λ-calculus originated in the 1930s by the work of A. Church as a theory to study functions [Chu41]. Ever since, it has inspired many other important developments, such as LISP (McCarthy), denotational semantics (Strachey), and functional programming (Henderson, Turner). By now, λ-calculus has grown into a major topic in programming language theory. It is used to study computation, design and semantics of programming languages, as well as specialized computer architectures [Gor88]. Barendregt [Bar84] is a solid treatise on the theory of the λ-calculus.

In this section we follow the description of Gordon [Gor88], replacing his (sometimes informal) definitions by modules of our algebraic specification.

3.1 Syntax

The module Lambda-syntax (Fig. 6) defines the syntax of the λ-calculus. The consecutive lines of the context-free syntax section define λ-expressions to be

1. variables (`x`, `y`, ...);
2. *abstractions* of the form `lambda x y . E` with *bound* variables `x`, `y` and *body* `E`. The `ID+` indicates that `lambda` should be followed by at least one bound variable.
3. Function *applications*: if `E1` and `E2` are λ-expressions, then so is `E1 E2`. It is intended to denote the result of applying function `E1` to an argument `E2`.

The `left` declaration indicates that function application is left-associative, i.e., `E1 E2 E3` means `((E1 E2) E3)`. The `priorities` declaration indicates that `lambda V . E1 E2` is to be read as `(lambda V . (E1 E2))` rather than as `((lambda V . E1) E2)` (i.e., the scope of the variable `V` extends as far to the right as possible). The single equation of the module states that `lambda V1 ... Vn . E` is just an abbreviation for `lambda V1 . ( ... .(lambda Vn . E))`. The brackets `"("` and `")"` can be used to override these conventions. In the variables section we have defined `V1`, `V2`, `E1`, `E2`, ... which we will use for arbitrary variables and λ-expressions respectively.

3.2 Substitutions

In Section 3.3 we will explain how a function abstraction `lambda x . E1` can be "called" with actual value `E2` by substituting the actual value `E2` for all occurrences of the formal parameter `x` in expression `E1`. Before doing so, we have to define the substitutions themselves (module Substitute, Fig. 7). A substitution of expression `E'` in expression `E` for all free occurrences of variable `V` is denoted by `E[E'/V]`. A variable is *free* in an expression, if it is not bound by a `lambda` abstraction. Free variables are defined precisely by the equations `[f1]`, `[f2]`, and `[f3]`, following [Bar84, p. 24]. When defining substitutions `E[E'/V]` care has to be taken that variables free in `E'` do not become bound in `E[E'/V]`. The specification does so, and follows the *valid* substitutions of Gordon[6] [Gor88, p. 73]. Consequently, the λ-expression `(lambda y . y x)[y/x]` is equal to `(lambda y' . y' y)`.

[6] It is possible to merge equations `[s5]` and `[s6]` into a single equation which omits the check whether `V'` is free in `E`, and always introduces a fresh variable, This, however, leads to the introduction of unnecessary fresh variables.

```
module Substitute
imports Booleans  Lambda-syntax IdSets
exports
  context-free syntax
    L-EXP "[" L-EXP "/" ID "]"                    -> L-EXP
    free-vars( L-EXP )                            -> ISET
    fresh-var( ID, L-EXP )                        -> ID

equations

  [s1]  V  [E/V] = E

  [s2]  V != V'
        =============
        V' [E/V] = V'

  [s3]  (E1 E2) [E/V] =  (E1[E/V]) (E2[E/V])

  [s4]  (lambda V . E1) [E/V] = lambda V . E1

  [s5]  V != V',   member-of?(V', free-vars(E)) = false
        ===============================================
        (lambda V' . E1) [E/V] = lambda V' . (E1[E/V])

  [s6]  V != V',  member-of?(V', free-vars(E)) = true,
        fresh-var(V', (E E1)) = V''
        =======================================================
        (lambda V'.E1) [E/V] = lambda V'' . ( E1[V''/V'][E/V] )


  [f1]  free-vars(V) = [V]
  [f2]  free-vars(E1 E2) = free-vars(E1) U free-vars(E2)
  [f3]  free-vars(lambda V . E) = free-vars(E) - V


  [g1]  member-of?(V, free-vars(E)) = true
        ========================================
        fresh-var(V, E) = fresh-var(prime(V), E)

  [g2]  otherwise: fresh-var(V, E) = V
```

Figure 7: Module Substitute for valid substitutions

```
module Convert
imports Substitute
exports
  context-free syntax
    alpha( L-EXP )                      -> L-EXP
    beta(  L-EXP )                      -> L-EXP
    eta(   L-EXP )                      -> L-EXP

equations
  [b1]  beta( (lambda V . E1) E2 ) = E1 [ E2/V ]
  [b2]  otherwise:  beta(E) = E

  [a1]  V' = fresh-var(V, E)
        =======================
        alpha( lambda V . E ) = lambda V' . (E[V'/V])

  [a2]  otherwise: alpha(E) = E


  [e1]  member-of?(V, free-vars(E)) = false
        ===================================
        eta( lambda V . E V ) = E

  [e2]  otherwise: eta(E) = E
```

Figure 8: Module for α, β, and η conversion

3.3 Conversions

Conversion rules are ways to transform one λ-expression into another. Module Convert (Fig. 8) defines the so-called α, β, and η-conversions. The most important one is β-conversion, which simulates evaluating a function: `(lambda V . E1) E2` is by β-conversion equal to `E1[E2/V]`, i.e., by replacing the formal parameter `V` by an actual value `E2` (equation `[b1]`). Functions that have the same form apart from the names of the bound variables denote the same function by α-conversion. Thus, `lambda V1 . E` can be replaced by `lambda V2 . (E[V2/V1])`, provided `V2` does not occur free in `E` (equation `[a1]`) [Bar84, p. 26]. By η-conversion, functions do not change when "putting a `lambda` around an existing function". For example, by η conversion `lambda x. (sin  x)` denotes the same function as `sin` itself (equation `[e1]`). If a λ-expression `E` is not an α, β, or η-convertible, then the `otherwise:` equations `[b2`, `[a2]`, and `[e2]` guarantee that functions `alpha`, `beta`, and `eta` are equal to the unchanged expression `E`.

3.4 Left-most reductions

In general, given a λ-expression `E`, it may be possible to apply β-conversion at several places called *redexes*. After repeated application of β-conversion a λ-expression in which no β-redex is available, the *normal form*, may be reached. Whether a normal form is found may depend on the order in which β-reduction is applied to the redexes. A strategy that always leads to a normal form (if it exists) is left-most reduction, which repeatedly reduces the left-most redex [Gor88, p.121].

```
module Reduce
imports Convert
exports
  context-free syntax
    lm-step( L-EXP )                -> L-EXP
    lm-red(  L-EXP )                -> L-EXP
    "is-beta-redex?" ( L-EXP )      -> BOOL
    "has-beta-redex?"( L-EXP )      -> BOOL

equations
  [i1]  is-beta-redex?( (lambda V . E1) E2 ) = true
  [i2]  otherwise:  is-beta-redex?(E) = false

  [h1]  has-beta-redex?(E1 E2) = is-beta-redex?(E1 E2) or
          has-beta-redex?(E1) or has-beta-redex?(E2)
  [h2]  has-beta-redex?( lambda V . E ) = has-beta-redex?(E)
  [h3]  has-beta-redex?(V) = false


  [l1]  is-beta-redex?(E1 E2) = true
        ============================
        lm-step(E1 E2) = beta(E1 E2)

  [l2]  is-beta-redex?(E1 E2) = false,  has-beta-redex?(E1) = true
        ========================================================
        lm-step(E1 E2) = lm-step(E1) E2

  [l3]  is-beta-redex?(E1 E2) = false,  has-beta-redex?(E1) = false
        =========================================================
        lm-step(E1 E2) = E1 lm-step(E2)

  [l4]  lm-step(lambda V . E) = lambda V . lm-step(E)
  [l5]  lm-step(V) = V


  [l6]  has-beta-redex?(E) = true
        ============================
        lm-red(E) = lm-red(lm-step(E))

  [l7]  otherwise: lm-red(E) = E
```

Figure 9: Module Reduce for left-most β reductions

```
module Let
imports Lambda-syntax Substitute
exports
  sorts DEF LET
  context-free syntax
    expand( L-EXP, LET )                -> L-EXP
    "(" ID ":" L-EXP ")"                -> DEF
    "(" let DEF+ ")"                    -> LET
    % empty %                           -> LET
  variables
    D[0-9']*"+"         -> DEF+
    D[0-9']*            -> DEF
equations
  [e0]  expand(E, ) = E
  [e1]  expand(E, (let (V:E'))) = E[E'/V]
  [e2]  expand(E, (let D+ D)) = expand(expand(E, (let D)), (let D+))
```

Figure 10: Module Let

Module Reduce (Fig. 9) defines left-most reductions on λ-expressions. The function `lm-step` yields the result of exactly one left-most step. It uses the auxiliary function `has-beta-redex?` to find the left-most redex. The function `lm-red` repeats left-most steps until the λ-expression does not change any more. If a λ-expression `E` has a normal form, then `lm-red(E)` is equal to that normal form. Module Reduce only defines left-most β-reduction. It can easily be extended to cover η-reduction as well, but we omitted this to keep our example simple.

3.5 λ-definitions

Besides being a language for reasoning about functions, the λ-calculus is used to represent all kinds of objects. Similar to the way natural numbers can be represented by the sets $\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \ldots$ in set theory, all kinds of objects can be represented by λ-expressions. Module Let (Fig. 10) introduces notation for such λ-definitions. For example, in the classical work of Church, a number N is represented by the normal form `lambda f x . f`^{N} `x`. A way to obtain this is by defining:

```
(let  (zero:  lambda f  x . x)
      (succ:  lambda n f x . n f (f x)))
```

According to these definitions, `succ (succ zero)` can be β-reduced to `lambda f x . f (f x)`. λ-definitions can be used in plain λ-expressions by replacing the name by the corresponding definition; this is specified by the `expand` function (Fig. 10). It would be fairly easy to extend the module `Let` to cover `letrec` structures for recursive functions as well (which can, of course, be rewritten to a `let` containing a fixed point operator, and an application of this operator to the function defined in the `letrec`). Again, we omitted this to keep the specification as small as possible.

3.6 Correctness

Let S be the specification as described in the previous sections, and let $S' = S\backslash$`[16]` be the specification without equation `[16]` of module Reduce, which defines how function `lm-red` repeatedly performs beta reductions in a left-most order.

The main argument when discussing the correctness of the specification is that each term of sort `L-EXP` in S' is equal to either a single (free) variable, an application, or an abstraction with one bound variable. In other words, the specification is *sufficiently-complete* with respect to the constructors `V`, `E1 E2`, and `lambda V . E` of sort `L-EXP`. This can be shown using *structural induction*, arguing that simple λ-expressions satisfy the property, and that the more complex ones maintain it. Equation `[1]` of module Lambda-syntax guarantees that any abstraction with several bound variables is equal to an expression with exactly one bound variable. Equations `[s1]` to `[s6]` of module Substitute eliminate terms of the form `E1[E2/V]`, distinguishing the three possible constructors of `E1`, and the occurrences of free variables in `E2`. Each of the functions for α, β, and η conversion also satisfy the property; non-convertible λ-expressions are covered by the `otherwise:` equations. The function `lm-step` of module Reduce again is sufficiently complete; cases are distinguished according to the constructor of the expression, and to whether the (sub)terms have β-redexes in case of application.

Taking equation `[16]` also into account, the sufficient completeness is lost; the function `lm-red` operating on a "looping" λ-expression like `(lambda x . x x)(lambda x . x x)` cannot be eliminated.

In a similar way, one can easily show that a term rewriting system obtained from the specification S' by orienting the equations from left to right is sufficiently complete, has unique normal forms, and does not contain infinite reductions.

4 The Generated Environment

4.1 Tools

The two most important tools that can be derived from an ASF+SDF specification by the ASF+SDF-system [Kli91, Hen91] are a *parser* and a *term rewriting machine*. A parser is a program that analyses the structure of a sentence according to a given grammar. From the SDF part of an ASF+SDF specification, a parser can be generated that is capable of parsing sentences and mapping these to the corresponding terms over the derived signature [HKR90]. A term rewriting system automatically evaluates terms by performing reductions according to the equations [Klo91]. Each equation is interpreted as a rewrite rule by giving it an orientation from left to right [BHK89]. For instance, in the module Booleans (Fig. 2) the term `true and false or true` can be rewritten according to equation `[1]` to `false or true`, which in turn can be rewritten according to rule `[4]` to `true`. Note that, in general, the term rewriting system obtained in this way can be incomplete with respect to the original algebraic specification; this is due to the fact that equations are interpreted only from left to right.

Both term rewriting machines and parsers can be "specialized" to one module. E.g., a parser restricted to module Lambda-syntax only knows how to parse λ-abstractions, variables, and applications, but does not know anything about substitutions. Likewise, term rewriting machines can be restricted to modules or even particular functions. In this

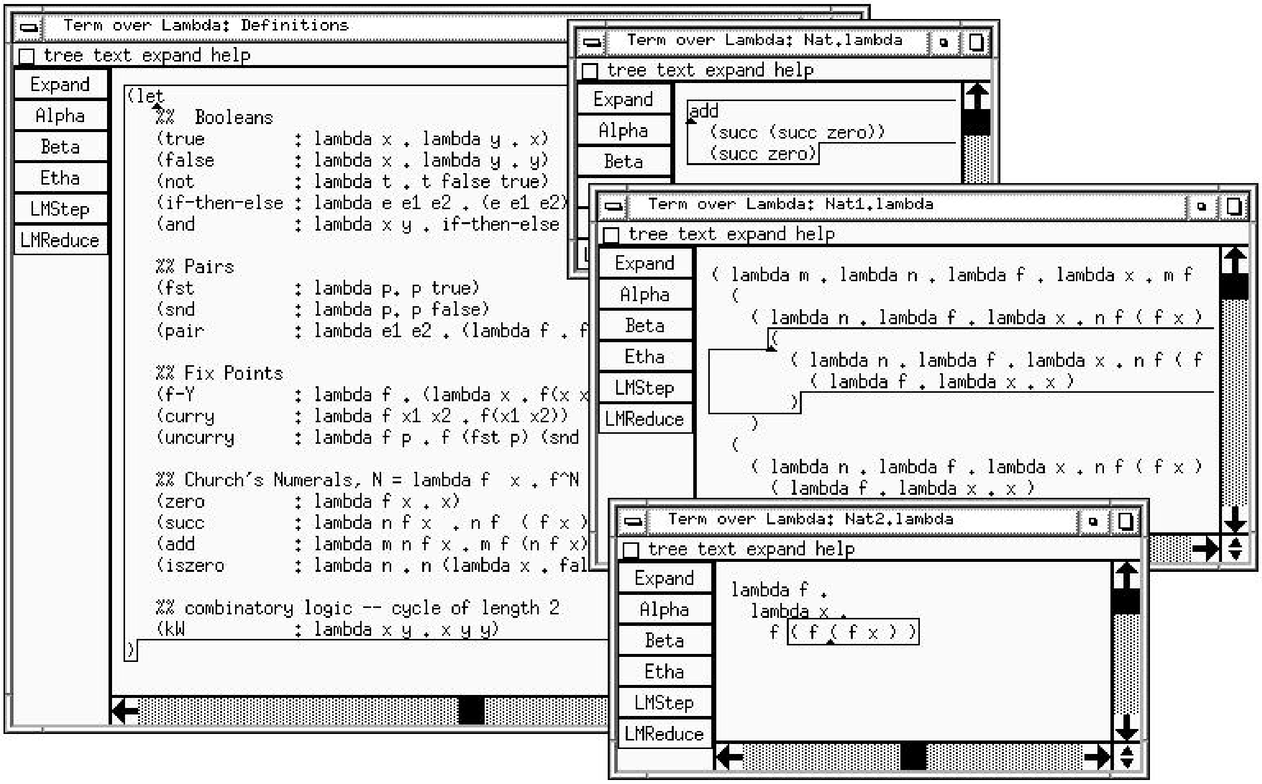

Figure 11: Example of the generated λ-calculus environment

way, numerous parsers and term rewriters can be derived from a single specification.

Given a parser, a *syntax-directed editor* can be derived [Koo92b], allowing both textual and structural editing on the tree obtained by parsing the text. Plain text editing is allowed within a *focus* designating one particular subtree. Structural editing enables focus movements and expansion of nonterminals according to the grammar.

Moreover, with the various user-interface events (such as mouse clicks, buttons pushed, or key stroke sequences) occurring in the editor, term rewriting actions can be associated. For example, the action "call `beta` with current focus as argument" can be attached to a button "Beta". A collection of such editors typically forms the basis for an inter-active, generated environment.

4.2 Layout of the Environment

From the modules presented in this paper the ASF+SDF system,[7] generates the λ-calculus environment showed in Fig. 11. We will discuss this generated environment first.

Fig. 11 displays four windows, each containing a syntax-directed editor. The largest window contains a `let` expression showing all kinds of λ-definitions that the user wishes to

[7]Actually, the ASF+SDF-system can do more than just generate the environment; a major part of it is the *meta-environment*, an interactive, incremental environment supporting editing, checking and testing specifications [Kli91].

experiment with. If desired, he or she can edit these definitions, add new ones, and so on. In the three small windows, λ-expressions can be manipulated. They can be edited, and the focus can be positioned on every subexpression. The subexpression in the focus can be changed by the various buttons attached to each λ-editor. There are buttons to α, β, or η-convert the focus, to perform one left-most reduction step, or to reduce the expression in the focus by left-most reduction to its λ normal form. The expand button replaces all occurrences in the focus of λ-defined identifiers by their corresponding definition given in the `let`-construct (from the big window in Fig. 11).

As an example of the practical use of such a generated environment, let us consider λ-definitions of numerals. Wadsworth [Wad80] gives several alternative λ-definitions for numbers, and proves all kinds of propositions about them. To develop some intuition concerning his definitions, one could edit the λ-definitions in the `let`-editor, and add:

```
(let     (cK     : lambda x y . x)
         (cI     : lambda x . x)
         (w-zero : cK cI)
         (w-succ : cK)
)
```

Now a term like `w-succ (w-succ w-zero)` can be entered in some λ-editor. In this editor it is possible to experiment with the Wadsworth numeral representations by clicking the various buttons with the focus at different positions, thus performing α, β, η-conversion, or left-most reduction (steps) on any desired subexpression. The intuition thus gained may help in proving, disproving, or conjecturing statements about Wadsworth's λ-definitions for numbers.

4.3 From Tools to Environment

In Section 4.1 we have seen that single tools can be derived from an algebraic specification, and in Section 4.2 we presented an example environment using tools derived from the λ-calculus specification. In this section we will briefly discuss how the exact layout and behavior of the environment can be defined. The environment has been generated from (1) the modules as described in Section 3, and (2) a user interface description. The user interface description of the λ-calculus environment is the topic of this section.

The basic desired functionality of the λ-calculus environment is given in Fig. 12. It summarizes the buttons of a typical editor for λ-expressions. For each button it gives (1) the name, (2) the desired behavior as a function of the subexpression currently under the focus, and (3) the module in the algebraic specification defining that function. The functionality indicates that the expression under the focus should be replaced by the result of rewriting the indicated function with the expression under the current focus as argument. For example, if the focus is on the λ-expression `lambda x . sin x`, clicking the Eta button will reduce the term `eta(lambda x . sin x)` according to the definitions of module Convert, and will replace the focus by `sin`, the result of the reduction. The five buttons for conversions and reductions only need the focus of the λ-calculus editor itself as input for the functions to be evaluated. The button Expand, by contrast, needs more. It assumes the existence of a second editor (the *Definitions*-editor) and assumes that it can access the expression under the latter's focus.

Button	Functionality	Module
Alpha	`alpha(` *Current-focus* `)`	`Convert`
Beta	`beta(` *Current-focus* `)`	`Convert`
Eta	`eta(` *Current-focus* `)`	`Convert`
LMStep	`lm-step(` *Current-focus* `)`	`Reduce`
LMReduce	`lm-red(` *Current-focus* `)`	`Reduce`
Expand	`expand(` *Current-focus* `,` *Definitions-focus* `)`	`Let`

Figure 12: Buttons for the λ-calculus environment

The user interface layout and behavior of the λ-environment of Section 4.2 and Fig. 11 was specified using a preliminary version of a user interface description formalism developed by Koorn [Koo92a]. The formalism contains, for instance, primitives to retrieve and update focus values in different editors, or to move the focus around in a particular editor. The description for the λ-calculus environment basically consists of the table with the 6 button definitions of Fig. 12, and covers about 25 lines. Given this description of the buttons for λ-calculus editors, and the nine modules of the algebraic specification of the λ-calculus presented in Sections 2 and 3, the λ-calculus environment of Fig. 11 was generated completely automatically using the ASF+SDF-system. No programming was needed.

5 Concluding Remarks

We have presented an algebraic specification of the λ-calculus. Thanks to the free syntax of the formalism, the specification closely resembles the description of the λ-calculus given by Gordon or Barendregt. Examples where this similarity is quite clear include the valid substitutions [Gor88, p.73], free variables [Bar84, p.24], or the syntax with its notational conventions [Gor88, p.62]. The formal definition has been used to obtain an environment for experimenting with the λ-calculus for free; this environment supports α, β, and η conversion, left-most reductions, and λ-definitions.

The ASF+SDF specification presented in this paper focuses on the basics of the λ-calculus. It is easy to extend the ASF+SDF specification to cover other reduction strategies, to extend to a typed λ-calculus, to translate to De Bruijn sequences [Bru72], to experiment with the explicit substitutions in the $\lambda\sigma$-calculus [ACCL90], and so on. Again, having specifications of these immediately provides one with tools to experiment with them.

Our specification shows that from a purely formal definition, inter-active environments can be generated. We realize that the λ-calculus is not a very complex example, which is relatively close to algebraic specification and term rewriting. Nevertheless, the simplicity of the example illustrates the ideas of the environment generator all the better. We hope (and expect) that in the future more and more compilers or environments will be derived as nice and easy as this λ-calculus environment.

Acknowledgements

The careful reading of Wilco Koorn, Jan Heering, and an anonymous CSN'92 referee significantly improved the paper. The idea to specify the λ-calculus originally came from Paul Hendriks.

References

[ACCL90] M. Abadi, L. Cardelli, P.-L. Currien, and J.-J. Lévy. Explicit substitutions. In *Proceedings of the 17th conference on the Principles of Programming Languages*, 1990.

[Bar84] H.P. Barendregt. *The Lambda Calculus; its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathatematics*. North-Holland, 1984.

[BCD+89] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, 1989. Appeared as *SIGPLAN Notices* 14(2).

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[Bru72] N. G. de Bruijn. Lambda caclulus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34:381–392, 1972.

[BS86] R. Bahlke and G. Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.

[Chu41] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[DK92] A. van Deursen and P. Klint. Default rules, positive conditions, and negative conditions in term rewriting systems. CWI, Amsterdam. Manuscript in preparation, 1992.

[Gor88] M.J.C. Gordon. *Programming Lanuage Theory and its Implementation*. Prentice-Hall, 1988.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, pages 80–149. Prentice-Hall, 1978.

[Hen91] P.R.H. Hendriks. *Implementation of Modular Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[HKR90] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990. Also in: *SIGPLAN Notices*, 24(7):179-191, 1989.

[HN86] A. N. Habermann and D. Notkin. Gandalf: software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, 1986.

[Kap88] S. Kaplan. Positive/negative conditional rewriting. In S. Kaplan and J.-P. Jouannaud, editors, *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, volume 308 of *Lecture Notes in Computer Science*, pages 129–143. Springer-Verlag, 1988.

[Kli91] P. Klint. A meta-environment for generating programming environments. In J.A. Bergstra and L.M.G. Feijs, editors, *Proceedings of the METEOR workshop on Methods Based on Formal Specification*, volume 490 of *Lecture Notes in Computer Science*, pages 105–124. Springer-Verlag, 1991.

[Klo91] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol II.* Oxford University Press, 1991. Also published as CWI report CS-R9073, Amsterdam, 1990.

[Koo92a] J.W.C. Koorn. Connecting semantic tools to a syntax-directed user-interface. Technical Report P92xx, University of Amsterdam, 1992.

[Koo92b] J.W.C. Koorn. GSE: A generic text and structure editor. Report P9202, University of Amsterdam, 1992.

[MS88] C.K. Mohan and M.K. Srivas. Conditional specifications with inequational assumptions. In S. Kaplan and J.-P. Jouannaud, editors, *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, volume 308 of *Lecture Notes in Computer Science*, pages 161–178. Springer-Verlag, 1988.

[RT89] T. Reps and T. Teitelbaum. *The Synthesizer Generator: a System for Constructing Language-Based Editors.* Springer-Verlag, 1989.

[Wad80] C.P. Wadsworth. Some unusual λ-calculus numeral systems. In J.P. Seldin and J.R. Hindly, editors, *To H.B. Curry: Essays on Combinatory Logic, λ-calculus, and Formalism*, pages 215–229. Academic Press, 1980.

[Wal91] H.R. Walters. *On Equal Terms, Implementing Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 675–789. Elsevier Science Publishers, 1990.