Basic Research in Computer Science

CASL for CafeOBJ Users

Peter D. Mosses

BRICS Report Series **RS-00-51**

ISSN 0909-0878 **December 2000**

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/00/51/`

CASL for CafeOBJ Users*

Peter D. Mosses

BRICS & Dept. of Computer Science, Univ. of Aarhus, Denmark

Abstract. CASL is an expressive language for the algebraic specification of software requirements, design, and architecture. It has been developed by an open collaborative effort called CoFI (Common Framework Initiative for algebraic specification and development). CASL combines the best features of many previous main-stream algebraic specification languages, and it should provide a focus for future research and development in the use of algebraic techniques, as well facilitating interoperability of existing and future tools.
This paper presents CASL for users of the CafeOBJ framework, focussing on the relationship between the two languages. It first considers those constructs of CafeOBJ that have direct counterparts in CASL, and then (briefly) those that do not. It also motivates various CASL constructs that are not provided by CafeOBJ. Finally, it gives a concise overview of CASL, and illustrates how some CafeOBJ specifications may be expressed in CASL.

1 Introduction

CASL is an expressive language for the algebraic specification of software requirements, design, and architecture. It has been developed by an open collaborative effort called CoFI (Common Framework Initiative for algebraic specification and development).

This paper presents CASL for users of the CafeOBJ framework.

▷ CASL is intended as the main language of a coherent family of languages.

Vital for the support for CoFI in the algebraic specification community is the coverage of concepts of many previous specification languages. How could this be achieved, without creating a complicated monster of a language? And how to avoid interminable conflicts with those needing a simpler language for use with prototyping and verification tools?

By providing not merely a single CASL language but a coherent language family, CoFI allows the conflicting demands to be resolved, accommodating advanced as well as simpler languages. At the same time, this family is given structure by being organized largely as restrictions and extensions of CASL.

* To appear as Chapter 6 of *CAFE: An Industrial-Strength Algebraic Formal Method*, ed. K. Futatsugi, A. T. Nakagawa, and T. Tamai, Elsevier, 2000. All citations should refer to the published version.

Restrictions of CASL [13, 14] correspond closely to languages used with existing tools for rapid prototyping, verification, term rewriting, etc. Extensions to CASL are to support various programming paradigms, e.g. object-oriented, higher-order [11], reactive.

▷ A specification framework is more than just a language.

Apart from the CASL family of languages, the common framework is also to provide tool support, development methodologies, training materials, libraries, reference manuals, formal semantics and proof systems, and conversion to and from other specification languages.

▷ CASL is competitive in expressiveness with many previous algebraic specification languages.

The choice of concepts and constructs for CASL was a matter of finding a suitable balance between advanced and simpler languages, taking into account its intended applicability: for specifying the functional requirements and design of conventional software packages as abstract data types.

The design of CASL is based on a critical selection of the concepts and constructs found in previous algebraic specification frameworks. The main novelty of CASL lies in its particular *combination* of concepts and constructs, rather than in the latter *per se*.

▷ The CASL design has been tentatively approved by IFIP WG 1.3.

The design proposal for CASL [7] was submitted to IFIP Working Group 1.3 (Foundations of System Specification) in May 1997, and tentatively approved at the IFIP WG 1.3 meeting in Tarquinia, June 1997 (subject to reconsideration of some particular points [6] and the development of a satisfactory concrete syntax, both of which have now been completed [8]). The formal semantic description of CASL has been completed [9]. Tools and methodology for CASL are being developed.

▷ CoFI is open to contributions and influence from all those working with algebraic specifications.

The design of CASL has been the main responsibility of the CoFI Language Design task group, coordinated by Bernd Krieg-Brückner. At any time during the three years it took to design CASL, there were 10–20 active participants in this task group—most of them with previous experience of designing algebraic specification frameworks. The work of the CoFI task groups on Methodology, Semantics, and Tools influenced and provided essential feedback to the language design. Numerous study notes were written on various aspects of language design, and discussed at working and plenary meetings. The study notes and various

drafts of the design summary were made available electronically, and discussed on the language design mailing list (`cofi-language@brics.dk`).

The openness of the CASL design effort should have removed the risk of bias towards constructs favoured only by some particular 'school' of algebraic specification. It is hoped that CASL incorporates just those features for which there will be a *wide consensus* regarding their appropriateness, and that the common framework will indeed be able to subsume many previous frameworks and be seen as an attractive basis for future development and research—with high potential for strong collaboration.

So much for the background of CASL.

▷ Readers of this paper are assumed to be familiar with the CafeOBJ language and system.

For an introduction to CafeOBJ, see the CafeOBJ Report [10].

▷ The aims of CASL are somewhat different from those of CafeOBJ.

From The CafeOBJ Official Home Page[1] (February 2000):

> *CafeOBJ is a new generation algebraic specification and programming language. As a direct successor of OBJ, it inherits all its features (flexible mix-fix syntax, powerful typing system with sub-types, and sophisticated module composition system featuring various kinds of imports, parameterized modules, views for instantiating the parameters, module expressions, etc.) but it also implements new paradigms such as rewriting logic and hidden algebra, as well as their combination.*

In fact CASL too enjoys flexible mix-fix syntax, a powerful typing system with sub-types, and a module composition system featuring imports, parameterized modules, views for instantiating the parameters, module expressions, etc. Many of these features of CASL were directly inspired by OBJ3, and their inclusion reflects that they are indeed seen as part of the well-established, main-stream, traditional algebraic specification practice.

▷ CASL does not provide direct support for the new paradigms of rewriting logic and hidden algebra.

The omission of these new paradigms is partly because when CASL was being designed, they were indeed rather new—and are still perhaps not generally regarded as belonging to the main-stream of algebraic specification—partly because they were felt to be somewhat too advanced for inclusion in a general-purpose specification language. The design of CASL reflects the state-of-the-art of the early 1990's [2], and leaves more recent advances to be incorporated in extensions of

[1] http://www.caraway.jaist.ac.jp/cafeobj/

CASL (which are also to provide features oriented towards particular programming paradigms, such as object orientation and concurrency).

As a consequence, CASL is directly comparable only to the equational specification fragment of CafeOBJ. The design of such a specification language is, however, still a non-trivial task, and it is quite interesting to compare the design of CASL with that of equational CafeOBJ.

▷ CASL specifications need not be executable.

From the CafeOBJ Report [10]:

> *[Equational specification and programming] is inherited from OBJ [...] and constitutes the basis of the language, the other features being built on top of it. As with OBJ, CafeOBJ is* executable*, which gives an elegant declarative way of functional programming, often referred [to] as* algebraic programming*.*

Executability is an important concern also in the rewriting logic and hidden algebra paradigms, both for testing specifications and for reasoning about them.

CASL is intended primarily for expressing requirements and design decisions during software development. The constructs of CASL have been selected on the grounds of their expressiveness and semantic simplicity—but without any regard to their executability. Of course, it is important for the authors and readers of specifications to be able to investigate the consequences of axioms; for CASL specifications, this may require the use of interactive theorem-provers, in general (CASL interfaces to the Isabelle and INKA systems are already available). Nevertheless, automated rapid prototyping of some CASL specifications should still be possible. Various executable sub-languages of CASL are in any case provided [12, 14].

When requirements are expressed independently of executability concerns, specifications may gain clarity and keep closer to informal statements of requirements. As Amir Pnueli put it in his invited talk at ETAPS'98: "requirement specifications and programs are two different views that should be checked (and debugged) against each other".

With the stated differences of aims, it is only to be expected that there are some differences between the design of CafeOBJ and that of CASL. The purpose of this paper is to explain both similarities and differences—addressing primarily readers who are already familiar with CafeOBJ. Please note that both languages seem to reflect coherent designs, and that individual design choices cannot be changed without reconsideration of many other interacting issues, in general. Thus no critique of the overall CafeOBJ design is implied when some advantages of particular CASL features are pointed out (or *vice versa*).

▷ A proper comparison of the traditional algebraic approach with the new paradigms incorporated in CafeOBJ is out of the scope of this paper.

Such a comparison would have to take into account many methodological aspects of software specification and development, including case studies. The focus of this paper is on presenting CASL and explaining how its language constructs relate to those of CafeOBJ. The many novel features of CafeOBJ that have no direct counterpart in CASL are mentioned only very briefly.

Plan

First we consider the intersection of CafeOBJ and CASL: those concepts and constructs that are common to both languages. For each such CafeOBJ construct, we see how it might be expressed concretely in CASL, and consider any significant differences in the details. Then we mention the remaining constructs of CafeOBJ: those that cannot (straightforwardly) be expressed in CASL. After that, we list those constructs of CASL that cannot (straightforwardly) be expressed in CafeOBJ, and motivate their inclusion in CASL. We finish the presentation of CASL by summarizing its constructs, and by giving a few simple examples of CASL specifications.

▷ All the main points in this paper are displayed like this.

The paragraphs following each point provide details and supplementary explanation. To get a quick overview, simply read the main points and skip the intervening text. It is hoped that the display of the main points does not unduly hinder a continuous reading of the full text. (This style of presentation was inspired by a book of Alexander [1].)

2 The Common Features: CafeOBJ ∩ CASL

▷ CafeOBJ and CASL have a large number of features in common.

In this section we consider the intersection of CafeOBJ and CASL: those concepts and constructs that are common to both languages. For each such CafeOBJ construct, we see how it may be expressed in CASL. Translation between specifications involving just these constructs could probably be automated without much difficulty (although to prove that the translation is somehow semantics-preserving might not be so easy).

The high degree of overlap between the equational fragment of CafeOBJ and CASL reflects the extent to which their designs have been influenced by previous main-stream algebraic specification frameworks, of which OBJ3 is a good example. Note however that CafeOBJ was designed deliberately as a successor to OBJ3, whereas the CASL design was intended to be unbiased towards any particular previous framework (and it is as close to LSL, the Larch Shared Language, as it is to OBJ3).

Both CafeOBJ and CASL distinguish between basic and structured specifications. Let us focus first on their main constructs, deferring the various abbreviatory 'convenience features' to the end of the section.

It may be helpful to see some comparable examples straight away. The following simple examples of equational specifications in CafeOBJ are taken from the CafeOBJ Report [10, pages 18–19] (where they are used to illustrate the CafeOBJ treatment of partial functions by sort membership predicates):

```
mod* GRAPH {
  [ Node Edge ]

  ops (s_) (t_) : Edge -> Node
}

mod! PATH (G :: GRAPH) {
  [ Edge < Path ]

  op _;_ : ?Path ?Path -> ?Path (assoc)
  ops (s_) (t_) : Path -> Node

  var E : Edge
  var EP : ?Path

  ceq (E ; EP) : Path = true
        if (EP : Path) and (s EP) == (t E) .
  ceq s(E ; EP) = s(E)  if (E ; EP) : Path .
  ceq t(E ; EP) = t(EP) if (E ; EP) : Path .
}
```

Here is how the same examples could be expressed in CASL (illustrating its straightforward treatment of partiality without involving error sorts):

spec GRAPH =
  sorts $Node, Edge$
  ops $s\_\_, t\_ : Edge \to Node$

spec PATH[GRAPH] = **free {**
  sorts $Edge < Path$
  op $\_ :: \_ : Path \times Path \to ? \; Path, assoc;$
  $s\_\_, t\_ : Path \to Node$
  vars $E : Edge; \; EP : Path$
  • $def \; E :: EP \; if \; s \; EP = t \; E$
  • $s(E :: EP) = s(E) \; if \; def \; E :: EP$
  • $t(E :: EP) = t(EP) \; if \; def \; E :: EP$ **}**

In fact the semantics is not quite the same: CASL does not allow models with empty carriers, so neither the empty graph, nor discrete (edge-less) graphs are models of the specification GRAPH. This discrepancy could be remedied in various ways, not relevant to the illustrative purpose of these examples.

2.1 Basic Specifications

▷ Both languages allow declarations and axioms to be interleaved.

Basic specifications consist of declarations and axioms, written in any order such that symbols are declared before they are used.

▷ A basic specification determines a signature and a class of models.

Both CafeOBJ and CASL define the semantics of a specification in terms of model classes (rather than theories). Moreover, both provide so-called institutions for basic specifications, connecting the notions of declarations, axioms, models, and satisfaction. Both also provide the means to restrict the models of a specification to initial models.

Declarations

▷ Declarations of sorts and subsorts are similar in CafeOBJ and CASL.

The general form of a declaration of one or more sorts, together with subsort inclusions, is in CafeOBJ:

$$\texttt{[}\ S_{11}\ \ldots\ S_{1n_1}\ \texttt{<}\ \ldots\ \texttt{<}\quad S_{m1}\ \ldots\ S_{mn_m}\ \texttt{]}$$

and in CASL:

$$\textbf{sorts}\ S_{11},\ldots,S_{1n_1} < S'_1;\qquad \ldots;\qquad S_{m1},\ldots,S_{mn_m} < S'_m$$

Thus a little extra conciseness is obtainable in CafeOBJ (when the same sort is declared to be a subsort of more than one supersort).

▷ The interpretation of subsort declarations is slightly different.

CafeOBJ interprets subsort declarations as *set inclusions* between the corresponding carrier sets. The CASL interpretation is a bit more general, allowing (1–1) embeddings as well as inclusions. Such embeddings are especially relevant in the context of software design and implementation, where the models considered are not necessarily term models, since they permit more efficient representations to be used for the restricted sets of values in subsorts. For instance, in models of a CASL specification of numbers with subsorts, the representation of values in the sort of integers may be different from their embeddings into a

sort of reals or rationals. (Overloaded operations are required to commute with subsort embeddings, so the satisfaction of formulae such as $2 + 2 = 4$ is independent of the subsort embeddings involved.) Moreover, the subsort relation in CASL is in general only a pre-order, rather than a partial order, and it is possible to declare that different sorts are to be isomorphic.

▷ Declarations of total operations are similar.

A declaration of an ordinary, total operation (or constant) is in CafeOBJ:

`op` o `:` s_1 `...` s_n `->` s

In CASL it is:

op $o : s_1 \times \ldots \times s_n \rightarrow s$

or just **op** $o : s$ when the operation is just a constant.

By the way, the non-ASCII mathematical symbols shown in CASL specifications are a feature of the CASL display format, and CASL specifications have to be input in ASCII (or ISO Latin-1). For instance, '$\times$' may be input as '`*`', or directly in ISO-Latin-1, and '$\rightarrow$' is always input as '`->`'.

▷ Declarations of partial operations look quite similar, but have different semantics.

CafeOBJ does not directly support partial operations, but rather handles them indirectly via error sorts and a sort membership predicate. A declaration of a partial operation is thus written:

`op` o `:` s_1 `...` s_n `->` `?`s

where '`?`s' is an error sort (implicitly declared as a supersort of s when s is maximal in a connected component of sorts). An application of the operation is well-formed provided that the sort of each argument term is in the same connected component as the declared sort of that argument (as in Membership Equational Logic). The definedness of the value of a term corresponds to whether it belongs to a non-error sort, and can be tested (or asserted) using the sort membership predicate, written 'T `:` s'. Equations may (but need not) hold when the values of the two terms are both error values.

CASL supports partial operations directly, without implicit error sorts (cf. the example given at the beginning of Sect. 2). A declaration of a partial operation is written:

op $o : s_1 \times \cdots \times s_n \rightarrow ?\ s$

Note that in CASL, the '?' is part of the notation for declaring the profile (argument and result sorts) of operations, and '?s' is not even a valid sort name. Moreover, the result sort is not required to be maximal among the declared sorts. The definedness of the value of a term is expressed by a special atomic formula, written '*def* T'. Ordinary equations hold when both terms have equal defined values, and (always) when both values are undefined.

One noticeable difference between the two treatments of partiality is in the well-formedness of terms: in CafeOBJ, the check for well-formedness is as if each sort were to be replaced by the error supersort of its connected component of the sort hierarchy; in CASL, the check for well-formedness insists that the sorts of argument terms are subsorts of the corresponding argument sorts declared for the operation (the difference disappears when operations in CASL are declared as taking arguments in maximal supersorts).

Another difference is that in CASL, the undefinedness of any argument in an application of an operation implies that the result of the application is undefined too; moreover, a predicate never holds when any argument is undefined. In CafeOBJ, applying an operation to error values may result in a non-error value. To get the same effect in CASL one would have to declare supersorts and error values explicitly.

▷ Also declarations of predicates in CASL and CafeOBJ look similar, but here too there are some significant differences.

A declaration of a predicate is in CafeOBJ:

`pred` p `:` s_1 `...` s_n

and in CASL:

pred $p : s_1 \times \cdots \times s_n$

Predicates in CafeOBJ are, however, merely syntactic sugar for operations to the built-in sort `Bool`. To express that a predicate holds in an axiom (or not) the value of the application has to be compared to the constant `true` (or `false`). Predicates may also be used in `Bool`-sorted arguments to ordinary operations. Note that there is a distinction between the ordinary equality '`=`' and the corresponding predicate '`==`', in that the former is an atomic formula and cannot be used at all inside terms.

In CASL, predicates are kept separate from operations. An application of a predicate to argument terms is itself an atomic formula, and does not need comparing with true or false—in fact in CASL there is no built-in sort of Boolean values at all (a standard Boolean specification is however available for use if needed, for instance in sub-languages that do not allow user-declared predicates).

Regarding semantics, the CASL treatment entails that in initial models, predicates fail to hold by default (just like equations), so it is sufficient to specify only the cases where they are supposed to hold. In CafeOBJ, however, to get the expected operational semantics and initial model class, it is necessary to

specify also the cases where the result is intended to be false. In fact in the `PATH` example given in CafeOBJ at the beginning of Sect. 2, the implicit constraint that the sort `Bool` (declared by a built-in specification that is inherited by every CafeOBJ specification, and used in the representation of sort membership predicates) has only two elements prevents the existence of initial models, making (most instantiations of) the given specification inconsistent.[2]

The fundamental reason for this semantic difference is that the values `true` and `false` are treated uniformly by model homomorphisms in CafeOBJ models, whereas in CASL, homomorphisms preserve only the holding of predicates, not their non-holding. (The CASL semantics of predicates is equivalent to representing them as *partial* operations to a sort constrained to have just a single element, the definedness of the operation corresponding to the holding of the predicate.)

▷ Both languages allow overloading.

In CafeOBJ, overloading is allowed provided that signatures are 'sensible':

> *If the same operation name is declared with argument sorts* $s_1, \ldots, s_n$ *and result sort* s *and also with argument sorts* $s'_1, \ldots, s'_n$ *and result sort* s'*, then* s *and* s' *are in the same connected component iff for each* i*,* s_i *and* s'_i *are in the same connected component.*

Although this condition appears to be an improvement on several previous conditions proposed for order-sorted signatures, it prohibits declaring the same *predicate* symbol for unconnected argument sorts (in the same module), e.g. '`_<=_`' for both numbers and strings, since predicates are simply operations of sort `Bool`. Moreover, it appears that the implicitly-declared equality predicate '`_==_`' *always* leads to non-sensible signatures (even when there is only one declared sort—recalling that error sorts are implicitly declared). A relatively minor further point is that constants cannot be overloaded between different connected components, whereas it could be convenient to use the same symbol for the nil list in different sorts of lists, for instance. Finally, in connection with structured specifications, there is the problem that the restriction to sensible signatures might not be preserved by the structuring operations.

CASL, in contrast, does not impose any restrictions on overloading at all. The combination of overloading of operations with subsort inclusions between their argument and result sorts simply entails some implicit axioms, which ensure that terms have the same value whenever they are identical up to the commuting of overloaded operations with subsort inclusions. When axioms are well-formed, their satisfaction is insensitive to overloading resolution. (One might fear that such a lack of discipline would unduly complicate overloading resolution, but it can be implemented so that in the case the signature happens to be regular, the

[2] Also [10, Example 20] implicitly imports `BOOL` and exhibits the same problem. A referee pointed out that one may override the implicit importation of `Bool` in protecting mode in such examples by importing it explicitly in extending mode; however, the consequences of thus allowing 'junk' values in `Bool` in CafeOBJ are unclear.

complexity is that same as that in OBJ3, and moreover the slow-down appears to be insignificant for the non-regular signatures that occur in practice. In any case, the authors of specifications have the possibility of indicating the intended sorts when using heavily-overloaded operations.)

Axioms

▷ Both languages allow equations.

In CafeOBJ equations are written 'T_1 `=` T_2', except in conditions, where they are written as applications of the equality predicate: 'T_1 `==` T_2'.

Also CASL has two kinds of equations, but here the difference is with respect to undefined values. An ordinary equation '$T_1 = T_2$' is interpreted as asserting that either both values are both defined and equal, or they are both undefined. An existential equation '$T_1 \stackrel{e}{=} T_2$' asserts that the values are both defined and equal. The two kinds of equations are actually inter-definable in CASL, but they are both provided as language constructs, since existential equations are more appropriate in conditions (one does not usually want coincidental undefinedness of two terms to have further consequences) and ordinary equations are particularly useful in inductive definitions of partial functions.

▷ Both languages allow sort membership assertions.

In CafeOBJ sort membership is a predicate, and the assertion that a term T has sort s is written 'T `:` s `= true`'.

In CASL, a sort membership assertion is an atomic formula, written '$T \in S$'.

▷ Both languages allow conditional formulae.

In CafeOBJ a conditional equation is written '`ceq` T_1 `=` T_2 `if` T', where T is a term of sort `Bool`, and may thus involve not only the equality and subsort membership predicates but also all the logical connectives that are defined on `Bool`.

In CASL, a conditional equation is a special case of a conditional formula, which may be written in two ways: '$F_1 \Rightarrow F_2$' (not to be confused with the CafeOBJ notation for rewriting transitions) or 'F_2 *if* F_1', where the F_i are arbitrary formulae.

2.2 Structured Specifications

Specifications may be structured both by module expressions and by the naming of (possibly parameterized) specification modules.

▷ There are some syntactic differences between CafeOBJ and CASL concerning module expressions and module bodies.

In CafeOBJ, module expressions are used mainly for specifying parameters of generic specifications and views, whereas in CASL they are also used as the bodies of named specifications. Moreover, although both languages allow references to named specifications in module expressions, CafeOBJ does not (according to [10, Appendix]) allow a reference to a named specification to be replaced by its body, nor the direct use of a basic specification in a module expression, in contrast to CASL.

Extensions

▷ Both languages allow extensions of named specifications.

In CafeOBJ, an extension is always specified by a named module, which refers to the named modules being extended as imports, along with declarations and axioms in its body:

`mod` SN `...` { m_1 SN_1 `...` m_n SN_n `...` }

The mode m of each import is specified as `protecting`, `extending`, or `using`.

In CASL, an extension is itself a module expression 'SP_1 **then** SP_2'. The combination of extension with naming is written:

spec SN ... = SN_1 **and** ... **and** SN_n **then** ...

No explicit mode for the extension is specified (although one may use annotations, not affecting the algebraic semantics of specifications, to indicate properties such as the conservativeness of an extension).

▷ Both languages allow protecting extension of imports.

The strongest importation mode of CafeOBJ, '`protecting`', allows the expansion to add new sorts, operations, and predicates, but not to add new values to previous sorts, nor to equate old values. If the semantics of the imported specification is loose, however, a 'protecting' extension that requires new values, or the identification of old values, can still have models: namely, those models of the imported specification that *already* had such 'junk' or 'confusion'. This mode of importation corresponds to ordinary extension in CASL: 'SP_1 **then** SP_2'.

▷ Both languages allow loose and initial semantics.

In CafeOBJ the semantics of a module is specified by the keyword used when naming it: '`mod*`' gives loose semantics, '`mod!`' gives initial semantics.

In CASL, the initial semantics of a specification is obtained from the default loose semantics using the specification expression '**free** SP'. When used together with naming it is written:

spec SN ... = **free** ...

In both CafeOBJ and CASL, it may happen that the loose semantics of a specification is a non-empty class of models that has no initial model, in which case its initial semantics is the empty model class.

▷ Both languages allow free extension.

Free extension in CafeOBJ generalizes initial semantics to the case when the specification has imports, still being specified by use of the keyword '`mod!`' when naming the specification. CafeOBJ takes all models of the extension that are free extensions of *any* models of the imports (according to the somewhat informal semantics given in the CafeOBJ Report [10]).

Also in CASL, free extension is specified by a combination of the constructs used for expressing initial semantics and extension, being written 'SP_1 **then free** SP_2'. It denotes the class of models of the extension that are free extensions of *their own reducts* to SP_1.

In both languages, the possibility of combining loose semantics and free extensions allows specifications whose semantics is partly loose and partly initial (as with the use of so-called data constraints in Clear).

▷ Both languages allow union of specifications.

The CafeOBJ construct 'SP_1 `+` SP_2' is called a shared sum, and it is somewhat different from the union construct 'SP_1 **and** SP_2' in CASL. In CafeOBJ, a symbol declared by both SP_1 and SP_2 is regarded as the same symbol only when it has a unique origin in some imported module, in general; thus some kind of qualification of symbols by module names may be needed to disambiguate identical symbols of different origins.

In CASL, however, all common symbols of SP_1 and SP_2 are regarded as identifying the *same* entities (cf. union of traits in LSL). In fact the 'same name, same thing' philosophy is quite pervasive in CASL. (This does not prevent overloading, since operation symbols with different profiles are considered to be different names.)

Clearly, when specifications written by different people are united, there is a danger of unintentional clashes of names. But in CASL it is straightforward to hide auxiliary symbols that are introduced purely for internal use, leaving visible only the symbols that were originally *intended* to be specified. (N.B. The notion of hiding here is completely different from that found in the 'hidden algebra' approach.) Unintentional clashes of names in the *interface* of a module *need* to be resolved in any case, by renaming. Tools should be able to warn about clashes that might be unintentional (ignoring symbols that get indirectly imported from the same origin).

▷ Both languages allow translation of symbols declared by specifications.

A translation in CafeOBJ is written:

SP `*` { `...,` `sort` s_1 `->` s_2`,` `...,` `op` o_1 `->` o_2`,` `...` }

and in CASL:

SP **with** $\ldots$, $s_1 \mapsto s_2$, $\ldots$, $o_1 \mapsto o_2$, $\ldots$

(Keywords such as '**sort**' and '**op**' may be inserted here also in CASL, when desired.) Symbols that are to be left unchanged may be omitted in both languages.

Parameters

▷ Both languages allow parameterized modules.

A module with a parameter in CafeOBJ is written:

`mod` SN `(`X_1 `::` m_1 SP_1`)` { `...` }

and in CASL:

spec SN $[SP_1] = \ldots$

In CASL, the parameter is simply a specification expression (often just a name, in practice) without any label or mode, simply indicating which part of the extension of the parameter SP_1 by the body SP is intended to vary.

▷ Both languages allow instantiation of parameters with module expressions.

In CafeOBJ an instantiation of parameter X_1 with a module expression SP_2 in a module named SN can be written 'SN`(`X_1 `<= view to` SP_2{SM_1}`)`', where SM_1 is a symbol mapping (as in a translation). In simple cases the label X_1 can be omitted and the explicit view can be replaced by SP_2 itself, giving simply 'SN`(`SP_2`)`'

In CASL a similar instantiation can be written '$SN[SP_2$ **fit** $SM_1]$', or just '$SN[SP_2]$' in simple cases.

The semantics of instantiation corresponds to a pushout construction in both languages.

▷ Both languages allow views to be named and used in instantiations.

The fitting between a particular parameter specification and argument specification may itself be named in CafeOBJ: '`view` VN `from` SP_1 `to` SP_2 { SM_1 }' and VN may then be used instead of the explicit view in instantiations.

Named views are defined similarly in CASL: '**view** VN **from** SP_1 **to** $SP_2 = SM_1$', whereafter '**view** VN' may then be used instead of 'SP_2 **fit** SM_1'.

▷ Although both languages allow multiple parameters, their treatment of shared symbols is different.

The basic treatment of multiple parameters in CafeOBJ is non-sharing, where the same symbol occurring in different parameters is distinguished by the different labels of the parameters. However, there is also a construct '`share(`*SN*`)`' that allows unwanted duplication of the module *SN* (due to importation in different instantiated parameters) to be avoided.

In contrast, multiple parameters in CASL are regarded as parts of a single united parameter, thus symbols common to different parameters always share, and have to be instantiated uniformly. Multiple parameters that are intended to be independent should therefore always have distinct symbols. Thus to specify generic pairs with different sorts of components, one should declare the parameters as PAIR[**sort** *Elem1*][**sort** *Elem2*], rather than as PAIR[**sort** *Elem*][**sort** *Elem*].

The CASL 'same name, same thing' philosophy eliminates any need for 'dot notation', which is used in CafeOBJ to determine the parameter to which a symbol is supposed to refer.

2.3 Convenience Features

▷ Both languages allow attributes in operation declarations.

Both CafeOBJ and CASL provide the attributes of associativity, commutativity, idempotency, and unit/identity for binary operations.

▷ Both languages allow simultaneous declarations.

Several operations with the same profiles (and attributes) may be declared together, abbreviating a sequence of declarations.

▷ Both languages allow both global and local variable declarations.

Explicit variable declarations are global in CafeOBJ, whereas implicit declarations in terms are presumably local to the rest of the enclosing equation.

Variable declarations in CASL are global, abbreviating universal quantification over the declared variables in all the subsequent axioms of the enclosing basic specification—unless they are followed by a '•', in which case their scope is restricted to the following •-separated list of axioms.

▷ Both languages allow mixfix notation.

The usefulness of mixfix notation, generalizing infix, prefix, and postfix notation to allow arbitrary sequences of fixed tokens and arguments, is so obvious that the CafeOBJ Report [10] hardly bothers to mention it. The main difference in CASL is that place-holders are written with double underscores (leaving single underscores available for separating words in identifiers).

3 The Differences: CafeOBJ \ CASL

The main features of CafeOBJ that are missing from CASL include built-in support for hidden algebra, behavioural equivalence, and rewriting logic, as well as indications of import and parameter modes.

3.1 Basic Specifications

▷ CASL does not allow 'hidden' sorts or behavioural operations.

Sort symbols can be hidden in CASL, but the meaning is completely different from hidden sorts in CafeOBJ: in CASL, the sorts are simply removed from the declared signature (together with all operations whose profiles involve them).

Similarly, there is no way of distinguishing behavioural operations from other ones in CASL.

▷ CASL does not allow hidden and behavioural equivalence.

The CASL methodology is to provide a notion of behavioural refinement between CASL specifications, based on behavioural equivalence [3].

▷ CASL does not provide built-in transitions.

The carriers in CASL are just sets, rather than (thin) categories, and operations are not regarded as functors.

3.2 Structured Specifications

▷ CASL does not allow translation to derived operations.

In CASL one may concisely define the desired operations in terms of the original ones, and subsequently hide the latter.

▷ CASL does not allow different modes of imports and parameters.

The only mode provided by CASL corresponds to 'protecting' in CafeOBJ. For loose specifications there is not so much difference between the 'protecting', 'extending', and 'using' modes of CafeOBJ. The effect of the latter two modes for extending specifications with initial semantics can generally be obtained delaying the initial semantics until after the extension (although this may require naming extra loose specifications). The methodological implications of adding new values, or identifying old ones, in a specification with initial semantics appear to be not so clear.

▷ CASL does not allow parameters to be implicit.

The rule in CASL is that whenever reference is made to a named specification, fitting arguments have to be provided for all parameters. This should help both readers and writers of parameterized specifications to keep track of just which parameters have to be instantiated. Note that in practice, parameter specifications are often themselves named (or simply the basic specification '**sort** *Elem*') so partial instantiation can still be expressed quite concisely.

▷ CASL does not allow parameters to be labelled.

The symbols declared by the parameters of a generic specification are used directly in the body of the specification, without any distinguishing dot-notation. This is consistent with the CASL treatment of instantiation, where a symbol declared by more than one parameter has to be instantiated uniformly by each argument.

3.3 Convenience Features

▷ CASL does not allow long forms of keywords.

The keywords used in the CASL concrete syntax are mostly quite short, many of them being abbreviations of English words. Keywords at the beginning of lists of declarations, etc., can generally be used in either singular or plural form (without regard to the number of items in the list) but otherwise, each keyword has a unique spelling.

4 The Differences: CASL \ CafeOBJ

The features of CASL that are missing from CafeOBJ include explicit treatment of partiality, general first-order axioms, some convenient abbreviations, and several constructs for structuring specifications and implementations.

Adding some of these features to CafeOBJ (e.g., first-order axioms) would probably hinder execution of specifications using term rewriting; other constructs could probably be added without any significant problems.

4.1 Basic Specifications

▷ CASL allows sort generation constraints.

A sort generation constraint in CASL is essentially a sentence, but it is specified by prefixing a group of signature declarations with the keyword '**generated**'. The effect is not only to eliminate the possibility of 'junk' in the carrier of each declared sort, but also to identify which operations are sufficient basis to generate

all the values of the sort. This ensures the soundness of structural induction proofs with cases for the generating operations. In contrast to initiality and free extension, the values of the generated sorts may later be identified without violating the constraint.

▷ CASL allows unrestricted first-order formulae.

CASL provides standard notation for first-order quantifiers and logical connectives, including negation and disjunction. In practice, the majority of specifications will probably use only positive Horn-clauses, universally quantified over all variables; such axioms are also known to be theoretically sufficient for specifying all computable functions. Sometimes, however, the extra expressiveness of the quantifiers and connectives can be used to achieve significantly greater conciseness and comprehensibility.

Some readers might here be wondering whether CASL should be called an *algebraic* specification language at all, since the majority of algebraic specification languages hitherto have been limited, like CafeOBJ, to equational or positive conditional axioms, ensuring close connections with fundamental concepts of universal algebra, and keeping contact with the notion of 'high-school algebra'. The traditional focus on initial-algebra semantics of specifications reinforced the impression that these limitations were somehow essential.

But the term 'algebraic specification' has another widely-held interpretation: simply the specification of (classes of) algebras, regardless of the logic employed and the properties of the specified classes. This is the main sense in which it is meant in 'CASL'.[3]

Universal quantification in CASL is written $\forall V : S \bullet F$. Existential quantification is written using $\exists$. The standard logical connectives are written $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \Rightarrow F_2$ (alternatively F_2 *if* F_1), $F_1 \Leftrightarrow F_2$, and $\neg F$; the atomic formulae *true* and *false* are provided too.

4.2 Structured Specifications

▷ CASL allows declared symbols to be hidden.

The hiding of some of the declared symbols in CASL is written:

SP **hide** $SY_1, \ldots, SY_n$

[3] The design and institution-independent semantics of structured specifications in CASL have a 'strongly-algebraic flavour' also in the sense of universal algebra.

When a sort is hidden in CASL, all the operations whose profiles involve that sort get hidden too (ensuring that the resulting signature is well-formed).

One may also specify just those symbols that are not to be hidden, i.e., the symbols to be revealed, using a similar construct.

▷ CASL allows specification of architectural structure.

The essential difference between the *structure of a specification* and a *specification of model structure* is reflected in CASL by providing two kinds of specifications, called respectively *structured* and *architectural*.

Structured specifications allow a large specification to be presented in small, logically-organized parts, with the pragmatic benefits of comprehensibility and reusability. In CASL, the use of these constructs has absolutely no consequences for the structure of models, i.e., of the code that implements the specification. For instance, one may specify integers as an extension of natural numbers, or specify both together in a single basic specification; the models of the complete specification are the same.

It is especially important to bear this in mind in connection with parameterized specifications. The definition of a parameterized specification of lists of arbitrary items, and its instantiation on integers, does *not* imply that the implementation has to provide a parameterized program module for generic lists: all that is required is to provide lists of integers (although the implementor is free to *choose* to use a parameterized module, of course). Sannella, Sokołowski, and Tarlecki [15] provide extensive further discussion of these issues.

In contrast, an architectural specification requires that any model should consist of a collection of separate component units that can be composed in a particular way to give a resulting unit. Each component unit is to be implemented separately, providing a decomposition of the implementation task into separate subtasks with clear interfaces.

In CASL, an architectural specification consists of a collection of component unit specifications, together with a description of how the implemented units are to be composed. A model of such a specification consists of a model for each component unit specification, and the described composition. See [4] for further details, motivation, and examples.

CafeOBJ provides structured specifications, but since it does not support architectural specifications, there is no way of expressing whether the various modules are intended to be implemented separately (which might be overly general) or together (where the knowledge of the way that they are combined might be exploited, possibly hindering later reuse of the software in other contexts).

4.3 Convenience Features

In CASL, the main purpose of providing abbreviations is conciseness and perspicuity, which are perhaps even more important qualities for the readers of a specification than for its writer(s). A secondary purpose is to provide syntactic support for particular methodologies.

▷ CASL allows *definitions* of subsorts, operations, and predicates.

A subsort definition is written:

sort $S = \{\ V : S' \ \bullet\ F\ \}$

and declares S to be the subsort of S' consisting of just those values of the variable V for which the formula F holds.

A total function definition is written:

op $f(V_1 : S_1; \ldots; V_n : S_n) : S\ =\ T$

and a constant definition as:

op $c : S\ =\ T$

CASL also provides similar constructs for defining partial functions and predicates.

▷ CASL allows declarations of datatypes with constructors and selectors.

In a practical specification language, it is important to be able to avoid tedious, repetitive patterns of specification, as these are likely to be carelessly written, and never read closely. The CASL construct of a datatype declaration collects together several such cases into a single abbreviatory construct, which in some respects corresponds to a type definition in STANDARD ML, or to a context-free grammar production in BNF.

A datatype declaration is written

type $S\ ::=\ A_1 \mid \ \ldots\ \mid A_n$

It declares the sort S, and lists the alternatives A_i for that sort. An alternative may be a constant c, whose declaration is implicit; or it may be a list of sorts, written *sorts* $S_1, \ldots, S_n$, to be embedded as subsorts; or it may be a 'construct' (essentially an indexed product) written $f(\ldots; f_i{:}S_i; \ldots)$, given by a constructor function f together with its argument sorts S_i, each optionally accompanied by a selector f_i. The declarations of the constructors and selectors, and the assertion of the expected axioms that relate them to each other, are implicit.

Datatype declarations may be prefixed by '**generated**' or '**free**'. Both of these provide the appropriate sort generation constraint, and the latter construct ensures moreover that the ranges of the constructors are disjoint. In particular, the free datatype declaration of constants:

free type $S\ ::=\ c_1 \mid \ \ldots\ \mid c_n$

corresponds to an ordinary enumerated type as found in programming languages (although no implicit successor function is provided here).

▷ CASL allows conditional terms.

A conditional term is written 'T_1 *when* F *else* T_2', where the condition is a formula F. Its use in a term abbreviates a formula of the form:

$(F \Rightarrow \ldots T_1 \ldots) \wedge (\neg\, F \Rightarrow \ldots T_2 \ldots)$

It appears that use of such a construct would provide a significant abbreviation in specifications following the style of some of those illustrated in the CafeOBJ Report [10, pp. 153–155].

▷ CASL allows omission of repeated keywords.

An item of a specification is assumed to be of the same kind as the previous item, unless an explicit keyword indicates otherwise. For instance, it is sufficient to write the keyword '**ops**' just once at the beginning of a list of separate operation declarations.

▷ CASL allows display annotations.

CASL provides a uniform way of influencing the way that (user-declared) symbols are to be displayed. CASL input syntax for symbols is restricted to ISO Latin-1 characters, but display annotations allow mathematical and other symbols to appear in the output, potentially enhancing readability.

▷ CASL allows local specifications.

A local specification allows declared symbols to be automatically hidden after use. It is written:

local SP_1 **within** SP_2

▷ CASL allows compound identifiers.

CASL allows the use of compound sort identifiers of the form '$I[\ldots, I_i, \ldots]$' in generic specifications. For instance, instead of using just *List* for the sort of lists, it may be a symbol formed with the sort of items *Elem* as a component: $List[Elem]$. The fitting of the parameter sort *Elem* to an argument sort affects this compound sort symbol for lists too, giving distinct symbols such as $List[Int]$, $List[Char]$ when instantiating lists with integers *Int* and characters *Char*, and thereby avoiding the danger of unintended identifications (and the need for explicit renaming) when combining such instantiations.

Note that CafeOBJ allows instead the disambiguation of sort names by qualifying them with module expressions that identify their origins.

▷ CASL allows libraries of specifications to be named and referenced.

In CASL, libraries of definitions may be named and installed on the Internet. A library may specify the downloading of named definitions from named libraries, possibly providing a new local name. Downloading from previous versions of libraries can be indicated. Local libraries, which have not yet been installed for global access, are referred to temporarily by their URLs.

The CafeOBJ system also supports libraries, but this feature is external to the CafeOBJ language (and thus not described in the CafeOBJ report).

5 Conclusion

Taking into account the different aims and methodologies espoused by the designers of CafeOBJ and CASL, the correspondence between the equational fragments of the two languages is perhaps as close as could reasonably be expected—especially in view of the intended close relationship between CafeOBJ and OBJ3, and the CoFI requirement for CASL to be unbiased towards any particular previous framework. Some of the differences could easily be eliminated, e.g., datatype declarations might be added to CafeOBJ, derived operations might be added to CASL. Others would require more work, e.g., adding syntactic and semantic support for rewriting logic to CASL. It appears that some problems with the representation of predicates in CafeOBJ might be removed by adopting an approach closer to that adopted for CASL; similarly for overloading, although the mixing of grouping analysis and overloading resolution in CafeOBJ may make the implementation of CASL-style overloading less tractable.

Acknowledgements Christine Choppy, Răzvan Diaconescu, Kokichi Futatsugi, and Till Mossakowski provided useful comments on an earlier version of this paper, as did several participants of the CafeOBJ Symposium at which it was presented. The five anonymous referees pointed out several inaccuracies in the submitted version, and made many useful suggestions for improvement. Special thanks to the participants of the various CoFI task groups for the collaborative effort which has resulted in the design of CASL. The author gratefully acknowledges the support of BRICS (Basic Research in Computer Science, a centre of the Danish National Research Foundation).

References

1. C. Alexander. *A Timeless Way of Building*. Oxford University Press, 1979.
2. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors. *Algebraic Foundation of Systems Specification*. IFIP State-of-the-Art Reports. Springer-Verlag, 1999.
3. M. Bidoit, D. Sannella, and A. Tarlecki. Behavioural encapsulation. Language Design Study Note MB+DTS+AT-1, in [5], 1996.
4. M. Bidoit, D. Sannella, and A. Tarlecki. Architectural specifications in CASL. In *AMAST '98, Proc. 7th Intl. Conference on Algebraic Methodology and Software Technology, Manaus*, volume 1548 of *LNCS*, pages 341–357. Springer-Verlag, 1998.
5. CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from `http://www.brics.dk/Projects/CoFI`.
6. CoFI Language Design Task Group. Response to the Referee Report on CASL. Documents/CASL/RefereeResponse, in [5], Aug. 1997.
7. CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Design Proposal. Documents/CASL/Proposal, in [5], May 1997.
8. CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary. Documents/CASL/Summary, in [5], Oct. 1998.
9. CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language – Semantics. Note S-9 (version 0.95), in [5], Mar. 1999.

10. R. Diaconescu and K. Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.
11. A. Haxthausen, B. Krieg-Brückner, and T. Mossakowski. Extending CASL with higher-order functions — design proposal. Note L-8 (see also Note L-10), in [5], Jan. 1998.
12. H. Kirchner and C. Ringeissen. Executing CASL equational specifications with the ELAN rewrite engine. Note T-9, in [5], Oct. 1999.
13. T. Mossakowski. Sublanguages of CASL. Note L-7, in [5], Dec. 1997.
14. T. Mossakowski. Two "functional programming" sublanguages of CASL. Note L-9, in [5], Mar. 1998.
15. D. Sannella, S. Sokołowski, and A. Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Inf.*, 29:689–736, 1992.

Appendix: CASL Overview and Examples

This section gives a concise overview of all the main CASL features, covering both that are in common with CafeOBJ as well as those that are not.

▷ Basic specifications in CASL list declarations, definitions, and axioms.

Functions are partial or total, and predicates are allowed. Subsorts are interpreted as embeddings. Axioms are first-order formulae built from definedness assertions and both ordinary and existential equations. Sort generation constraints can be applied to groups of declarations. Datatype declarations are provided for concise specification of enumerations, unions, and products.

▷ Structured specifications in CASL allow translation, reduction, union, and extension of specifications.

Extensions may be required to be conservative and/or free; initiality constraints are a special case. A simple form of generic specification is provided, together with instantiation involving parameter-fitting translations that affect compound identifiers.

▷ Architectural specifications in CASL express implementation structure.

The specified software is to be composed from separately-developed, reusable units with clear interfaces.

▷ Libraries in CASL provide collections of named specifications.

Downloading involves retrieval of specifications from distributed libraries.

CafeOBJ Examples in CASL

The following specifications illustrate how one might translate some of the examples given in the CafeOBJ Report [10] into CASL. See also the examples given at the beginning of Sect. 2 (Example 12 in [10]) for an illustration of the declaration of partial operations.

Example 1

```
mod! BARE-NAT {
  [ NzNat Zero < Nat ]

  op 0 : -> Zero
  op s_ : Nat -> NzNat
}
```

CASL:

spec BARENAT =
  free types
    $Nat \ ::= \ 0 \mid sort\ NzNat;$
    $NzNat \ ::= \ s\_\_(Nat)$

Example 23

```
mod! SIMPLE-NAT {
  protecting(BARE-NAT)

  op _+_ : Nat Nat -> Nat {comm}

  eq s(N:Nat) + M:Nat = s(N+M) .
  eq N:Nat + 0 = N .
}
```

CASL:

spec SIMPLENAT =
  BARENAT **then**
  op $\_ + \_ : Nat \times Nat \to Nat, comm$
  vars $M, N : Nat$
  • $s(N) + M = s(N + M)$
  • $N + 0 = N$

Example 26

```
mod* MON* (X :: TRIV) {

  op nil : -> Elt
  op _;_ : Elt Elt -> Elt {assoc idr: nil}
}

mod* CMON* (Y :: MON) {

  op _;_ : Elt Elt -> Elt {comm}
}
```

CASL:

spec MON [**sort** *Elt*] =
  ops $nil : Elt$;
  $\_ * \_ : Elt \times Elt \to Elt, assoc, unit\ nil$

spec CMON [MON [**sort** *Elt*]] =
  op $\_ * \_ : Elt \times Elt \to Elt, comm$

Example 27

```
mod* MON-POW (POWER :: MON, M :: MON)
{
  op _^_ : Elt.M Elt.POWER -> Elt.M

  vars m m' : Elt.M
  vars p p' : Elt.POWER

  eq (m ; m')^ p  = (m ^ p) ; (m' ^ p) .
  eq m ^ (p ; p') = (m ^ p) ; (m ^ p') .
  eq m ^ nil      = nil .
}
```

CASL:

spec MONPOW [MON [**sort** *Elt*]] [MON [**sort** *Pow*]] =
  op $\_ \uparrow \_ : Elt \times Pow \to Elt$
  vars $e, e' : Elt$; $p, p' : Pow$
  - $(e * e') \uparrow p = (e \uparrow p) * (e' \uparrow p)$
  - $e \uparrow (p * p') = (e \uparrow p) * (e \uparrow p')$
  - $e \uparrow nil = nil$

Recent BRICS Report Series Publications

RS-00-51 Peter D. Mosses. *CASL for CafeOBJ Users*. December 2000. 25 pp. Appears in Futatsugi, Nakagawa and Tamai, editors, *CAFE: An Industrial-Strength Algebraic Formal Method*, 2000, chapter 6, pages 121–144.

RS-00-50 Peter D. Mosses. *Modularity in Meta-Languages*. December 2000. 19 pp. Appears in *2nd Workshop on Logical Frameworks and Meta-Languages*, LFM '00 Proceedings, 2000.

RS-00-49 Ulrich Kohlenbach. *Higher Order Reverse Mathematics*. December 2000. 18 pp.

RS-00-48 Marcin Jurdziński and Jens Vöge. *A Discrete Stratety Improvement Algorithm for Solving Parity Games*. December 2000.

RS-00-47 Lasse R. Nielsen. *A Denotational Investigation of Defunctionalization*. December 2000. Presented at *16th Workshop on the Mathematical Foundations of Programming Semantics*, MFPS '00 (Hoboken, New Jersey, USA, April 13–16, 2000).

RS-00-46 Zhe Yang. *Reasoning About Code-Generation in Two-Level Languages*. December 2000.

RS-00-45 Ivan B. Damgård and Mads J. Jurik. *A Generalisation, a Simplification and some Applications of Paillier's Probabilistic Public-Key System*. December 2000. 18 pp. Appears in Kim, editor, *Fourth International Workshop on Practice and Theory in Public Key Cryptography*, PKC '01 Proceedings, LNCS 1992, 2001, pages 119–136. This revised and extended report supersedes the earlier BRICS report RS-00-5.

RS-00-44 Bernd Grobauer and Zhe Yang. *The Second Futamura Projection for Type-Directed Partial Evaluation*. December 2000. To appear in *Higher-Order and Symbolic Computation*. This revised and extended report supersedes the earlier BRICS report RS-99-40 which in turn was an extended version of Lawall, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '00 Proceedings, 2000, pages 22–32.