Facilitating Modular Property-Preserving Extensions of Programming Languages

Claus Pahl
Department of Computer Science, University College Cork
Cork, Ireland

Abstract

We will explore an approach to modular programming language descriptions and extensions in a denotational style. Based on a language core, language features are added stepwise on the core. Language features can be described separated from each other in a self-contained, orthogonal way. We present an extension semantics framework consisting of mechanisms to adapt semantics of a basic language to new structural requirements in an extended language preserving the behaviour of programs of the basic language. Common templates of extension are provided. These can be collected in extension libraries accessible to and extendible by language designers. Mechanisms to extend these libraries are provided. A notation for describing language features embedding these semantics extensions is presented.

1 Introduction

When we are faced with the task of formally describing a programming language, we could start from a simple language core, which exhibits the main principles of the language, and add a number of language features onto that core in a **modular**, **property preserving** way. This process of extending languages is similar to refinement approaches in software development. A language core and language features can be specified separately. Features will be specified as parameterised operators, thus allowing them to be applied to a number of basic languages. Features are specified as independent constructs only referred to by a formal parameter interface to the languages which should be extended by them. Languages and their extensions are presented in a denotational semantics style. Properties are expressed in an equational metalanguage, represented by equational theories.

Peter Mosses [1] criticises the lack of modularity in traditional denotational semantics. David Schmidt [2, 3] suggests structuring semantics into units (semantics algebras) and to base extensions on these algebras. These ideas will be picked up and implemented in a framework called extension semantics. The approach will allow us to define languages in a formal and modular way. The general purpose of this framework of **extension semantics** is to support the exploration of and experimentation with semantics. The development of a new, or the description of an existing full-blown language is not our aim, but features and their interaction can be studied with our approach. Danvy and Hatcliffe have used the notion of a *programming language workbench* [4]. The development of a whole language is an industrial process, but certain prototypical aspects can be extracted a priori in order to be analysed formally in depth with appropriate tools. Sample case studies will illustrate these ideas.

The **preservation of behaviour** of programs is one of the central issues in language extensions. It guarantees that programs written in the basic language can be reused in the extended language, i.e. they are be executable. Furthermore, they are also executed in a way similar to the execution as a program of the base language. The similarity is formally captured by a notion of observability. We will provide a number of common templates for extensions which guarantee behaviour preservation. Behaviour preservation is of particular importance since it guarantees *orthogonality* of the new feature with the basic language. A new feature can be added to a language such that there are no conflicts and, thus, the semantics of the basic constructs will remain. Extending by simply adding elements is easy and should not cause difficulties, but consider redefinitions: these might have side-effects on other constructs which use the redefined one. We will concentrate in particular on this issue.

We are aiming at a usable framework for the extension of languages defined in the style of classical denotational semantics. We will provide a notation based on a library of predefined extension operators, called templates. The application of these operators will guarantee the preservation of properties. We will provide a mathematical framework which allows the language designer to extend the given library by adding new templates easily. The notation supports the extension by providing a variety of powerful constructs. This allows the language designer to focus on the specification of new features rather than on some tedious extension and reformulation of existing elements.

Our framework formalises ideas for modular development and presentation of denotational semantics suggested by David Schmidt [2, 3] and Peter Mosses [1] and also provides notational support. A comprehensive framework for language engineering, i.e. the extension, restriction and combination of languages or language aspects, including a formal theory and an applicable notational and methodological support, does not exists at the moment. Certain aspects are currently under investigation in the project DESTIJL [5].

Section 2 introduces basic ideas of language extension together with our formal framework. A more comprehensive explanation of the framework and more properties are presented in section 3. Section 4 introduces equational theories as a representation of a simple equational language used to express properties of the semantics. Effects of semantics extensions on these theories are investigated. This is followed by a sample extension in section 5. Some specific problems such as operator combinations are addressed in section 6. Section 7 presents three case studies which illustrate the area of applications of our approach. We conclude with comparing similar approaches and some summarising thoughts.

2 Extension of Semantics and Preservation of Properties

In this section, we will introduce the basic ideas of our approach illustrated by the `csh`-language. The Unix C-shell `csh` is a shell command interpreter with a C-like syntax [6]. The interactive shell reads commands from the terminal. The input is parsed into words. Two languages for a csh-like language shall be investigated: the first, called *core* (figure 1), describes the principle idea of the language of processing the commands $echo$ and $pipe$ on input- and output-streams (essentially strings in this core). The second language, called *extension*, includes basic elements such as success values for commands. Other features that might be included are for example environments to store values in variables. This extended language shall be developed in the remainder using our approach of extension semantics.

LANGUAGE DESCRIPTION
syntax
Cmd ::= *'echo' StrExpr* |
'pipe' Cmd Cmd |
StrExpr ::= *Chr**
semantic algebras
$domain\ String$
$domain\ Stream \triangleq String$
semantic functions
$\mathbf{S} : StrExpr \to String$
$\mathbf{S}[\![se]\!] \triangleq$ if $se =$ $<>$ then $<>$ else $hd\ se$ ^ $\mathbf{S}[\![tl\ se]\!]$
$\mathbf{C} : Cmd \to Stream \to Stream$
$\mathbf{C}[\![echo(se)]\!]i \triangleq \mathbf{S}[\![se]\!]$
$\mathbf{C}[\![pipe(c_1, c_2)]\!]i \triangleq$ let $o_1 = \mathbf{C}[\![c_1]\!]i$ in $\mathbf{C}[\![c_2]\!]o_1$
END

Figure 1: Core

Semantics of a language can be structured into semantic algebras. This approach was already proposed by Schmidt [2]. Structured extensions of the language will be carried out based on this semantical structuring, rather than the classical viewpoint of obtaining structure by syntax.

Definition 2.1 *A* **semantic algebra** *is a heterogeneous algebra consisting of*

- *semantic domains $D_1, \ldots, D_n$ (possibly compound)*
- *functions on these domains*

Semantic domains are sets. A typical semantic algebra would for example consist of a set of states and operations, such as an update on states (with the usual substitution or override semantics). Let Alg be the collection of all semantic algebras. With each semantic algebra A we have associated a **signature** $sig(A)$, sometimes denoted Σ. The symbols for semantic entities are used as syntactic elements. An equational metalanguage based on Σ-terms will be introduced later (see section 4). For extensions to be applied to a core language, we might want to identify a **domain of interest** $S \in \{D_1, \ldots, D_n\}$. It serves to denote that part of the basic language semantics that shall be redefined. The redefinition of a domain, which represents the structure of some language feature, is not always aimed at, but – as already mentioned – is the complicated case and, thus, addressed here. Based on the domain S, a domain extension will be carried out.

Modularity is a key objective of our approach, thus complete features shall be added onto the basic language step by step. As in similar refinement approaches, the preservation of properties is essential. Behaviour (of programs) is the property we are interested in when languages are extended. Some of the principal ideas of extension shall be illustrated by using the $echo$ command of the `csh` $core$ language as a very primitive program whose behaviour has to be preserved in an extension of the notion of streams:

$$\mathbf{C}[\![echo(se)]\!]i \triangleq \mathbf{S}[\![se]\!]$$

$echo(se)$ is a phrase of the syntactic domain Cmd. The semantic function **C** maps from Cmd to $Stream \to Stream$. An input stream i is mapped to an output stream when $echo(se)$ is executed. Let us now consider an extension of the $core$ language. Commands shall work on a configuration consisting of streams and file systems, the extension is expressed by an operator T[1].

$$T : Stream \mapsto Stream \times FileSys$$

The semantic function for $echo$ has to be adapted

$$\_^* : \mathbf{C}[\![echo(se)]\!] \mapsto \mathbf{C}[\![echo(se)]\!]^*$$

such that $\mathbf{C}[\![echo(se)]\!]^* : TStream \to TStream$. A $Stream \times FileSys$-based algebra certainly preserves the behaviour of the $echo$ command on $Stream$, if the following diagram commutes:

$$\begin{array}{ccc} Stream \times FileSys & \xrightarrow{\mathbf{C}[\![echo(se)]\!]^*} & Stream \times FileSys \\ \uparrow \phi_{Stream} & & \uparrow \phi_{Stream} \\ Stream & \xrightarrow{\mathbf{C}[\![echo(se)]\!]} & Stream \end{array}$$

where ϕ_{Stream} maps streams into the extension $TStream$. A stream $s \in Stream$ is mapped to a pair (s, f) consisting of the stream and some element f representing the file system. An injection shall be used to extend the domain $Stream$:

$$\text{INJECT } Stream \text{ INTO } Stream \times FileSys$$

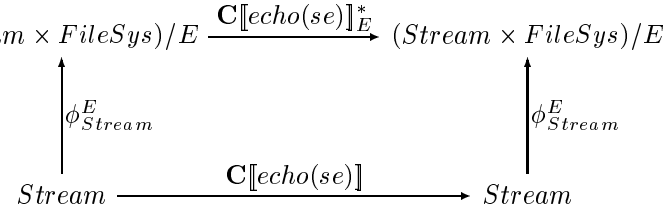
Streams s shall be refined to equivalence classes $[(s, f)]_E$, expressed by a mapping r, where $[(s, f)]_E \triangleq \{(a, b) \mid a = s\} \subseteq Stream \times FileSys$. We associate equivalence classes instead of single pairs (s, f) to each stream s. We use an equivalence E on $Stream \times FileSys$ to define relevant or observable behaviour. Two pairs are equivalent if their

[1] If not mentioned otherwise, T will be the identity on all domains except the domain of interest, here $Stream$.

$Stream$-parts are equal. This equivalence defines a notion of *observability*. We can now relax our formulation of behaviour preservation. Extended functions are only expected to behave correctly with respect to these classes, i.e.

$$\phi^E_{Stream}(\mathbf{C}[\![echo(se)]\!](i))\ E\ \mathbf{C}[\![echo(se)]\!]^*(\phi^E_{Stream}(i))$$

or expressed diagrammatically:



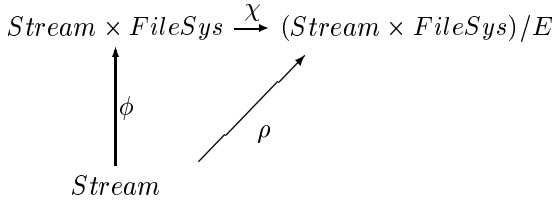
The mapping ϕ^E_{Stream} maps to equivalence classes instead of mapping to single pairs. It compares to a retrieve function (e.g. known from VDM [7]). The retrieve function would map all elements from one equivalence class to the associated element in $Stream$. We prefer the representation with explicit equivalence classes since the equivalence will be used later on in the construction of appropriate templates.

Equivalence relations on the extension are used to define the property that certain elements are similar with respect to some observation. In particular, we will express behaviour preservation requirements by equivalence relations. A quotient algebra is obtained from applying the equivalence to a given algebra. The equivalence has to be a congruence. A **congruence relation** E on a set S is an equivalence relation which satisfies the substitution property with respect to functions f on that set (let $a, b \in S$): $aEb \Rightarrow f(a)Ef(b)$. Let A be an algebra and E a congruence relation on the sets D_i in A. The **quotient algebra** A/E is defined by

$$D_i/E \triangleq \{[a]_E \mid a \in D_i\} \text{ and } f_E([a]_E) \triangleq [f(a)]_E$$

We can say that the quotient structure is an *abstract interpretation* abstracting from specific implementation details of the extension. The quotient structure focuses on those properties which express the behaviour preservation. Note, that properties of the basic algebra are preserved. The abstract interpretation is a homomorphic image of the extended algebra.

Let us look at the existence of the mappings involved now. The canonical mapping $c : (s, f) \mapsto [(s, f)]_E$ does always exist, $p : s \mapsto (s, f)$ can be constructed from c and r. These set-based mappings r, c, p can be extended to morphisms ρ, ϕ, π on algebras by using the constructs congruence E and quotient algebra $State/E$:



ρ is normally denoted as ϕ^E. ϕ is a morphism which preserves behaviour and χ is a canonical morphism onto the quotient algebra. Assume ϕ is surjective as well (we can consider working on the redefined part B of an extension B'). In this case, we can construct an onto-mapping ϕ^{-1}.

Proposition 2.1 *If ρ and χ are morphisms, then there exists a behaviour preserving morphism $\phi : A \to B$ such that $\chi \circ \phi = \rho$ for $\chi : B \to B/E$ and $\rho : A \to B/E$.*

Proof: See e.g. [8]. □

A special morphism on algebras shall be introduced which formalises the idea of behaviour preservation.

Definition 2.2 *Let A and B be semantic algebras. Let B/E be a quotient algebra with respect to a congruence relation E on B. Let $T : A_i \mapsto B_i$ be an association between semantic domains. An* **extension morphism** *$\phi : A \to B$ is called* **behaviour preserving***, if*

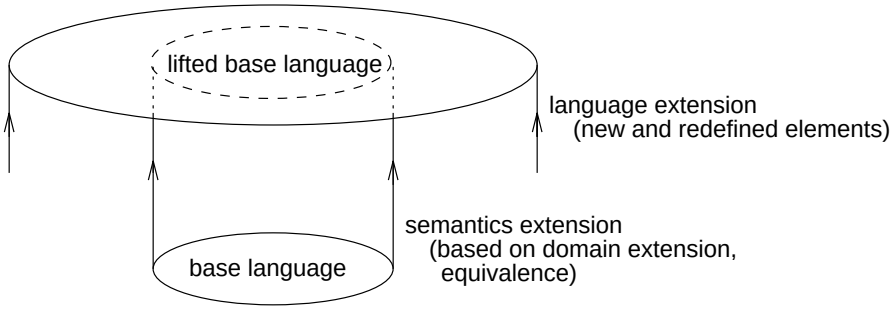
- *ϕ is a sorted set of mappings:* $\phi \triangleq \{\phi_i : A_i \to B_i \mid A_i \text{ is a domain in } A\}$,
- *ϕ preserves the operation behaviour with respect to the congruence E, e.g. for* $op : A_i \to A_j$ *and* $x \in A_i$ *it holds* $\phi_j^E(op(x)) =_E op_E^*(\phi_i^E(x))$.

The equality $=_E$ is the equality on the equivalence classes. Behaviour preservation requires only equivalence, not identity of results on the extended level (cf. the commuting diagrams above). As we will see later, this construction will allow us to define orthogonal language extensions.

Remark 2.1 *The lifting of functions* $\_^*$, ϕ, *and the equivalence E depend on each other. These dependencies will be further investigated when templates are introduced in section 3. Behaviour preserving extensions of equational theories are analysed in section 4. Usually associated with a set of operations is a set of operation combinators (higher order operators). The definition of semantic algebras and behaviour preservation given above does not involve these combinators. We will argue that the separated treatment of composition operators will lead to an improvement of the presented extension techniques (see section 6).*

3 The Construction of Extension Morphisms and Templates

A notation for defining languages has been implicitly introduced in the previous section in Figure 1. This section contains notation and formal background for extensions of languages. Before we dive into technical details, let us sketch the outline of our extension approach. A *language description* consists of *syntax, semantic entities* and *semantic functions*. We see semantics as the main aspect on which a language definition should be based and also on which a language extension should be based. Thus, we will provide two major constructs for language definition: *semantic algebras* and *language descriptions*, and also two constructs for extension: *semantics extensions* on semantic algebras and *language extensions* on language descriptions.



A language extension is the construct which serves to specify a new language feature in its semantics, and also in its syntactical interface. A language extension is a self-contained specification of a language feature. Before any new elements are added or existing elements are redefined, the semantics extension can be applied. Based on the domain construction and an equivalence relation specifying how behaviour has to be preserved, the semantics extension lifts the semantics of the basic language such that the domain constructions are adapted and behaviour is preserved. This is a complete redefinition of existing language elements. The semantics extension is a construct which does not depend on or refers to the basic language semantics directly. *Extension templates* will be introduced to facilitate the definition of such semantics extensions. The language extensions are based on semantics extensions. The latter would be executed first when applied to a basic language. Then additions and further redefinitions on the lifted base language can be carried out.

This two-tiered approach is one of the essential characteristics of our approach. It provides strong support for the extension and allows the language designer to concentrate on the description of the new feature.

3.1 Semantics Extensions

A language description based semantically on semantic algebras can be extended to another language defined by other semantic algebras. This involves defining the extending morphism, describing the relevant behaviour and proving the that the defined morphism is behaviour preserving with respect to the relevant behaviour expressed by the equivalence. This procedure shall be facilitated for the language designer by providing operators working on semantic algebras which adapt the basic semantic algebra to some extended domain construction. These operators will take as argument some basic information about the intended extension. They will allow the construction of an extension morphism from this information.

A *semantics extension* is an operator on semantic algebras. The operator has to satisfy some properties (congruence, behaviour preservation) in order to allow us to construct a behaviour preserving morphism (see definition 2.2). A semantics extension shall allow us to provide a morphism on algebras which lifts a basic algebra according to a domain extension, e.g. from $Stream$ to $Stream \times FileSys$ including a lifting of existing functions. New functionality, e.g. operations using $FileSys$, is not added by this construct.

Definition 3.1 *A* **semantics extension** *from semantic algebra A to B is a 5-tuple $(T, \phi^E, E, \_^*, d)$ where*

- *$T : A \mapsto B$ is a type constructor on semantic algebras,*
- *$\phi^E : Alg \to Alg/E$ is a mapping on algebras,*
- *E is a binary relation on $T(S)$ if S is domain of interest of A,*
- *$\_^*$ is a function lifting from $f : A \to B$ to $f^* : TA \to TB$,*
- *d is a collection of default values for each equivalence class in $T(S)/E$.*

For all domains D not equal to S, E_D is assumed to be the equality (i.e. a congruence) and ϕ_D is assumed to be the identity mapping. Using the result from section 2 we can for instance derive ϕ from ϕ^E and d.

3.2 Extension Templates

In this subsection, some properties shall be investigated allowing the derivation of behaviour preserving semantics extensions from a reduced amount of information in some particular situations. It will be proven for each particular case, that using a canonical construction for the equivalence E and for the extended functions f^*, an extension morphism can be derived such that behaviour preservation is guaranteed. In general, all elements of a semantics extension are necessary to define an extension on semantic algebras. A semantics extension provides a frame to derive behaviour preserving semantics extensions. The process of constructing semantics extensions shall be looked at: properties simplifying this process shall be elaborated and a library of common semantics extensions, called templates, shall be introduced.

Definition 3.2 *Let $S \mapsto T(S)$ be the domain extension. Based on the given type extension T of a semantics extension $(T, \phi^E, E, \_^*, d)$, we can use standard constructions for the remaining elements. A* **template** *allows the generation of a behaviour preserving semantics extension based on predefined constructions for type extensions.*

Examples of these templates based on type extensions are injection $S \mapsto S \times T$ (see figure 2) or indexing $S \mapsto (I \to S)$. These templates can form a *library of extension operators* for the language designer. A sample application of this template was already presented in section 2 to inject streams into a product of streams and file systems. The extension morphism ϕ is constructed from ϕ^E and a default value r_0. The equivalence E expresses that only the first component, e.g. a stream, is relevant for behaviour preservation.

The idea behind templates is to reduce the amount of information that a language designer has to give as a parameter for an extension. The operator T (domain type extension) is essential, but then, based on the type constructor, we can start using standard constructions.

It will be shown first that the function lifting $\_^*$ can always be defined in a way such that behaviour preservation is guaranteed. Once the equivalence classes exist, the defaults can be obtained just by selecting one for each class.

EXTENSION TEMPLATE

INJECT S INTO $S \times R$ = (

$T : S \mapsto S \times R$

$\_^* : f_i \overset{c}{\mapsto} f_i^*$

$(s,r)E(s',r')$ iff $s = s'$

$\phi_S^E(s) = [(s,r_0)]_E$

(s,r_0) for each $[(s,r_0)]$

)

Figure 2: Extension template INJECT

Proposition 3.1 *The operator for function lifting* $\_^*$ *for a function* $f : A \to B$ *and* $a \in A$

$$f^*(\phi_A(a)) := \phi_B(f(a))$$

guarantees behaviour preservation for a semantics extension $(T, \phi^E, E, \_^*, d)$.

Proof: The definition of f^* is partial, but total on required subset of TA. This definition guarantees $f^*(\phi_A(a)) = \phi_B(f(a))$, i.e. equality as a particular equivalence (more than the behaviour preservation criterion requires). □

Common templates based on a particular domain extension are the following (we will give $E, \phi^E, \_^*, d_E$ for each T on a domain S):

1. $T : S \mapsto S \times R$ **:** $(s,r)E(s',r')$ iff $s = s'$; $\phi^E : s \mapsto [(s,r)]_E$; (s,r_0) is default for $[(s,r_0)]$
2. $T : S \mapsto S + R$ **:** xEx' iff $x = x'$; $\phi^E : s \mapsto [s]$; s is default for $[s]$
3. $T : S \mapsto (I \to S)$ **:** tEt' iff $\forall i \in I.t(i) = t'(i)$; $\phi^E : s \mapsto t$ with $t(i) = s$ for all $i \in I$; f_0 with $f_0(i) = s$ is default for $[f_0]$
4. $T : S \mapsto \mathcal{P}(S)$ **:** pEp' iff $p = p'$; $\phi^E : s \mapsto [\{s\}]$; $\{s\}$ is default for $[\{s\}]$

The first case is an injection INJECT which has already been used in section 2 (see also figure 2 where the template is represented in our extension notation). All templates follow classical ways of injecting or embedding simple values into more complex domain constructions. For a more thorough investigation, more general constructions such as indexed products could be investigated.

The templates so far are purely set-based, but using the function lifting, the congruence property for the equivalence can be shown and, thus, the set-based mappings can be extended to homomorphisms.

Proposition 3.2 *Given a template* $(T, \phi^E, E, \_^*, d)$ *based on a set-based mapping* ϕ^E*, we can*

- *obtain a set-based mapping* $\phi : S \to TS$ *and*
- *extend* ϕ *to homomorphism on algebras where* S *is mapped to* TS *and each function* f *to* f^*.

Proof: ϕ is obtained by using the defaults d for each equivalence class. The mapping on set and function symbols (T and $\_^*$) is a signature morphism. The function lifting guarantees the substitution property (i.e. congruences) regarding the signature morphism. □

All suggested templates allow us to derive behaviour preserving semantics extensions. The templates have to satisfy a number of constraints: the relation E is an equivalence, there is a default value for each equivalence class.

Proposition 3.3 *The templates allow us to obtain behaviour preserving semantics extensions.*

Proof: The well-formedness of the template components for the four templates is easy to see, classical injections or embeddings are used. □

Some knowledge from universal algebra is required for the construction of E and ϕ. A retrieval function $\psi : TS \to S$ can be seen as the central function. This could be used to derive an equivalence E on TS. Given a function $\psi : TS \to S$, we define the **equivalence relation** $E(\psi)$ **of** ψ on TS by

$$E(\psi) = \{(s, s') \in TS \times TS \mid \psi(s) = \psi(s')\}$$

We have chosen to separate E and ϕ (the inverse of ψ) to distinguish the activities of **partitioning** (obtaining E) and **associating** (obtaining ϕ^E) in the process of constructing a template. Universal algebra shows the equivalence of representations. For example, ϕ^E can be derived from ϕ and E or, the other way round, ϕ can be derived from ϕ^E and the default values. Analysing the powerset template shows the following property $rng(\phi) \subseteq dom(\psi) \subseteq TS$ for $\phi : S \to TS$ and $\psi : TS \to S$. Partiality has to be considered (ψ for the powerset, for example, is only total on singleton sets, but is certainly onto).

3.3 Language Extensions

In section 3.1, we have introduced semantics extensions on semantic algebras. Now, we will introduce operators on whole language descriptions, called **language extensions**, including syntax, semantic algebras and semantic functions.

Figure 3 contains an example which introduces variables and an environment in which these variables can be stored into the `csh`-language. String expressions consist now of either strings or variables. To evaluate variables, string expressions and streams are indexed with environments. Here, we have used the templates UNION, INDEX, and MAP INJECT to express these extensions. Due to the lack of space, their full definition will not be presented. The application of templates is explained in detail in section 5. UNION is based on the disjoint union of two domains; INDEX indexes a given domain by an index domain (elements of the new function space are considered equivalent, if they map to the same element). MAP INJECT is an adaptation of INJECT for domain S to functions mapping from S to S. UNION is used to allow variables as string expressions. String expressions, now including variables, are indexed by environments in order to substitute variables by their values during execution. MAP INJECT defines an extension which makes environments modifiable. The application of these templates within the language extension guarantees that the resulting language description preserves the behaviour of the basic language description. Some specific templates for the extension of operation combinators, such as $pipe$ (see section *composition* in figure 3), are used. They are explained in section 6.

The application of templates prepare the definition of the new feature (see **New Feature** in the example). The semantics of se includes the interpretation of variables, $setenv$ modifies (overwrites) an environment.

A language extension is divided into two parts. The first part 'Extension' deals with a potential base language and its adaptation to the requirements. The second part 'New Feature' contains the definition of the new feature. The first part is divided into three subsections:

- *Syntax*: A renaming operator for syntactical identifiers might be applied[2].
- *Semantic algebra*: Semantic algebras of a basic language can be lifted to the extension level, normally by applying templates. Templates can be used in a *constrained* form applicable to particular semantics entities and in an *unconstrained* form.
- *Composition*: Application of specific extension templates for higher-order operators.

The second part is also structured into three, but slightly different parts:

- *Syntax*: Syntactical constructions for the new feature can be specified. When the extension is applied to a basic language, these elements will be added.
- *Semantics*: Semantic algebras can be provided. A simple semantic domain is considered as a semantic algebra without functions. Note, that domains can always be named, i.e. specified by an equation.

[2] In principle, a signature morphism is appropriate. We will ignore syntactical issues here.

LANGUAGE EXTENSION

ABSTRACTION =
Extension
syntax
semantic algebras
(for $\mathbf{S} : StrExpr \to String$ do
UNION $StrExpr = Chr^*$ INTO $StrExpr = (Variable + Chr)^*$;
for $\mathbf{S} : StrExpr \to String$ do
INDEX $String$ BY $Env \to String$) ,
for $\mathbf{C} : Cmd \to Stream \to Stream \times Exit$ do
MAP INJECT Env INTO $Stream \to Stream \times Exit$
composition
use STANDARD $\times$ SEQUENCE for argument $\mathbf{C}[\![cond]\!]$,
use STANDARD $\times$ DISTRIBUTE for argument $\mathbf{C}[\![pipe]\!]$,
use STANDARD $\times$ LAST RESULT for result $\mathbf{C}[\![pipe]\!]$, $\mathbf{C}[\![cond]\!]$
New Feature
syntax
StrExpr ::= (Chr | Variable)*
Cmd ::= 'setenv' Variable StrExpr
semantic algebras
domain Variable;
domain Env = Variable $\to$ String
semantic functions
$\mathbf{V} : Variable \to Env \to String$
$\mathbf{V}[\![v]\!]e \triangleq e(v)$;
$\mathbf{S}[\![se]\!]e \triangleq$ if $se =<>$ then $<>$ else let $s =$ case $hd\ se$ in
$var(v) \Rightarrow$ if $v \in dom(e)$ then $\mathbf{V}[\![v]\!]e$ else $<>$
$chr(c) \Rightarrow\ < c >$
in $s \hat{}\ \mathbf{S}[\![tl\ se]\!](e)$;
$\mathbf{C}[\![setenv(v, s)]\!](e, i) \triangleq (e \dagger [v \mapsto \mathbf{S}[\![s]\!]e], <>, true)$

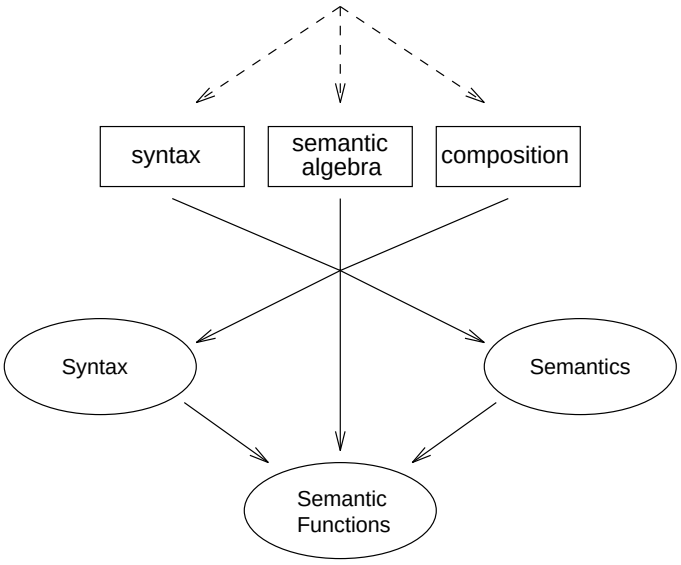
END

Figure 3: Extension by environments

- *Semantic functions*: New semantic functions can be defined, existing ones can be redefined.

Two orthogonal operators are provided for the notation: ',' and ';'. ',' is a separator for denoting independent specifications, its semantics is union. ';' is a sequencing operator, its semantics is override. The override semantics allows the redefinition of elements. The full semantics of this language extension notation will not be presented formally. The semantics extensions underlying the extension have already been presented. The next section will investigate an equational metalanguage which is in fact the basis for the description of language operators.

The following diagram shows the main constituent parts of language extensions and their dependencies.



One of the objectives of our approach is a modular presentation of semantics. Separating core and extensions is one way of achieving this. An aim of our approach is to allow language features to be specified as entities of their own which can be reused in different contexts[3]. We have here increased the degree of modularity by introducing a *parameterisation* concept. Identifiers of the feature specification will be associated with identifiers in the basic language, thus forming formal parameters of the operator. This allows the definition of language features independently of concrete core languages. David Schmidt uses the notion of *orthogonality* of language features to indicate the aim of having standardised language parts that can be assembled to a larger language in a predictable way.

4 An Equational Metalanguage for Expressing Properties

In this section, some foundations underlying the extension notation, which was presented in the previous section, shall be investigated. We will discuss an equational metalanguage to express algebraic theories, i.e. to specify and to reason about language properties such as equaivalence of programs. Programs can be specified in the metalanguage, they are interpreted in the corresponding semantic algebras. We will discuss theories represented as equivalences on terms. We will investigate the effect of extensions on these theories. Each algebra A with signature Σ – such as our semantic algebras — gives rise to a term algebra ΣA. An equivalence $\sim$ on the set of terms can be obtained by considering all terms as equivalent which are equal under some interpretation. An algebra has properties, possibly stated in an **algebraic theory**, here an equational theory. Equations hold between equivalent terms, thus, a theory can be represented by an equivalence on terms.

A description of an operation $\mathbf{C}[\![echo(se)]\!] : Stream \to Stream$

$$\mathbf{C}[\![echo(se)]\!]i \triangleq \mathbf{S}[\![se]\!]$$

can be seen as an equational specification of $\mathbf{C}[\![echo(se)]\!]$, i.e. the notation that has been implicitly introduced for language descriptions and language extensions is based on an equational language. A metalanguage is introduced to express theories and to reason about language specifications. Algebraic theories are presented as quotients of term algebras where the equivalence $\sim$ expresses equal interpretation and $\sim^E$ behaviourally equivalent interpretation of terms.

Specifications of the basic language (figure 1) can be seen as equational specifications in a metalanguage. We will now discuss the effect of extending algebras on the specifications. How do the respective term algebras and their

[3] We have chosen very concrete names directly referring to the core or more basic languages in the examples, since the aim here in this paper is the structured development and representation of one language. Reusing the feature definition, i.e. abstracting the feature definition from concrete base languages is possible, but not investigated here.

quotients relate if the basic algebra A is extended to B? Assuming that we have a semantics extension from A to B, let us investigate the relation between ΣA and ΣB and also between their quotients $\Sigma A/\sim_A$ and $\Sigma B/\sim_B$. Let us assume a syntactic extension $\sigma : \Sigma A \mapsto \Sigma B$ derived from the semantics extension using the type extension T on domains and from the function lifting $f : A \to B$ to $f^* : TA \to TB$. It can be easily seen that σ is a signature morphism. The criterion for a correct extension is, for any terms t_1 and t_2:

$$\sigma(t_1) \ \sim_B^E \ \sigma(t_2) \quad \text{iff} \quad t_1 \sim_A t_2$$

The interpretations of equivalent terms t_1 and t_2 have to be behaviourally equivalent with respect to E in the extension. Instead of $\sim_B$, we have required a weakened equivalence $\sim_B^E$ based on the equivalence E on a certain domain (as described in the previous sections). Based on Σ-terms (as usual variables and operation applications), equations can be introduced using the equivalence.

Let A be an algebra with signature Σ_A. v_A is an interpretation of Σ_A-terms t in A, i.e. $v_A(t) \in A$ is a value. $t_1 \sim_A t_2$ iff $v_A(t_1) = v_A(t_2)$ for all variables in the two terms. Analogously for an algebra B. Let B now be constructed from A via type constructor T with $B = T(A)$. Let ΣA be the term algebra, and $\Sigma A/\sim_A$ the quotient with respect to equal interpretation of terms. The signatures of A and B can be related by a signature morphism from A to TA such that $\sigma(\Sigma_A)$ is a subsignature of Σ_B as described above. As a result of the application of the signature morphism, the set of Σ-terms changes. The signature morphism resembles a renaming, if T is applied to all domains A and TA is considered as a new symbol for A. The problem with this interpretation is, that, in general, new terms are introduced and old terms are preserved (consider e.g. the introduction of a product), i.e. the signature morphism embeds Σ_A in Σ_B. Thus, the corresponding term algebras are in general not isomorphic. We can define ΣB as a homomorphic image of ΣA, if the extension is based on a semantics extension (these were introduced in section 2 and are formally defined in section 3.1).

The relation between $\sim_A$ and $\sim_B$ depends on how the interpretation v on A is adapted to an extended version v^* on B. v^* has to preserve the equivalence of terms, i.e. $t_1 \sim_A t_2 \Rightarrow t_1' \sim_B t_2'$ or $v(t_1) =_A v(t_2) \Rightarrow v^*(t_1') =_B v^*(t_2')$[4], if t_i' is an extended term for t_i. This property is called **interpretation preservation**. The interpretation can be adapted by a canonical mapping from v to v^*:

$$v^*(\sigma(f(x))) = (v(f))^*(\phi(v(x)))$$

where $\phi : A \to B$ is the extension morphism.

Proposition 4.1 *The interpretation adapted by the canonical construction $v \mapsto v^*$ is interpretation preserving.*

Proof: $\sigma(f(x))$ is the syntactically extended term for $f(x)$. The new interpretation of the extended term is constructed from the extended semantics: the original semantics $v(f)$ is semantically lifted to $(v(f))^*$, the value $v(x)$ of the argument x is mapped into the extended domain $\phi(v(x))$. Since semantical equality is the criterion for the equivalence, and the extension of v is defined via the semantics, the equivalence is preserved. □

The interpretation determines the equivalence of terms. If we adapt the interpretation, then the equivalence is also adapted. It is obvious that the canonical extension of v is interpretation preserving. The equivalence $\sim$ is preserved for a special case, but remember, that only equivalence E, i.e. $\phi_E(f(x)) \ =_E \ f_E^*(\phi^E(x))$ is required for arbitrary behaviour preservation based on E. For the general case, we define

$$t_1' \sim_B^E t_2' \ \stackrel{\triangle}{=} \ v^*(t_1') \ E \ v^*(t_2')$$

The notion of interpretation preservation has to be adapted appropriately. Still, v is preserved by v^*, or reformulated $\sim_A$ is preserved by $\sim_B^E$.

Proposition 4.2 *Assuming a behaviour preserving extension, the equivalence on terms $\sim_A$ is preserved by an extension $\sim_B^E$.*

[4]Later on we will consider an equivalence instead of an equality.

Proof: The criterion of interpretation preservation was relaxed such that equivalences can be dealt with. □

Based on $t_1 \sim_A t_2$ iff $v(t_1) = v(t_2)$, each term t_i can be uniquely extended to t_i' such that $t_1 \sim_A t_2 \Rightarrow t_1' \sim_B t_2'$ (but not the other way round) with canonical definitions of $\sim_A$ and $\sim_B$. We can achieve *bijectivity*, i.e. $t_1 \sim_A t_2 \Leftrightarrow t_1' \sim_B t_2'$ if we consider equivalence classes of terms also with respect to the second equivalence E besides $\sim$, i.e. $t_1' \sim_B^E t_2'$ iff $v^*(t_1') \; E \; v^*(t_2')$. This says that the equivalence classes are mapped 1-1 from $\sim_A$ to $\sim_B^E$. The result is that we have a homomorphism on term algebras, but not a bijective one (similar to the results on algebras themselves). And we have preservation of equivalence $\sim$ on the term algebra quotients (based on a suitable redefinition of the interpretation function).

We have neglected a satisfaction relation for our equational language so far, since the preservation of a satisfaction relation is a straightforward implication from previous results. The equation $t_1 = t_2$ is satisfied in an algebra A, if $t_1 \sim t_2$ holds for a given interpretation on A. Using the equivalence E, we have to weaken our statement. $t_1 \; E \; t_2$ is satisfied, if $t_1 \sim^E t_2$. As we have seen above, interpretations (and thus the corresponding equivalences) are preserved.

5 A Sample Extension

A simple example of an extension shall be looked at. The extension by abstraction mechanisms presented in figure 3 resembles more what we might expect as a self-contained feature, but due to the lack of space we will only explain a simpler example in detail (figure 4). Exit values shall be added to the core (figure 1) indicating whether a command was executed successfully or not. The abstract situation can be described as follows. An algebra A shall be extended to B by using the technique of *injection* for the semantic domains $S \mapsto S \times T$. The injection template INJECT summarising the definitions from section 3.2 is presented in figure 2. The proof obligation of behaviour preservation is fulfilled by using a template.

Applying the template INJECT in a language extension is presented in figure 4. A new domain $Exit$ is injected by INJECT into the result domain of commands; we have numbered the syntactical occurrences of $Stream$ – without any semantical relevance. By using the template, we map streams to pairs of streams and exit values. The template specifies that with respect to behaviour preservation of operations, only the behaviour on the stream component is relevant, but not the exit component.

As explained above, identifiers in the operator description are only formal parameters which have to be substituted by actual ones when the language extension is applied to a language description. For the sake of simplicity, we will omit the explicit application of the extension operator and assume an application with a one-one correspondence between the names of formal and actual parameters.

The conditional command $cond$ is newly introduced, thus, there is no proof obligation with respect to behaviour preservation. The other proof obligations concerning existing commands are discharged by using the template which guarantees behaviour preservation. For instance, the resulting definition for $echo$ after applying the template would be:

$$\mathbf{C}[\![echo(s)]\!](i) = (\mathbf{S}[\![s]\!], true)$$

which preserves the original behaviour (observe the first component).

The feature specified here by a language extension is an exit value concept with one operator $cond$. This operator is an operation combinator whose final result depends on the exit value of its first argument. Using the INJECT template (which allows us to derive a semantics extension), the semantics of a basic language, to which the language extension might be applied, is lifted according to the pattern $Stream_2 \mapsto Stream_2 \times Exit$ such that behaviour is preserved. The identifier $Stream_2$ is a formal parameter when applied to a basic language. It might be matched for example with a domain $State$ of a basic language when applied to that language. The semantics extension derived from a template adapts any argument semantics to the extended domain construction as specified by the language designer. The base language is now available in the extended language. Any proof obligations are automatically discharged. Its extended definition is consistent and behaviour preserving. On top of this lifted base language, the language designer can specify a command $cond$ with syntax and semantics as it is done in figure 4. If such a command already exists in the base language, it is overridden, otherwise it is a new definition.

LANGUAGE EXTENSION

EXIT_VALUES =
Extension
syntax
semantic algebras
for $\mathbf{C} : Cmd \to Stream_1 \to Stream_2$ do
INJECT $Stream_2$ INTO $Stream_2 \times Exit$ with $t_0 = true$
composition
New Feature
syntax
Cmd ::= 'cond' Cmd Cmd
semantic algebras
domain Exit
semantic functions
$\mathbf{C}[\![cond(c_1, c_2)]\!](i) \triangleq$ let $(o_1, s_1) = \mathbf{C}[\![c_1]\!](i)$ in
let $(o_2, s_2) = \mathbf{C}[\![c_2]\!](<>)$ in
if $s_1 = true$ then $(o_1 \hat{} o_2, s_2)$ else (o_1, s_1)
END

Figure 4: Extension by exit values

We have provided two distinct extension constructs, the first, *language extension*, is dedicated to the full specification of the properties of the new feature, the second, *semantics extension*, is dedicated to the behaviour preserving lifting of the basic language to some extended domain construction necessary for the new feature. The language designer shall be freed from adapting the definitions of the basic language explicitly and prove the preservation of properties and should instead be allowed to focus on the specification of the new feature.

6 Typing and Higher-Order Operators

Some more conceptual issues shall be looked at in this section. The first one concerns the adaption of functions whose types have been modified. Then, we address a particular, but very important kind of functions: higher-order functions which appear in our approach in the form of operation combinators.

6.1 Typing

The key concept of denotational semantics is the compositionality of its definitions of semantic functions, i.e. semantic functions are applied within definitions of other semantic functions. An example of an operation definition containing applications of semantic functions is the semantic function $\mathbf{E}$ for binary addition expressions in a stateless setting:

$$\mathbf{E}[\![e_1 + e_2]\!] = \mathbf{E}[\![e_1]\!] + \mathbf{E}[\![e_2]\!]$$

In an extension by state we would expect the definition to be parameterised by a state variable $s : State$.

$$\mathbf{E}^*[\![e_1 + e_2]\!] = \lambda s.\mathbf{E}^*[\![e_1]\!]s + \mathbf{E}^*[\![e_2]\!]s$$

The *type* of the semantic function changes, due to the application of the signature morphism based on the type operator T and the function lifting $\_^*$. Each occurrence of a modified function in the body – the call $\mathbf{E}[\![e]\!]$ – has to be substituted by a call of the extended version $\mathbf{E}^*[\![e]\!]s$. This applies to every function, not only the semantic functions. The form of the substitution depends on the domain extension, here $Expr \to Val$ is extended to $Expr \to Store \to Val$ by indexing Val with $Store$. Let γ in the following be the function which syntactically substitutes

in expressions defining functions. All applications of functions which have changed their types are syntactically modified. A canonical construction of γ based on the domain extension which is applied is possible. The canonical function extension $f \overset{c}{\mapsto} f^*$ has to be adapted by this substitution, e.g. $f^*(\phi(s)) = \gamma \circ f(s)$. γ is an extension of the signature morphism σ which works on metalanguage specifications.

A question that can also be asked here is whether an argument (such as s on the outer level in the example) has to be given to all of the subordinated calls unmodified. The example could have been extended in another way, e.g. adding side-effects in expressions would require different states to be used by $\mathbf{E}[\![e_1]\!]$ and $\mathbf{E}[\![e_2]\!]$. This imposes a new evaluation order on a binary expression. This issue is investigated below.

6.2 Operation Combinators

Templates describe transformations on domains and on operators on these domains. Higher order operators define the composition of operators. Operators of the language are combined to non-primitive ones. If basic operators are extended e.g. in their argument or result type, the composition of these operators has also to be adapted. It will turn out that there are certain variants in which operation combinators are defined. These variants will lead to some **templates for operator combination and extension**. Let us assume an operation combinator $\circ$ on two basic operators c_1 and c_2 and a semantic function **C**. An abstract form of a definition for **C** is:

$$\mathbf{C}[\![\circ(c_1, c_2)]\!]x = g(\mathbf{C}[\![c_1]\!]f_1(x), \mathbf{C}[\![c_2]\!]f_2(x))$$

Arguments and results to the functions shall be looked at. *Arguments* x to the argument functions c_1 and c_2 of a composition $\circ$, which are given firstly to $\circ$, have to be assigned to the argument functions. There are two common possibilities:

- DISTRIBUTE: $f_1(x) = f_2(x) = x$, i.e. $f_1 = f_2 = id$.
- SEQUENCE: $f_1(x) = x$ and $f_2(x) = g'(\mathbf{C}[\![c_1]\!]f_1(x))$ for some function g'.

The *result* of applying a function composition has to be a value of the result type of each of the functions.

- COMPOSE: the result r is a composition $r = g(r_1, r_2)$ of the results r_1 and r_2 of both argument functions (where g is an arbitrary expression on the arguments). The composition operator g has to be explicitly specified.
- LAST RESULT: often, only the result of the second argument function is taken (if the composition should implement a form of sequencing), i.e. $r = r_2$.

These variants should be preserved if an operation combinator is extended. This shall be referred to by the template name STANDARD. STANDARD is a higher-order template like operation combinators are higher-order functions. For these templates for operation combinators, behaviour preservation is guaranteed, if the basic operations are extended with preservation of behaviour.

We have already seen the INJECT template which would allow us to add a new argument or a new result domain to operators. If new arguments and/or results are added to basic operators, we do not need to stick to the given variants, since the new component is irrelevant for behaviour preservation.

Peter Mosses [1] introduces several operation combinators, called *action combinators* in Action Semantics. These combinators serve to reduce overloading of generally applied operators such as $\circ$ for function composition or the sequence ; for composition of program constructs as commands or declarations.

7 Case Studies

7.1 Revisiting David Schmidt's Book(s) on Denotational Semantics

We have mainly focused our interest on [2], but aspects of [3] are also considered. This attempt is described in [9]. One of Schmidt's examples is the extension of a simple imperative language by I/O commands working on input and

output buffers. Schmidt proposes to structure the semantical side into separate algebras, each implementing a language feature. Let us consider a basic algebra implementing a store for a simple imperative language:

$$\begin{array}{ll}\textbf{domains} & Store = Id \to Nat \\ \textbf{opns} & \mathbf{C}[\![x := e]\!] : Store \to Store \\ & \quad \mathbf{C}[\![x := e]\!]s \triangleq update(s, x, \mathbf{E}[\![e]\!]s)\end{array}$$

Stores shall be injected into a product of stores, input buffers and output buffers such that I/O-specific commands like *put* can be realised.

$$\begin{array}{ll}\textbf{domains} & State = Store \times Input \times Output \\ \textbf{opns} & \mathbf{C}[\![x := e]\!]^* : State \to State \\ & \quad \mathbf{C}[\![x := e]\!]^*(s, i, o) \triangleq (update(s, x, \mathbf{E}[\![e]\!]^*s), i, o) \\ & \mathbf{C}[\![put(x)]\!]^* : \ \ldots\end{array}$$

This extension can be expressed by using the injection template:

INJECT $Store$ INTO $State = Store \times Input \times Output$

such that behaviour is preserved. We have proven for some parts of Schmidt's examples that his informally described extensions are behaviour preserving.

7.2 The `csh` Case Study

In our second case study, we have investigated the csh-language in much more detail than described so far, see [10, 11] for details. This investigation was carried out accompanying a students project on specification and language semantics held at the Danish Technical University in Lyngby during the author's stay in Denmark.

In several steps, concepts such as file systems, exit values, aliases, I/O-redirections, or variable and command substitution were added. We have also investigated parallel extensions instead of a sequence of extension steps. In sequential extension, feature are added step by step to a basic language. In parallel extension, all features are added onto the basic language resulting in a number of language extensions. Under certain circumstances (no interactions between the features), these extensions can be merged to one final extension.

7.3 The RAISE Concurrency Model

Another application of our approach can be found in [12]. The concurrency model of the specification language RAISE [13, 14] is based on Hennessy's acceptance trees, adapted to the particular needs of RAISE, see [15] for details of this adaptation. We have used our approach of extension semantics to reformulate this adaption in a rigorous way, thereby proving that essential properties — the behaviour and structural constraints — are preserved (the formal proof of property preservation is missing in the original description).

The basic model is a recursive space of processes P_0 defined by

$$P_0 = ((\Sigma \to P_0) \times \mathcal{P}\mathcal{P}\Sigma)_\perp$$

where Σ is a set of events. The first component of those pairs is a mapping from events to processes in P_0. The second component is an acceptance set. An acceptance set is a set of possible internal states which can be reached non-deterministically by executing a process. Each of these states is a set of actions that can be taken in that particular state. The domain is defined recursively, but a solution for this equation exists (see [15] Chapter 5). $\perp$ is the semantic correspondence to the $chaos$ process.

We have partitioned the event space into two forms of events (in and out events denote the direction of cmmunication via channels), in the first extension step using a specific template to obtain process space P_1:

PARTITION $(\Sigma \to P_0)$ INTO $(\Sigma_{in} \to P_1) \times (\Sigma_{out} \to P_1)$

PARTITION is a behaviour preserving template. In a second extension, values V and states S are introduced using injection. The final process space P_2 has the structure

$$P_2 = (\mathcal{P}(S \times V) \times (\Sigma_{in} \to V \to P_2) \times (\Sigma_{out} \to V \to P_2) \times \mathcal{P}\mathcal{P}\Sigma)_\perp$$

obtained by using templates for indexing and injection.

This case study is a classical situation in which our approach can be used. It is a rigorous, modular development of one aspect — here the crucial concurrency model of the RAISE specification language — of the semantics of a real specification language. In this case, the approach of extension semantics was used as a tool for analysis and design of the language semantics. We have gained a clearly structured description of the development of the RAISE concurrency model based on a widely accepted model for concurrency. Additionally, we proved the extension to be correct, i.e. property-preserving.

8 Related Work

The process of subsequent extensions is a *refinement* process. There is, for instance, a similarity between the notion of behaviour preservation and the retrieve function in VDM (see e.g. [7]). The question has been addressed already. A similar approach to ours – also pointing out the similarity to refinement – is presented in [16]. There, a refinement relation between denotationally specified languages is provided. This paper follows in its presentation Schmidt's book [2]. Riddle and Wallis see definitions of semantic functions as semantic equations and define a correctness preserving refinement relation based on these equations. Constructive support, e.g. in form of a refinement calculus is not provided. Other approaches with similar mathematical frameworks are [17] or [18]. Another possible area of application is e.g. [19] where 27 languages derived from another are presented (in a slightly different denotational framework using metric spaces).

Abstract interpretation is a notion which we could use to describe the way we express behaviour preservation. Behaviour preservation is formalised by mapping an extended algebra to a more abstract one which neglects details, but focuses on those properties that have to be preserved from the basic language. The approach of abstract interpretation is well-know in language semantics [20, 21], but it is mostly used for optimisation purposes.

A paper on language semantics cannot leave Category Theory unmentioned. One of the most popular approaches to modularity in language semantics is based on *monads*, see e.g. [22, 23, 24]. A number of common language features have been successfully modelled as separate units based on monads. Moggi calls these descriptions *notions of computation*. There is less experience with the extension of monads. [25] provides some basic definitions, such as monad morphism, but a suitable, well-founded notation for language extensions does not exist at the moment (see more recent work on the extension of monads [26, 27]). Classical denotational semantics provides a well-understood framework on which an extension approach like ours can be based. A lot of existing semantical descriptions are only available in a classical denotational style as we have tried to indicate with the RAISE-example. Ideas realised in our framework of language extension such as the provision of a notation or a library of templates, can also be applied to a monadic framework. This is currently under investigation based on monads and their morphisms.

9 Conclusions

We have presented a language description in form of a stepwise development by extension. Based on a language core exhibiting the basic ideas of language, language features are added onto that core step by step. The language features can be specified without referring directly to the core on which they should be added. This guarantees a high degree of modularity in language design and language presentation. Language features can also be investigated as self-contained constructs of their own.

We have presented in our framework of *extension semantics* a two-level approach using two different kinds of extension operators. The first adapts the semantics of the basic language to an extended domain construction specified by the language designer. The use of this operator was simplified and supported by a notion of extension templates. This support was given in order to allow the language designer to focus on the specification of the new features to be added. The technical support by semantics extensions is crucial for the creative part of specifying the new feature. This

technical support adapts basic language constructs automatically preserving their behaviour. This support is essential for the feasibility of the extension approach. The idea of abstraction and preservation of properties can also be found in *abstract interpretations* [20, 21, 28], but this approach is mostly used to abstract in order to solve problems in a simpler (more abstract) domain. Properties of semantics can be described in form of an equational theory. We have presented mechanisms to extend equational theories according to new structural requirements such that behaviour is preserved.

Since in each step only a few concepts are explicitly added or redefined, normally a large amount of rewriting would be necessary. We have facilitated extensions by providing templates which can be applied for a number of standard cases. Applying these templates also allows properties to be preserved. Respective proof obligations are automatically discharged.

We would like to refer to the work of David Schmidt. Some of the ideas presented here have been developed based on his text books on denotational semantics [2] and [3]. Schmidt uses the notion of *orthogonal language features* to point out that features should be designed as self-contained units understandable without reference to other language features (and the core). Language features should preferably not conflict. This idea was realised in our approach by the construct of language extensions, i.e. operators with parameters. In particular the semantics extensions guarantee that the behaviour of the basic language is preserved, which means that the new feature does not conflict with the basic language. More modular, or orthogonal, descriptions of languages are also aimed at by *Action Semantics* [1]. *Facets* are provided which contain constructs to describe the computation of different kinds of information, e.g. the feature description ABSTRACTION (figure 3) could be considered as a reduced facet for describing declarations.

The area of application of our approach is geared to those language manipulations that are expressible through language extensions. The combinations of different paradigms, such as the combinations of the state-based imperative language and a process-domain based concurrent language, is not intended. Merging two different languages based on two different paradigms would require different questions to be answered. Three case studies have been presented to indicate the variety of our approach even though not all problems can be solved. Certainly, programming languages have to be addressed as well. Java, and in particular security aspects of Java, are currently under investigation. Java is in particular interesting since it is a young, still evolving language. In the same sense, Perl can be a target language.

The approach can be further improved if we consider parallel extensions. Instead of extending step by step sequentially (investigated in depth in [10]), we could extend a common core in parallel by adding different new features as long as there are no dependencies between extensions. Issues like the commutativity of extensions arise; questions such as under which conditions can extensions be merged have to be answered. An extension notation based on an operator calculus to combine extenions is certainly an improvement to the expressivity of the approach (investigated in [11]).

References

[1] P.D. Mosses. *Action Semantics*. Cambridge University Press, 1992.

[2] D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Wm.C. Brown Publishers, 1986.

[3] D.A. Schmidt. *The Structure of Typed Programming Languages*. MIT Press, 1994.

[4] O. Danvy and J. Hatcliffe. On the Transformation Between Direct and Continuation Semantics. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings Mathematical Foundations of Programming Semantics MFPS'93*. Springer-Verlag, LNCS 802, 1993.

[5] DeStijl. Design and Specification through Interfacing and Joining Languages, EU-HCM project, http://www.cs.tcd.ie/DeStijl.

[6] Unix Manual Pages. csh - shell command interpreter with a C-like syntax, 1995.

[7] C.B. Jones. *Systematic Software Development with VDM*. Prentice Hall, 1990.

[8] G. Grätzer. *Universal Algebra*. D. van Nostrand Company, 1968.

[9] C. Pahl. Structural and Methodological Investigations into the Extension of Imperative Programming Languages. Technical Report IT-TR:1997-013, Department of Information Technology, Technical University of Denmark, 1997.

[10] C. Pahl. Modular, Behaviour Preserving Extensions of the Unix C-shell Interpreter Language. Technical Report IT-TR:1997-014, Department of Information Technology, Technical University of Denmark, 1997.

[11] C. Pahl. An Investigation into Parallel Extensions of the Unix C-shell Interpreter Language. Technical Report IT-TR:1997-015, Department of Information Technology, Technical University of Denmark, 1997.

[12] C. Pahl. A Modular Development of the Denotational RSL Concurrency Model. Technical Report IT-TR:1997-016, Department of Information Technology, Technical University of Denmark, 1997.

[13] C. George. The RAISE Specification Language – A Tutorial. In S. Prehn and W.J. Toetenel, editors, *VDM'91 – Formal Software Development Methods*. Springer-Verlag, October 1991. LNCS 552.

[14] The RAISE Language Group. *The RAISE Specification Language*. CRI A/S, Denmark, 1992.

[15] D. Bolignano and M. Debabi. On the Foundations of the RAISE Specification Language Semantics. Technical report, ESPRIT project LACOS, 1992.

[16] S. Riddle and P. Wallis. Denotational Semantics and Refinement. In S. Flynn and G. O'Regan, editors, *Proc. 1st Irish Workshop on Formal Methods, July 1997, Dublin, Ireland*. Springer-Verlag, 1997.

[17] T. Clement. Data Reification without Explicit Abstraction Functions. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*. Springer-Verlag, LNCS 1051, 1996.

[18] A. Mycroft. On the Integration of Programming Paradigms. *ACM Computing Surveys*, 28(2):309–311, June 1996.

[19] J.W. de Bakker and E. de Vink. *Control Flow Semantics*. MIT Press, 1996.

[20] S. Abramsky and C. Henkin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[21] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[22] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93:55–92, 1991.

[23] P. Wadler. The essence of functional programming. In *Proc. 19th ACM Symp. on Principles of Programming Languages, Austin, Texas*, 1992. invited talk.

[24] S. Liang and P. Hudak. Modular Denotational Semantics for Compiler Construction. In H.R. Nielson, editor, *Proceedings European Symposium on Programming ESOP'96*. Springer-Verlag, LNCS 1058, 1996.

[25] E. Moggi. An Abstract View of Programming Languages. Laboratory for Foundations of Computer Science Report ECS-LFCS-90-113, Edinburgh University, 1990.

[26] P. Cenciarelli. *Computational Applications of Calculi based on Monads*. PhD thesis, University of Edinburgh, 1996.

[27] P. Cenciarelli and E. Saaman. Using Monads in Algebraic Specification. In *2th Workshop on Algebraic Development Technology (WADT97), Tarquinia, 3-6 June 1997*, 1997.

[28] D. Dams, R. Gerth, and O. Grumberg. Abstract Interpretation of Reactive Systems. *ACM Transactions on Programming Languages and Systems*, 19(2), March 1997.