CONSTRUCTION AND REUSE OF FORMAL PROGRAM DEVELOPMENTS

J. Cazin P. Cros R. Jacquart M. Lemoine P. Michel
ONERA-CERT/DERI
2, Avenue Edouard Belin, 31055 Toulouse Cedex - FRANCE
Tel: (+33) 61 55 70 55
Fax: (+33) 61 55 71 12
Email: cazin@tls-cs.cert.fr

Introduction

A few years ago, formal developments were still considered an academic task. The main reasons were the size of addressable problems, the heaviness of notations, the lack of support tools, and the time overhead compared with usual empiric developments. These few arguments become more and more obsolete excepted for the last one: formalizing developments is still ressource over consuming. Nevertheless we can observe a growing interest of industrial companies for formal methods applied to real scale problems. Real experiences like compiler construction, user interface development of safety critical systems, security systems development are reported in [10]. Several reasons may explain this change of interest:

- formalizing developments aims at *mastering the correctness* of the developed objects. This is specially the case for safety critical systems. It has been observed that usual systematic testing methods become less and less applicable when the complexity of systems grows. Indeed, tests themselves must be formally specified to be credible, and this activity is also time consuming. In this case formal developments become competitive.
- it is in common agreement that *rigorous methods* are mandatory to be used to produce correct software. But it makes no use to define such methods without giving means to *control their application.* Such a control must dispose of a formal notion of development it can refer to when checking the correctness of a given development step.
- developments, when totally formalized, give a complete precise description of the set of steps leading from a specification to the corresponding program. If they are represented by terms which can be manipulated by higher order functions, they become *reusable* and applicable to different close problems. Then the overcost becomes acceptable.

The present paper is based on some results of two projects partially funded by the European ESPRIT-1 programme: ToolUse and REPLAY. The first one – **ToolUse** [2]– aimed at producing a software development environment offering a high level of parameterization. In this environment, software development methods are formally defined, and their application is checked. This implies the formalization of the developments themselves. A major issue of the project is the development language **DEVA** which will be overviewed in section 1. Examples of formal developments expressed with DEVA are given in section 2.

The second project – **REPLAY** [3] – aimed at studying reusability of formalized *developments*. This is an original point of view compared to the usual practice of reusing developed *components*. The approach will be shortly illustrated in section 3.

1. The development language DEVA

1.1. Technical basis

A major characteristics of the environment developed in the ToolUse project is to be *institution free*. That means notations and tools which are the basis of the environment are not devoted to a specific method or to particular specification and programming languages. As a consequence, the DEVA language has been defined as a very general notation to support a calculus on developments.

DEVA can be briefly described as a typed lambda-calculus. The main works which influenced its definition have been the Automath project [7], the Calculus of Constructions [4], and Intuitionistic Type Theory[6]. The general idea driving the definition is to give means to describe the developed objects, the development steps and the underlying rules as terms (the so-called *texts* introduced in section 1.2) and to use a general typing mechanism to control the correctness of these objects. Moreover developments, and underlying theories can be structured and organized using the *context* notion introduced in section 1.3.

In the current section, we will only introduce some basic elements which are useful for understanding the typing mechanism of DEVA and the examples of the next section. Nevertheless DEVA is not an extensive language and its complete syntax is presented in appendix A. A complete formal semantics can be found in [8]

The following syntactic conventions will be used to increase the readibility of the definition:

$x, y, z, x_i, y_i, z_i \ldots$	stand for variable symbols
$t, t_i, tt, tt_i \ldots$	are DEVA texts. More often tt_i is used to denote the type of the text t_i
$c, c_i \ldots$	are DEVA contexts

1.2. DEVA *texts*

The **texts** objects are the basic entities supported by DEVA. Each text is built up from other texts and is typed. Types are themselves typable texts. We dispose of a recursive system, supporting a type hierarchy whose the root is the untypable initial text **primal**.

Table 1 introduces some basic texts constructors: the texts are built from the initial text, or symbols defined in the current context (see below) using abstraction and application. Judgements are used to affirm that the type of a given text t_1 is t_2. They can be considered as formal comments, supporting partial specifications given by the user, and which will be checked by the DEVA evaluator. The validity of texts and the typing function which provides a basis for this mechanism are developed in section 1.

initial text	**primal**
symbol	x
abstraction	$[c \vdash t]$ or equivalently $\frac{c}{t}$
application	$t_1(t_2)$
judgement	$t_1 \therefore t_2$

Table 1 Some usual *text* constructors

1.3. DEVA *contexts*

The notion of **context** has been used in the previous section to define text abstraction. More generally, it allows to structure DEVA programs by expressing theories which these programs are based on: for example, basic mathematical theories, logics, algebraic data-types, specification and programming languages, and rules constituting a method.

Some basic context constructors are introduced in Table 2. Contexts are built up in a sequential way, starting from the empty context. A text declaration allows to introduce a new text symbol with its type. An anonymous declaration allows to introduce in a context a text which is unused in the following, but the type of which must be known. This is useful for defining, for example, the type profile of a function. A text definition gives a name to a given text, and can be considered as an abbreviation. In a similar way, a context definition allows to give a name to a locally defined context. The importation mechanism opens the imported context and gives access to all the declarations and definitions it contains.

The implicit definition allows to introduce in an abstraction a new text symbol with its type. The symbol plays the same role as any symbol introduced by a text declaration. Nevertheless, the corresponding argument will not require to be given when applying the abstraction : it will be synthesized from the other text arguments using a pattern matching mechanism. Such a feature avoids the user to give all the clerical typing details during applications.

empty context	**nilc**
sequential composition	$[\![c_1; c_2]\!]$
text declaration	$x : t$
anonymous text declaration	t
text definition	$x := t$
implicit definition	$x?t$
context definition	**part** $p := c$
context importation	**import** c

Table 2 Some of the main *context* constructors

The scoping rules are fairly simple: each symbol introduced at a given place in a sequence is usable in the following of the sequence. i.e. in the current context and in the innerly defined ones. The symbols defined in a given context are usable after the importation of that context and importation is transitive.

1.4. Informal examples

The first example shows how we can use DEVA to start the usual definition of natural numbers:

$$\begin{aligned}\textbf{part}\ \mathit{Naturals} := [\![\mathit{nat}\ &: \textbf{primal};\\ 0\ &: \mathit{nat};\\ \mathit{succ}\ &: [\mathit{nat} \vdash \mathit{nat}]\ ]\!]\end{aligned}$$

nat is introduced as a new symbol the type of which is **primal**. Then two constructors for naturals are declared: 0 the type of which is *nat* and *succ* which is declared as an abstraction from *nat* to *nat*.

If we want to develop some proofs from these definitions we can do that in a new context $Proofs - on - nats$:

$$\begin{aligned}\textbf{part}\ \mathit{Proofs} - \mathit{on} - \mathit{nats} := [\![&\textbf{import}\ \mathit{Naturals};\\ &\mathit{succ}(0) \therefore \mathit{nat}\ ]\!]\end{aligned}$$

In this context, we **import** the previously defined *Naturals* context, so we can use the definition of 0, *nat* and *succ*. Following the typing rules presented in the next section, we can apply the abstraction *succ* to any text of type *nat*, 0 for example. Doing so, we shall get a text the type of which is defined as the right hand side of the abstraction *succ*, i.e. *nat*. The judgement $succ(0) \therefore nat$ expresses this fact and will be successfully checked by the DEVA evaluator.

1.5. Validity and Typing rules

DEVA texts can be considered as λ–terms of a typed λ–calculus. A text is *well-formed* if it is built up using correct texts constructors introduced hereabove. A *well-formed* text is valid if it is made of valid texts and contexts and if it is *well-typed* (i.e. if some conditions, mainly applicability conditions and judgement validity,

hold). A context is valid if it is built from declarations and definitions of valid texts or contexts using correct contexts constructors.

The next three sections aim at giving some intuition of the validity rules for texts and contexts and of the typing mechanism supported by DEVA. The following notations will be used:

- $\mathbf{V}_t\{c, t\}$ is a predicate stating the validity of a text t in a given context c (i.e. the symbols used in t are declared or defined in c),
- $\mathbf{V}_c\{c_1, c_2\}$ states the validity of a context c_2 in the context of c_1. A context will be *totally* valid if it is valid in the empty context **nilc** (intuitively if it is self contained, and if each of its elements is valid),
- $\mathbb{T}(c, t)$ is the type of the text t in the context c.
- $\mathbb{E}_t\{c, t_1, t_2\}$ states the equivalence of two texts t_1 and t_2 in a given context c

Validity of *texts*

The *validity of texts* may be checked in a given *valid context.* Here follows some rules on which this validity is based:

- if a context is valid, then the text **primal** is valid in this context:

$$\frac{\mathbf{V}_c\{\mathbf{nilc}, c\}}{\mathbf{V}_t\{c, \mathbf{primal}\}}$$

- if a context c is valid, if another context c_1 is valid in c, and if t is a valid text in the context obtained by sequential composition of c and c_1, then the abstraction $[c_1 \vdash t]$ is valid in c:

$$\frac{\mathbf{V}_c\{\mathbf{nilc}, c\} \quad \mathbf{V}_c\{c, c_1\} \quad \mathbf{V}_t\{[\![c; c_1]\!], t\}}{\mathbf{V}_t\{c, [c_1 \vdash t]\}}$$

- the following rule ("applicability condition" rule) states the validity of the application of a text t_1 to a text t_2. The type of t_1 must be an abstraction, the first element of which is a declaration of a text equivalent to the type of t_2. Of course each element t_1, t_2 must be valid in c:

$$\frac{\mathbf{V}_c\{\mathbf{nilc}, c\} \quad \mathbf{V}_t\{c, t_1\} \quad \mathbf{V}_t\{c, t_2\} \quad \mathbb{E}_t\{c, t_1, [x : tt_2 \vdash t_3]\} \quad \mathbb{E}_t\{c, t_2, tt_2\}}{\mathbf{V}_t\{c, t_1(t_2)\}}$$

  This rule is used, for example, to decide the validity of *succ*(0) in the previous section.
- a judgement is valid if the type of the left hand side operand is equivalent to the right hand side. This is stated in the "judgement validity" rule:

$$\frac{\mathbf{V}_c\{\mathbf{nilc}, c\} \quad \mathbf{V}_t\{c, t_1\} \quad \mathbf{V}_t\{c, t_2\} \quad \mathbb{E}_t\{c, \mathbb{T}(c, t_1), t_2\}}{\mathbf{V}_t\{c, t_1 .\!:\! t_2\}}$$

Validity of *contexts*

Validity of contexts can be described by a a similar set of rules:

- the empty context is valid:

$$\frac{}{\mathbf{V}_c\{\mathbf{nilc}, \mathbf{nilc}\}}$$

- and it is valid in any valid context

$$\frac{\mathbf{V}_c\{\mathbf{nilc}, c\}}{\mathbf{V}_c\{c, \mathbf{nilc}\}}$$

- the sequential composition of two contexts c_1 and c_2 is valid in the context c, if c_1 and c_2 are respectively valid in c and in the sequential composition of c and c_1:

$$\frac{\mathbf{V}_c\{\mathbf{nilc}, c\} \quad \mathbf{V}_c\{c, c_1\} \quad \mathbf{V}_c\{[\![c; c_1]\!], c_2\}}{\mathbf{V}_c\{c, [\![c_1; c_2]\!]\}}$$

  Combined with the previous rule, this allows to check the validity of contexts from right to left.
- a declaration of a new symbol typed with a valid text is a valid context:

$$\frac{\mathbf{V}_c\{\mathbf{nilc}, c\} \quad \mathbf{V}_t\{c, t\}}{\mathbf{V}_c\{c, x : t\}}$$

  x does not need to be a new symbol; if it is still defined in c, the new definition hides the previous one
- the same for a new definition (i.e. an abbreviation):

$$\frac{\mathbf{V}_c\{\mathbf{nilc}, c\} \quad \mathbf{V}_t\{c, t\}}{\mathbf{V}_c\{c, x := t\}}$$

Typing rules

The following rules gives a definition by case of the typing function. Only some of the main cases are introduced:

- **primal** is never typable, whatever the context may be:

$$\mathbb{T}(c, \mathbf{primal}) \equiv \text{undef}$$

- any symbol x is untypable in the empty context. This case is encountered when an undeclared symbol is used in a text.

$$\mathbb{T}(\mathbf{nilc}, x) \equiv \text{undef}$$

- scoping rules for symbols are from right to left, so contexts are scanned in this way to find the last definition or declaration associated to a given symbol:

$$\mathbb{T}([\![c; y : t]\!], x) \equiv \begin{cases} t \text{ if } x \text{ and } y \text{ are the same symbol} \\ \mathbb{T}(c, x) \text{ otherwise} \end{cases}$$

$$\mathbb{T}([\![c; y := t]\!], x) \equiv \begin{cases} \mathbb{T}(c, t) \text{ if } x \text{ and } y \text{ are the same symbol} \\ \mathbb{T}(c, x) \text{ otherwise} \end{cases}$$

- imported contexts are unfolded when encountered:

$$\mathbb{T}([\![c; \textbf{import } c_1]\!], t) \equiv \mathbb{T}([\![c; c_1]\!], t)$$

- the two following cases give the type of an abstraction and of the application:

$$\mathbb{T}(c_1, [c_2 \vdash t]) \equiv [c_2 \vdash \mathbb{T}([\![c_1; c_2]\!], t)]$$
$$\mathbb{T}(c, t_1(t_2)) \equiv \mathbb{T}(c, t_1)(t_2)$$

Equivalence of two texts

The equivalence of 2 texts $\mathbb{E}_t\{c, t_1, t_2\}$ is defined as the transitive closure of reduction operations. These operations are mainly the usual β–reduction, the elimination of judgement, and the unfolding of the texts and contexts definitions used in t_1 and t_2. They are a generalization of the usual reduction rules of the λ–calculus and they will not be detailed here.

2. The formal expression of developments with DEVA

2.1. Formal developments in a transformational approach

The developments that will be manipulated in the following are based on a transformational approach. In such a framework, the developed objects constitute a continuum from specification to program, and one object is produced from previously existing ones by applying elementary transformations.

The transformations we shall use are based on the *fold/unfold* system, originally developed by Burstall and Darlington [1]. It allows to transform specifications expressed as a set of equations down to the level of a programming language, and it is specially tailored for developing programs in an applicative style.

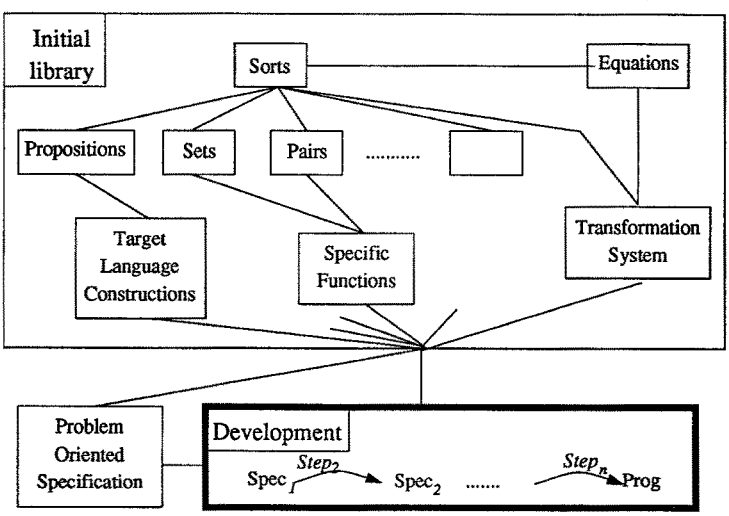


Figure 1 A schematic development and its context

To develop a program in such a framework, we have to dispose of an initial rigorous specification. In the following examples, this will be formulated in classical set theory. We do not worry about producing this early specification which is a matter of requirement engineering, out of the scope of our work.

The development does not start from scratch. Indeed an initial library contains a set of DEVA contexts expressing general theories or knowledge about development and their objects (such a library is enriched during succesive developments). This is summarized on figure 1 where boxes represent DEVA contexts and lines importation relationships:

- each data-type involved in the development (e.g. sets, pairs...) is DEVA-typed. Generic properties of these types are defined in a specific DEVA context *Sorts*
- the *Equations* context is generic and defines properties of equations between objects having the same type
- elementary data-types like *Sets*, *Pairs*, *propositions* are defined in specific contexts
- *Specific Functions* contains the definitions of functions which are in common use although not defined in basic data-types (like *Filter*, *Map*, aso...)
- *Target Language Constructions* allows to introduce in a progressive manner, the basic expressions which the developed program will be made of.

So, to complete a development, we have to import useful contexts from the initial library, to express the elements of the problem oriented specification as DEVA texts, and to express the development as a set of other DEVA texts $Step_1$, $Step_2$, ... $Step_n$. Each development step $Step_i$, when applied to one or several previously developed objects $Spec_{i-1}$, produces a new object $Spec_i$. This object is correct if it is well typed. The result of the last step is the program *Prog*.

2.2. A piece of development

The development presented hereafter is extracted from a case study which has been formally conducted with DEVA. The whole specification consists in the definition of 9 sets as presented in appendix B.

In the following, we shall focus on the development of the program computing the first set E^+ defined in this specification as:

$$E^+ \triangleq \{p \in \mathcal{P}(\mathcal{A}) \mid \langle p, + \rangle \in \mathcal{R}\}$$

this can be read as: "E^+ is defined as the set of elements p in the powerset of $\mathcal{A}$ such that the pair $\langle p, + \rangle$ belongs to the set $\mathcal{R}$".

The development corresponding to the computation of this set is given in a semi–formal way in figure 2, where the notations are as follows:

- $\frac{Exp_{in}}{Exp_{out}} Rule$ means that Exp_{out} is produced by applying an instance of the rule scheme *Rule* to a sub-expression of Exp_{in}, and unfolding Exp_{in} with the result. The sub-expression which is concerned is underlined in Exp_{in}.
- $x_{:t}$ denotes an element ot type t,
- A and R are respectively the types of the elements of $\mathcal{A}$ and $\mathcal{R}$. *SA* and *PSAR* abbreviate respectively *set(A)* and *pair(set(A),R)*,

$$E^+ \triangleq \{p \in \mathcal{P}(\mathcal{A}) \mid \langle p, + \rangle \in \mathcal{R}\}$$

$$\frac{}{E^+ \triangleq \left\{p_{:SA} \mid \underline{\langle p, + \rangle \in \mathcal{R}}\right\}} Int_{set}$$

$$\frac{}{E^+ \triangleq \left\{p_{:SA} \mid \exists y_{:PSAR}.y \in \mathcal{R} \wedge \underline{y = \langle p, + \rangle}\right\}} \exists_{intro}$$

$$\frac{}{E^+ \triangleq \left\{p_{:SA} \mid \exists y_{:PSAR}.y \in \mathcal{R} \wedge \underline{(p = \Pi_1(y) \wedge + = \Pi_2(y))}\right\}} \Pi_{intro}$$

$$\frac{}{E^+ \triangleq \left\{p_{:SA} \mid \exists y_{:PSAR}.y \in \mathcal{R} \wedge \underline{(+ = \Pi_2(y) \wedge p = \Pi_1(y))}\right\}} \wedge_{comm}$$

$$\frac{}{E^+ \triangleq \left\{p_{:SA} \mid \exists y_{:PSAR}.\underline{(y \in \mathcal{R} \wedge + = \Pi_2(y))} \wedge p = \Pi_1(y)\right\}} \wedge_{assoc}$$

$$\frac{}{E^+ \triangleq \underline{\{p_{:SA} \mid \exists y_{:PSAR}.y \in Filter(\lambda x.\Pi_2(x) = +, \mathcal{R}) \wedge p = \Pi_1(y)\}}} Filter_{intro}$$

$$\frac{}{E^+ \triangleq Map(\Pi_1, Filter(\lambda x.\Pi_2(x) = +, \mathcal{R}))} Map_{simpl}$$

Figure 2 The development related to E^+

- Π_1 and Π_2 are the projection operators associated to pairs

2.3. Expression using DEVA

To express formally this development with DEVA, we shall make an extensive use of the basic operation *"unfold"*. It allows to replace the left hand side of an equation by its right hand side inside the right hand side of another equation. This operation is defined in the *Transformation System* context of the initial library as a DEVA text:

$$unfold: \frac{s_1, s_2?sorts;\ x_1?s_1;\ x_2, y_2?s_2;\ f?[s_1 \vdash s_2]}{\dfrac{x_2 = y_2; \qquad x_1 = f(x_2)}{x_1 = f(y_2)}}$$

To use it, we have only to apply it to two arguments whose the DEVA types are $x_2 = y_2$, and $x_1 = f(x_2)$. The other parameters of the abstraction (s_1, s_2, ... f) will be synthesized, and we shall get a new text of type $x_1 = f(y_2)$.

Let us consider now the first development step:

$$\frac{E^+ \triangleq \{p \in \mathcal{P}(\mathcal{A}) \mid \langle p, + \rangle \in \mathcal{R}\}}{E^+ \triangleq \{p_{:SA} \mid \langle p, + \rangle \in \mathcal{R}\}} Int_{set}$$

It expresses that starting from a specification of E^+, we decide to keep the information $p \in \mathcal{P}(\mathcal{A})$ only as a typing information (i.e. we represent the powerset $\mathcal{P}(A)$ in intention). To express it with DEVA, we must represent:

- a first level to formalize the sets of the form $\{x \in S \mid P(x)\}$. This will be done with the constructor:

$$S_0: [s?sort;\ S: set(s);\ P: [s \vdash prop] \vdash set(s)]$$

- the second level where we decide of a representation in extension or in intention, and where we keep typing information: $\{x_{:t} \mid P(x)\}$ will be represented with another constructor:

$$S_1 : [s?sort;\ P : [s \vdash prop] \vdash set(s)]$$

- the rule representing the design decision to represent a set in intention:

$$intset : \left[\frac{s?sort;\ S : set(s);\ P : [s \vdash prop]}{S_0(S,P) = S_1(P)}\right]$$

- the initial definition of E^+ with the corresponding type:

$$E^+ : set(a);$$
$$defE^+ : E^+ = S_0(pow(\mathcal{A}), [p : set(a) \vdash isin(\langle p, + \rangle, \mathcal{R})])$$

- the first step consists then in unfolding the whole right hand side of the equation using the *intset* rule:

$$step_1 := unfold\big(intset(pow(\mathcal{A}), [p : set(a) \vdash isin(\langle p, + \rangle, \mathcal{R})]), defE^+\big)$$
$$\therefore\ E^+ = S_1([p : set(a) \vdash isin(\langle p, + \rangle, \mathcal{R})])$$

and so, the transformed definition for E^+ is the type of the resulting application (this is expressed in the right hand side of the judgement).

The whole development can be expressed in the same way. Each elementary step builds a *text*, the type of which is a new refined definition for E^+, which will be itself the type of input argument for the next step. The rules which are to be used for the whole development of E^+ are given in appendix C.

$$\begin{aligned}
&[... \\
&step_1 := unfold\big(intset(...), defE^+\big) \\
&\qquad \therefore\ E^+ = S_1([p : set(a) \vdash isin(\langle p, + \rangle, \mathcal{R})]); \\
&step_2 := unfold(exintro(...), step_1) \\
&\qquad \therefore\ E^+ = S_1([p : set(a) \vdash exists([y : pair(set(a), res) \\
&\qquad\qquad \vdash and(isin(y, \mathcal{R}), (y = \langle p, + \rangle))])]); \\
&step_3 := unfold(piintro(...), step_2) \\
&\qquad \therefore\ E^+ = S_1(...); \\
&\qquad \vdots \\
&step_7 := unfold(mapsimpl(...), step_6) \\
&\qquad \therefore\ E^+ = map(pi1, filter([x : pair(set(a), res) \vdash pi_2(x) = +], \mathcal{R})); \\
&\qquad ...]
\end{aligned}$$

The development steps $step_i$ which are detailed hereabove are definitions, i.e. abbreviations used in the next steps. So in each $step_i$ is accumulated the part of the development starting from $step_1$. Therefore, the last step $step_7$ is the formal object which represents the complete development:

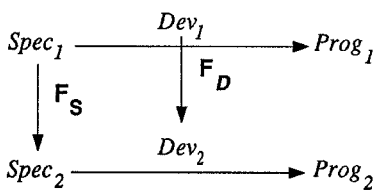
$$unfold(mapsimpl(...),$$
$$unfold(filterintro(...),$$
$$unfold(andassoc(...),$$
$$unfold(andcomm(...),$$
$$unfold(piintro(...),$$
$$unfold(exintro(...),$$
$$unfold(intset(...), defE^{+}))))))$$

3. Reusing formal developments

3.1. General idea

An interesting application of formally described developments is to reuse them to deal with problems close to each other. The intuitive idea to address this topic is to consider the development as a function which applies to a specification and gives the corresponding program. This function may be modified by a higher order function so as to take into account the differences between the two specifications. Then the resulting new function may be applied to the new specification.

This can be summarized on the following schema:



where $\mathbf{F}_S$ expresses the changes on the initial specification, and $\mathbf{F}_D$ the related changes in the development. In an ideal situation, it should be nice to give only the new specification $Spec_2$, and the main changes in the development (e.g. "change the set representation from list to ordered lists"), and to have the two functions automatically defined.

The work we present hereafter is far less ambitious and gives only means to formally express the changes from Dev_1 to Dev_2. The idea is not to automate the development of $\mathbf{F}_S$ and $\mathbf{F}_D$ functions, but rather to give means to express $\mathbf{F}_D$ in such a way that significant checked parts of the first development can be reused. In any case, the development of these functions only depends on the skill of the developer.

The approach that we propose is based on an *abstraction/instantiation* mechanism which can be applied at the level of DEVA texts or DEVA contexts as detailed hereafter.

3.2. Abstraction/instantiation on texts

When we have constructed a first development as a DEVA text, the idea is to get first a more general object, by abstracting on a part of this first development. The resulting text and its use, do not differ from a usual abstraction we could have written with DEVA. The main difference is that we are not enforced to write down first the most abstract development text before instantiate it on the first problem then on the second one.

The *a posteriori abstraction* can be defined as follows:

$$ABS(t_1, t_2) \equiv \left[x : tt_1 \vdash t_{2[x/t_1]}\right]$$

where x is a new symbol free in t_2

tt_1 is the type of t_1

$t_{2[x/t_1]}$ is the substitution of x for t_1 in t_2

The use of this simple operation to address reuse issues can be illustrated on another part of the example given in appendix B. Indeed the second set to compute is defined as

$$E^- \triangleq \{m \in \mathcal{P}(\mathcal{A}) \mid \langle m, - \rangle \in \mathcal{R}\}$$

We can notice that this specification can be deduced from the specification of E^+ only by changing $+$ to $-$, and the corresponding definition $defE^+$ to $defE^-$. This can be simply expressed in DEVA using two ABS abstractions as follows:

First we introduce the type and the definition of the new set:

$$E^- : set(a);$$
$$defE^- := E^- = S_0(pow(\mathcal{A}), [m : set(a) \vdash isin(\langle m, - \rangle, \mathcal{R})]);$$

Then we produce the abstract development from the one of E^+:

$$devabs := ABS(+, ABS(defE^+, step_7));$$

and finally, we apply the result of this abstraction to the new values characterizing E^-:

$$step_7' := devabs(-, defE^-)$$
$$\therefore E^- = map(pi1, filter([x : pair(set(a), res) \vdash pi_2(x) = -], \mathcal{R}));$$

And so we get in one step a correct development for the second set.

3.3. Abstraction/instantiation on contexts

The same mechanism can be applied to context themselves containing the trace of an existing development. It can be defined as follows:

$$CABS(t, c) \equiv [\![x : tt;\ c_{[x/t]}]\!]$$

where x is a new symbol free in c

tt is the type of t

$c_{[x/t]}$ is the substitution of x for t in c

To address reuse issues with this operation, we need also context application and renaming operations which are not detailed here. Their syntax is given in appendix A and their semantics in [8]. In this case, if the first development has been conducted in a first DEVA context:

part $ctxdev_{E^+} :=$
$[\![E^+ : set(a);$
$defE^+ : E^+ = S_0(pow(\mathcal{A}), [p : set(a) \vdash isin(\langle p, + \rangle, \mathcal{R})]);$
...
$step_7 := unfold(mapsimpl(...), step_6)$
$\therefore E^+ = map(pi1, filter([x : pair(set(a), res) \vdash pi_2(x) = +], \mathcal{R}))$
$]\!]$

we can then abstract this first context on +, then apply it to − :

$$\textbf{part } ctxdev_{E^-} := CABS(+, ctxdev_{E^+})(-)$$

If we import the resulting context as such, old definitions will be hidden by the new ones obtained after abstraction and instantiation:

; **import** $ctxdev_{E^-}$
; $E^+ \therefore set(a)$
; $defE^+ \therefore E^+ = S_0(pow(\mathcal{A}), [p : set(a) \vdash isin(\langle p, - \rangle, \mathcal{R})]);$
; ...
; $step_7 \therefore E^+ = map(pi1, filter([x : pair(set(a), res) \vdash pi_2(x = -)], \mathcal{R}))$

Moreover, we can use renaming to avoid the confusion introduced by hiding:

import $ctxdev_{E^-}(E^+ =: E^-, defE^+ =: defE^-, step_7 =: step_7')$;
$E^- \therefore set(a);$
$defE^- \therefore E^+ = S_0(pow(\mathcal{A}), [p : set(a) \vdash isin(\langle p, - \rangle, \mathcal{R})]);$
...
$step_7' \therefore E^- = map(pi1, filter([x : pair(set(a), res) \vdash pi_2(x = -)], \mathcal{R}))$

The difference with abstraction on texts is that this last solution gives access to each elementary results of the first development after adaptation to the characteristics of the second one. So any number of development steps can be reused from the first development and constitute potential choice points to apply new rules.

CONCLUSION

The DEVA language overviewed in this paper is a powerful notation which allows to express the semantics of development in a totally formal way. The examples given in this paper are very sketchy, but DEVA has proved to be able to support

various calculus like the Bird-Meertens'one [9] and significant parts of methods like VDM [5] and JSP/JSD.

The DEVA language is supported by an evaluator which can be considered as a high level type-checker. It forbids the user to cheat with the application of a method, enforcing him to formally describe all the developments steps he follows.

What may look like a heavy task in a first shot development is, on the contrary, fruitful in the context of reuse, and allows to get in a few steps programs correct wrt their specifications. This is made possible by the explicit manipulation of objects representing developments. Experiments conducted in the REPLAY project have shown that we can get other benefits of these formal objects. For example, applying them abstract interpretation and complexity analysis technics has proved to make it possible to master operational properties of the objects during the developments, and so to take them into account in the design decisions.

References

[1] R.M. Burstall and J.Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, 1977.

[2] J. Cazin, R. Jacquart, M. Lemoine, P. Maurice, and P. Michel. Method driven programming. In *IFIP 89*, 89.

[3] J. Cazin, R. Jacquart, M. Lemoine, and P. Michel. Manipulation of formal developments expressed in deva. In K.H. Bennett, editor, *Software Engineering Environments*. Ellis Horwood, 89.

[4] T. Coquand and G. Huet. Constructions: a higher order proof system for mechanizing mathematics. In *EUROCAL 85*, 1985.

[5] Ch. Lafontaine. Formalization of the VDM reification in the DEVA meta-calculus. In *Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods*, 1990.

[6] P. Martin-Lof. Constructive mathematics and computer programming. In Hoare and Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 167–184. Prentice Hall, 85.

[7] R.P. Nederpelt. *An Approach to Theorem Proving on the Basis of a Typed Lambda Calculus*. Springer Verlag, LNCS 87, 1980.

[8] M. Sintzoff, M. Weber, Ph. de Groote, and J. Cazin. Definition 1.1 of the generic development language deva. Technical report, Esprit, 89.

[9] M. Weber. Formalization of the Bird-Meertens algorithmic calculus in the DEVA meta-calculus. In *Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods*, 1990.

[10]J.C.P. Woodcock. Formal techniques and operational specifications. *Software Engineering Notes*, 14(5), 89.

Appendix A The complete DEVA syntax

- texts constructors

initial text	**primal**
symbol	x
abstraction	$[c \vdash t]$ or $\frac{c}{t}$
application	$t_1(t_2)$
judgement	$t_1 \therefore t_2$
sum	$[t_1 \mid t_2 \mid ... \mid t_n]$
product	$[x_1 := t_1, ..., x_n := t_n]$

- specific texts constructors for control expression

sequential composition	$t_1 \circ t_2$
case distinction	**case** t
iteration	**loop** t

- context constructors

empty context	**nilc**
sequential composition	$[\![c_1; c_2]\!]$
text declaration	$x : t$
text definition	$x := t$
implicit definition	$x?t$
context definition	**part** $p := c$
context importation	**import** c
context application	$c(t)$
symbol renaming	$c(x =: y)$
symbol hiding	$c \backslash x$

Appendix B A problem–oriented specification

The developments presented in section 2 and 3 are extracted from a case study formally conducted as a whole with DEVA. The developed program must process the result of experiments in biological area (typing of human cells and serum). The problem oriented specification of this program amounts to the definition of 9 sets which must be computed by the final program:

$$E^+ \triangleq \{p \in \mathcal{P}(\mathcal{A}) \mid \langle p,+\rangle \in \mathcal{R}\}$$
$$E^- \triangleq \{m \in \mathcal{P}(\mathcal{A}) \mid \langle m,-\rangle \in \mathcal{R}\}$$
$$H \triangleq \{p \in E^+ \mid \forall x \in p.\exists m \in E^-.x \in m\}$$
$$Cred \triangleq \{a \in \mathcal{A} \mid Cred(a,\mathcal{R})\}$$
$$E'^+ \triangleq \{p \in E^+ \mid p \cap \mathcal{A}_{cred} = \emptyset \Rightarrow \neg(p \in H)\}$$
$$E'^- \triangleq \{m \in E^- \mid \forall p \in H.m \cap p \cap \mathcal{A}_{cred} = \emptyset\}$$
$$H' \triangleq \left\{p \in E'^+ \mid \forall x \in p.\exists m \in E'^-.x \in m\right\}$$
$$E''^+ \triangleq E'^+ - H'$$
$$Result \triangleq \left\{ q \in \mathcal{P}(\mathcal{A}) \mid \exists p \in E''^+.q = p - \bigcup_{m \in E'^-} m \right\}$$

The target language in the case study is Common Lisp, and the resulting program is about 800 lines long.

The reuse aspects illustrated in this paper are based on the analogy between E^+ and E^- expressions. The formal development has been used as well to address reuse issues like change of data type representation for a given set, change of algorithm or local optimizations.

Appendix C Development rules used in section 2

$$intset\ :\ \frac{s?sort;\ S:set;\ P:[s \vdash prop]}{S_0(S,P) = S_1(P)}$$

$$exintro\ :\ \frac{s_1, s_2?sort;\ x:s_1;\ S:set(s_2);\ f:[s_1 \vdash s_2]}{isin(f(x),S) = exists([y:s_2 \vdash and(isin(y,S),(y=f(x)))])}$$

$$piintro\ :\ \frac{s_1, s_2?sort;\ p:pair(s_1,s_2);\ x_1:s_1;\ x_2 s_2}{(p=\langle x_1,x_2\rangle) = and((x_1 = pi_1(p)),(x_2 = pi_2(p)))}$$

$$andcomm\ :\ [p,q:prop \vdash\ and(p,q) = and(q,p)]$$

$$andassoc\ :\ [p,q,r:prop\ \vdash\ and(p,and(q,r)) = and(and(p,q),r)]$$

$$filterintro\ :\ \frac{s?sort;\ P:[s \vdash prop];\ x:s;\ S:set(s)}{and(isin(x,S),P(x)) = isin(x,filter(P,S))}$$

$$mapintro\ :\ \frac{s_1, s_2?sort;\ S:set(s_1);\ y:s_2;\ f:[s_1 \vdash s_2]}{exists([x:s_1 \vdash and(isin(x,S),(y=f(x)))]) = isin(y,map(f,S))}$$

$$memelim\ :\ [s?sort;\ S:set(s)\ \vdash S1([x:s \vdash isin(x,S)]) = S]$$