Tool support for CSP-CASL.

Gimblett, Andy

Tool Support for CSP-CASL

Andy Gimblett

Submitted to the University of Wales in fulfillment
of the requirements for the degree of
Master of Philosophy

September 2008

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

Summary

This work presents the design of the specification language CSP-CASL, and the design and implementation of parsing and static analysis tools for that language. CSP-CASL is an extension of the algebraic specification language CASL, adding support for the specification of reactive systems in the style of the process algebra CSP. While CSP-CASL has been described and used in previous works, we present the first formal description of the language's syntax and static semantics. Indeed, this is the first formalisation of the static semantics of any CSP-like language of which we are aware.

We describe CSP-CASL both informally and formally. We introduce and systematically describe its various components, with examples, and consider various design decisions made along the way. On the formal side, we present grammars for its abstract and concrete syntax, specify its static semantics in the style of natural semantics, and formulate a solution to the problem of computation of *local top elements* of CSP-CASL specifications.

Going on, we describe tool support for the language, as implemented using the functional programming language *Haskell*; in particular, we have a parser utilising the monadic combinator library *Parsec*, and a static analyser directly implementing our static semantics in Haskell. The implementation extends HETS, an existing toolset for specifications written in heterogeneous combinations of languages based on CASL.

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed ..... (candidate)

Date ..... 17/09/08 .....

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed ..... (candidate)

Date ..... 17/09/08 .....

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... (candidate)

Date ..... 17/09/08 .....

To Basheera

Acknowledgements

It is my pleasure to start this thesis by extending my sincere and heartfelt thanks to my long-suffering supervisor, Markus Roggenbach, for his guidance, expertise, encouragement, friendship and patience over the course of this work. I am sure the process was more than a little frustrating for a good deal of the time, as he watched me battle to balance my teaching work with this research, and frequently lose. It has, I am sure, been an education for us both. Within this context, my appreciation for Erwin R. Catesbeiana Jr's ongoing pastoral support for my supervisor is unbounded.

I thank Swansea University's Department of Computer Science as a whole for providing me with the opportunity to pursue this work, and gainful employment while doing so. It remains a stimulating and highly enjoyable place to work, and my colleagues are a source of constant inspiration and encouragement. In particular I would like to thank John Tucker, Faron Moller, Magne Haveraaen (visiting), Harold Thimbleby, Min Chen, Matt Jones, George Buchanan, Parisa Eslambolchilar, Monika Seisenberger, and Chris Whyley for their oft-sought and reliable encouragement and advice.

On the technical side, I thank Christian Maeder, Till Mossakowski and Klaus Lüttich for their help and guidance while working with HETS, and the Haskell community in general for being such a friendly and inspirational bunch.

My fellow research students have been a constant source of camaraderie, distraction and sympathy. Particular commendation goes to Will Harwood for his taste in movies, David Chisnall for his remarkable ability to expound on any subject, and Jo Gooch for, well, being Jo. Michaela Heyer, Dan Licata and Alexa Athelstan-Price provided refreshing outside perspectives on research life in their own different ways. Since, it seems, research isn't all drinking coffee and procrastinating, I should also thank Temesghen Kahsai, Liam O'Reilly and Gift Samuel, who have been a great pleasure to work with.

My drumming buddies at Shiko helped keep me sane over these last two turbulent and stress-filled years, and a nicer bunch of people I have yet to meet. Now this work is done, I say to them, "anta na courrou se nenta".

My parents have always believed in me, and from an early age spotted and encouraged my tendency towards 'absent minded Professor' of which this thesis is merely the latest and greatest manifestation. They also very helpfully let me work on this thesis over Christmas instead of doing the dishes. For all of this and more they have my undying love and thanks.

Finally, it is impossible to express adequately within the context of a work such as this my love and appreciation for Basheera, and what her love means to me. The words cannot be written, even with emacs.[1]

[1] http://xkcd.com/378/

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

This thesis is an account of recent work in developing tool support for the specification language CSP-CASL, in particular tools for parsing and static analysis of CSP-CASL specifications. In order to implement such tools it is necessary to formalise these items, therefore this thesis is also a presentation of formal abstract and concrete grammars, and a formal static semantics, for CSP-CASL.

1.1 Motivation and contributions

CSP-CASL [Rog03, Rog06] is motivated by a desire to integrate the specification of data and processes on a deep level. Historically, process algebras have paid little attention to modelling data, whereas algebraic specification languages have not directly supported concurrency; most previous approaches to integrating processes and data [BB87, Bri88, Sca98, FSE06] have restricted data to initial or concrete data types, disallowing loosely specified abstract data and refinement of data specifications. The main exception, μCRL [GP95], allows loose data specification but does not support subsorting or partiality (see section 2.2). Conversely, CSP-CASL allows stepwise development of both processes and data — either aspect may be specified at various levels of abstraction, and refined independently — and subsorting and partiality. This approach was successfully put into practice in [GRS05], where we used CSP-CASL to specify certain aspects of an electronic payment system at differing levels of abstraction (see section 4.3); [OIR07] describes refinement and integrated theorem proving on CSP-CASL, also using the electronic payment system example.

[Rog06] describes the language 'Core-CSP-CASL', concentrating particularly on a presentation of its *model* semantics. Our concern in this thesis is an expanded and tool-supported version of this language, a 'full' CSP-CASL, with formally specified syntax and static semantics.

There are several motivations for formalising CSP-CASL. Primarily, doing so provides an extremely valuable point of reference, of great utility at various stages. When designing the

language, it is very useful to have a language specification, which should be more readily comprehensible and pliable than an implementation, and thus useful as a basis for discussion and experimentation; a *formal* specification is then better than an informal one as it promotes unambiguity and completeness. Furthermore, the very act of formalisation requires us to shine a light into all of the dark corners of the language, and make (and defend) decisions which might otherwise go unconsidered. When implementating the tools, the formalisation provides a reference point which promotes directness of implementation, and increases our confidence that we are implementing what is actually intended. Finally, it also introduces the possibility of properly building multiple tools targetting the same language, which is harder if not impossible where the language definition is informal or consists merely of some 'canonical implementation'.

This work makes the following contributions. Most obviously, it makes the technical contribution of the design and implementation of CSP-CASL's abstract and concrete syntaxes and static semantics; while CSP-CASL is not itself entirely new, it has previously existed only as a 'Blackboard' language, intended for humans and incompletely specified: its formalisation here has necessarily required the resolution of a number of open design issues, and takes CSP-CASL to the state where tool support and automatic processing become possible, with all their associated benefits. This contributes directly to the field of formal methods in a number of ways:

- On the process algebra side, a formal static semantics of a CSP dialect has never been provided before to our knowledge, and doing so brings to light a number of issues; in particular, we provide for the first time a proper and rigourous treatment of the hitherto largely ignored issue of local vs. global variables of CSP processes.
- On the algebraic specification side, this work supports the programme of CoFI, the Common Framework Initiative [Mos96], by formally documenting the syntax and static semantics of a CASL extension language targetting reactive systems – as is required by CoFI for any CASL extension language.

In terms of tool support, we use our formalisation of CSP-CASL in order to extend HETS, a toolset aimed at functional specification of systems, towards full support for specifying systems with reactive/concurrent components. Industrial case studies and applications gain immeasurably from such tool support: in [GRS05] we produced a prototypical specification of an electronic payment system, and proved some properties of our specification on paper; producing a full specification to match or replace that actually used in industry, and to prove properties against it, undoubtedly requires tool support of the kind we are working towards here. The work described in this thesis is included in the HETS distribution, and may be experimented with freely via the HETS online interface[1].

Finally, the implementation part of this work contributes in a small way to the field of parser technology, by providing another case study and set of examples of using the Parsec monadic combinator library for recursive descent parsing in Haskell.

1.2 Related work

Circus [WC01, WC02] is a specification, design and programming language that combines CSP for process descriptions with the language *Z* [Spi89, WD96] for modelling complex data

[1] `http://www.informatik.uni-bremen.de/cgi-bin/cgiwrap/maeder/hets.cgi`

structures and state. Z's (and thus Circus') approach to specification of data differs rather from CASL's: Z takes the *model-oriented* approach, in which a system's behaviour is defined in terms of implicit state and transformations thereon; in contrast, CASL's *property-oriented* approach specifies behavioural properties using a system of axioms over some explicit state (in CASL's case, described algebraically) [HB99].

Circus has a rich set of tools including a parser, static type checker, model checker, and refinement checker; the Circus tools are implemented as extensions to the *CZT* toolkit for Z [MU05], written in Java. The (LALR) parser is automatically generated from a language description using the *JFlex*[2] scanner generator and the *Cup*[3] parser generator and is discussed in [MU05]. [XCS06] presents the static semantics of Circus in a style similar to that seen in chapter 10 of this work, though at a somewhat higher level of abstraction. The rules are then implemented as a CZT extension, utilising the visitor design pattern to define type checking actions for each syntactic category. Circus' approach to implementation of static checks is thus similar to ours for CSP-CASL: a formal specification on paper is implemented by hand in a programming language; our approach to parsing is rather different however, and does not involve automatic parser generation. The Circus parser and type checker have been validated informally using unit tests and by integration with several other tools in the Circus toolset.

In the field of tool support for CSP, the tools FDR (*Failures Divergence Refinement*) [FSE06] and ProBE (*Process Behaviour Explorer*) [FSE03] utilise the input language CSP-M ('Machine Readable CSP'). CSP-M is summarised in [FSE06] and described in detail in [Sca98]. While the latter presents a formal syntax (as an ASCII grammar intended for processing by the `bison`[4] parser generator) and formal model semantics for the language, neither work formalises the *static* semantics of the language. In fact, neither FDR nor ProBE perform explicit type checking, and while using those tools type errors are ordinarily exposed only during model checking or animation [FL08] — though [Sca98] does state that static type-checking is a desirable future improvement. [LZL99] presents a set of type of rules for CSP but they are "incomplete and not adopted by any tools", including FDR [XCS06].

An interesting recent development [FL08] sees CSP-M embedded into the functional programming language Haskell (see chapter 7) at the type-level, allowing CSP-M specifications to be type checked using standard Haskell tools. That is, a CSP-M specification is encoded into a (non-executable) Haskell program, whose type correctness implies the type correctness of the original CSP-M. In contrast, static analysis of CSP-CASL specifications explicitly includes type checks, though our approach differs from that described above in that it is a direct implementation in Haskell of the CSP-CASL static semantics described in chapter 10.

1.3 Thesis outline

The rest of this thesis is organised as follows.

Chapters 2 to 7 present the background material required to contextualise the rest of the work: chapters 2 and 3 introduce the specification languages CASL and CSP respectively, outlining their key features and presenting some simple examples. Chapter 4 then introduces CSP-CASL, concentrating on context and motivation rather than technical details, which come later. In chapter 5 we discuss *heterogeneous specification* and introduce HETS, the Heteterogeneous

[2] `http://jflex.de`
[3] `http://www2.cs.tum.edu/projects/cup/`
[4] `http://www.gnu.org/software/bison/`

Toolset, within which framework our tool is developed. In chapter 6 we introduce the *natural semantics* formalism, used later in the formalisation of CSP-CASL's static semantics. In chapter 7 we introduce the functional programming language *Haskell* and the monadic combinator parsing library *Parsec*, key technologies for the implementation of our tool; here we also briefly review some basic theory behind *recursive descent parsing*, as implemented by Parsec.

Chapters 8 to 12 represent the original contribution of this work, i.e. language design and tool implementation: In chapter 8 we informally describe our target language CSP-CASL, examining its various features, detailing design decisions, and setting the context for the following chapters. In chapter 9 we present the abstract and concrete syntax of CSP-CASL, and discuss some interesting aspects thereof. In chapter 10, we present the formal static semantics of CSP-CASL, using natural semantics, in the style of the CASL Reference Manual. In chapter 11 we examine in detail the problem of checking for *local top elements*, a novel aspect of CSP-CASL's static semantics. In chapter 12 we describe the implementation of our tool as an extension to HETS, outlining and illustrating the implementation strategies for the parser and static analyser within this framework.

Finally, in chapters 13 and 14, we present examples of the tool in action; evaluate the project, the language and the implementation; consider future work; and conclude.

Part I

Background

Chapter 2

CASL— The Common Algebraic Specification Language

Contents

CASL is the *Common Algebraic Specification Language*. In this chapter we present a brief overview of its key features, following [BM04, CoF04c]. Examples are derived from standard ones in [BM04].

2.1 Background

> "CASL, the Common Algebraic Specification Language, has been designed by CoFI, the Common Framework Initiative for algebraic specification and development. CASL is an expressive language for specifying requirements and design for conventional software. It is algebraic in the sense that models of CASL specifications are algebras; the axioms can be arbitrary first-order formulas." [BM04]

Algebraic specification is an approach to formal specification of systems based on the concepts of *universal algebra*, whereby the functional requirements and design of systems are specified as abstract data types (data types and operations thereon) using systems of *signatures* and *axioms* written in some logic (e.g. equational, conditional, first order). [Wag02]

Over the years a number of competing algebraic specification frameworks have been developed, with varying levels of tool support and industrial uptake; major examples include *OBJ* [GWM+93], *ACT-ONE/TWO* [CEW93] and *Extended ML* [KST97]. The CoFI project began in 1995, with the aim of designing a *Common Framework for Algebraic Specification and Development* — an attempt to create a *de facto* standard framework for algebraic specification, providing a "coherent family of languages, all extensions or restrictions of some main algebraic specification language". CASL is the language at the centre of that family (see figure 2.1), based on "a critical selection of the concepts and constructs found in existing algebraic

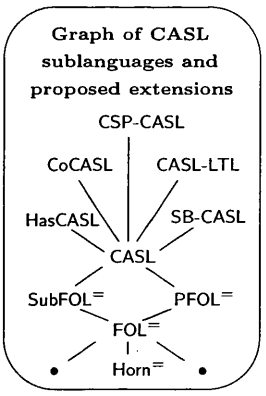


Figure 2.1: Part of the CASL family [Mos04a]

specification frameworks" and targetting "specifying requirements and design of conventional software packages". [Mos96, AS02]

In this chapter we examine CASL further, describing its key features and in particular those of CASL *basic specifications*, and considering its family of sublanguages (restrictions) and extensions, which includes CSP-CASL; we will consider working with CASL and in particular *heteregeneous specifications* involving different languages of the family in chapter 5.

2.2 Overview of CASL

CASL provides basic specifications consisting of many-sorted signatures with subsorting, partial functions, sort generation constraints and axioms written in first order logic. It also provides structured, architectural and library specifications for combining basic specifications in various manners. We now describe these features further, including some illustrative examples, following [BM04].

2.2.1 Basic specifications

A CASL *basic specification* consists of a set of declarations of symbols (i.e. names) for *sorts* (the data types of the specification), symbols and profiles for *operations* (total and partial functions on the sorts), symbols and profiles for *predicates* (relations on the sorts), and a set of *axioms* and *constraints* which restrict the interpretations of the declared symbols. As noted below, such specifications may be *named* for reference in structured specifications and named specifications may be *generic*, i.e. include parametrised elements. The axioms are formulas in two–valued first order logic with equality ($FOL^{=}$), with the usual connectives and universal quantifiers; furthermore, they may make assertions regarding definedness (e.g. of the results of partial functions) and subsorting.

In figure 2.2 we see an example CASL specification, for a *monoid*: a sort and an operation which is associative and has a unit element. This illustrates several key points just mentioned: we have a sort name declaration, two operations (one of which is 0-ary and hence defines a *constant*), and three axioms on the items declared (unit element and associativity).

The semantics of such a specification consists of a *a many-sorted signature* Σ containing the symbols (see section 10.2.1), and a class of Σ-*algebras* corresponding to interpretations of

spec MONOID =
sort *Elem*
ops $n : Elem$;
$\_*\_ : Elem \times Elem \rightarrow Elem$
$\forall x, y, z : Elem$
• $n * x = x$ %(left identity)%
• $x * n = x$ %(right identity)%
• $(x * y) * z = x * (y * z)$ %(associativity)%
end

Figure 2.2: CASL specification example: monoid.

the signature in which the axioms and constraints are satisfied. The *model semantics* of CASL, defining such algebras, is beyond the scope of this document; however, we note that a Σ-algebra contains *carrier sets* corresponding to Σ's sorts, and *functions* and *relations* corresponding to Σ's operations and predicates. In the monoid example in Figure 2.2, the class of Σ-algebras forming this specification model includes the natural numbers under multiplication with an identity of 1, the natural numbers under addition with an identity of 0, and lists under concatenation with an identity of the empty list.

2.2.1.1 Sort generation constraints

CASL includes *sort generation constraints* to further control the contents of the Σ-algebras' carrier sets. The models of a *loose* specification (the default interpretation) include all those with the properties defined by the specification's axioms, without further restraint on the carrier sets (so for example trivial carriers such as singletons will tend to be included); CASL also allows for *generated* and *free* datatypes, however. If a sort is declared as a *generated* datatype, values of the sort are built only using the sort's provided constructors ('no junk' – this provides an induction proof principle on such types); if a sort is declared as a *free* datatype, it is generated and furthermore, values denoted by different constructor terms are necessarily distinct ('no confusion').

The examples in figure 2.3 illustrate loose, generated, and free datatypes:

- SPEC1 and SPEC2 are semantically identical specifications. The **type** declaration in SPEC1 is simply syntactic sugar for the declarations seen in SPEC2 defining a new sort called *Container* and operations (the *constructors empty* and *insert*) on it and *Elem*.
- SPEC3 illustrates the **free type** declaration. Here the sort *Nat* has two constructors, $0 : Nat$ and $suc : Nat \rightarrow Nat$. All free types are generated, so *Nat*s may only be constructed using these operations: the carrier set in any corresponding Σ-algebra contains 'no junk'. However, because this is a free type, we also have 'no confusion': values with differing constructions are automatically distinct. e.g. $suc(0) \neq suc(suc(0))$ in the Σ-algebra, even though we have provided no explicit axioms to that effect.

  SPEC3 also defines an operation $+ : Nat \times Nat - > Nat$, whose meaning is axiomatised using the induction principle provided by the free type. This operation pertains to the generated type example in SPEC4.
- SPEC4 illustrates the **generated type** declaration. Note first that SPEC4 extends SPEC3 via the structuring keyword **then**.

spec SPEC1 =
  sort *Elem*
  type *Container* ::= *empty* | *insert*(*Elem*; *Container*)
end
spec SPEC2 =
  sorts *Elem*, *Container*
  ops *empty* : *Container*;
    insert : *Elem* × *Container* → *Container*
end
spec SPEC3 =
  free type *Nat* ::= 0 | *suc*(*Nat*)
  op __+__ : *Nat* × *Nat* → *Nat*
  ∀ *m*, *n* : *Nat*
  • 0 + *n* = *n*
  • *suc*(*n*) + *m* = *suc*(*m* + *n*)
end
spec SPEC4 = SPEC3
then generated type *Int* ::= __−__(*Nat*; *Nat*)
  ∀ *a*, *b*, *c*, *d* : *Nat*
  • *a* − *b* = *c* − *d* ⇔ *a* + *d* = *c* + *b*
end

Figure 2.3: CASL specification example: sort generation constraints.

The generated type *Int* is then defined by the operation − : *Nat* × *Nat* → *Int*. Again, as this is a generated type, *Int*s may only be constructed using this operation: 'no junk' in the carrier sets. However, as *Int* is *not* a free type, there *can* be 'confusion' in the carriers.

For example, taking the obvious interpretation of *Nat* as $\mathbb{N}$ and *Int* as $\mathbb{Z}$, and writing $0, 1, 2, 3, \ldots$ for the members of *Nat* (an abuse of notation, but a useful and clear one), we find that the *Int*s '1 − 3', '0 − 2' and '5 − 7' are indistinguishable: they are all −2 in $\mathbb{Z}$.

2.2.1.2 Subsorting

A basic specification may include declarations of *subsorts*, in order to capture the idea that instances of the subsort are 'special cases' of the supersort. Subsorts are interpreted not by simple set inclusion but by arbitrary *embeddings* (1-1 functions) between the sorts' corresponding carrier sets. For example, we might have *Char* < *String* ('*Char* is a subsort of *String*'), with disjoint carrier sets for *Char* and *String* but the obvious embedding between *Char*s and 1-element *String*s.

Subsorts are illustrated in the example in figure 2.4: once again we declare a free type *Nat*, plus three types *Car*, *Bicycle* and *Vehicle*, where *Car* and *Bicycle* are subsorts of *Vehicle*, along with some simple operations on these sorts. *Car*s and *Bicycle*s both have a maximum speed and a weight (as they are both *Vehicle*s), but only *Car*s have an engine capacity.

As discussed in chapter 11, subsorting is significant in the static semantics of CSP-CASL, as there is an extra requirement on any subsort relations defined in a CSP-CASL specification,

spec VEHICLE =
  free type $Nat ::= 0 \mid suc(Nat)$
  sorts $Car, Bicycle < Vehicle$
  ops $max\_speed : Vehicle \to Nat$;
    $weight : Vehicle \to Nat$;
    $engine\_capacity : Car \to Nat$
end

Figure 2.4: CASL specification example: subsorts.

namely that it has *local top elements*.

2.2.2 Structured, architectural, and library specifications

Structured specifications build on basic specifications by allowing them to be combined into larger and more complex specifications with the same underlying semantics. Specifically, a structured specification can combine basic specifications, references to *named* specifications, and instances of *generic* (i.e. parametrised) specifications, using constructs of *extension*, *union*, *hiding* and *renaming*. Like a basic specification, the semantics of a structured specification consists of a signature Σ and a class of Σ-algebras. As we shall see in chapter 5, these mechanisms play an important rôle in heterogeneous specification.

Structured specifications allow us to build complex specifications out of simpler ones; *architectural specifications* explicitly describe the intended structure/composition of a system in terms of its components; they target the *design* phase of development, in which the decomposition of a system into components/modules for further development and implementation in target languages is considered. We do not consider architectural specifications any further in this work.

Libraries are "named collections of named specifications", and provide the highest-level organisational mechanism in CASL, collecting specifications into libraries identified by name and version number, with the aim of promoting reuse of specifications. Libraries may be *local* (entirely self-contained) or *distributed* (referring to specifications to be downloaded for inclusion from elsewhere). Hierarchical version numbers support evolution of specifications and libraries of specifications over time (and version control). We will consider libraries of CASL specifications further in chapter 6, as a simple but representative example of how the static semantics of CASL has been formalised using *natural semantics*.

2.3 CASL sublanguages and extensions

As noted above, the CoFI project aimed to create not a single 'one size fits all' algebraic specification language, but rather a coherent family of languages, with CASL at its core (figure 2.1). The other members of the family are either restricted *sublanguages* or *extensions*.

2.3.1 Sublanguages

Sublanguages of CASL are formed simply by restricting certain features of CASL, and are intended for interfacing with other algebraic specification languages and existing tools, and exploring domains for which full CASL is too rich (e.g., term rewriting requires axioms written in equational or conditional logic: full first order logic is too rich). [BM04]

[CoF04c] describes 'A Language for Naming Sublanguages' in which CASL is $SubPCFOL^{=}$, meaning *first-order logic with equality, subsorting, partiality and sort generation constraints* — such names consist of a sequence of feature names, followed by a description of the axioms' expressiveness. Example sublanguages include $FOL^{=}$ (simple first order logic), $Horn_{,}$ (horn clauses, equivalent to Prolog), $SubPHorn^{=}$ ("the positive conditional fragment of CASL"), and $Eq^{=}$ (classical equational logic).

2.3.2 Extensions

Of more immediate relevance to this project, *extensions* add features to CASL, usually but not exclusively at the level of basic specifications. Since, by their nature, extensions add new features to CASL, they are subject to a similar review and approval process to that which produced CASL. In particular their syntax and semantics must be documented in relation to CASL (as we do for CSP-CASL in this document), and the usual CASL syntax and semantics must be respected. Some CASL extensions are:

- CSP-CASL— the subject of this thesis. [Rog03, GRS05, Rog06, KRS07, OIR07]
- HETCASL— heterogeneous CASL, allowing specifications written using a mixture of logics. Since this provides the basis for our tool implementation, and has important ramifications for our language design, we consider it further in chapter 5. [Mos04a, Mos05, MML07a]
- HASCASL — adding features from Haskell, including higher order data types. [SM02]
- COCASL — adding support for *co-algebraic* specification (and allowing algebraic and coalgebraic features to be mixed). [MRRS03, MRS03]

Chapter 3

CSP— Communicating Sequential Processes

Contents

CSP [Hoa85, Ros98, AJS05, Hoa06] is *Communicating Sequential Processes*, a well-known and widely used process algebra, which forms the basis of the reactive/process part of CSP-CASL. In this chapter we introduce the language, present an overview of its key features and syntax, briefly describe the various models of its denotational semantics, and present an example in some detail.

3.1 Background

The process algebra CSP is one of a number of formalisms for modelling and verifying *reactive systems*, i.e. ones which, rather than being characterisable by simple input/output functions, involve *interactions* with other systems, running in parallel, i.e. *concurrently*; process algebra as a field of study arose in reaction to perceived deficiencies of classical theories (i.e. ones based on automata) when dealing with such systems. In particular, where we might refer to the study of systems involving parallel or distributed aspects as *concurrency theory*, process algebra is an approach to this problem characterised by the observation of (usually discrete) behaviour of system components, usually entirely in terms of the inter-process communications which occur. [Bae05]

CSP is one the three process algebras which have historically dominated the field; the others are CCS (*Calculus of Communicating Systems*) [Mil89] and ACP (*Algebra of Communicating Processes*) [BW90]; *Petri nets* [Pet62] are related earlier work, and a recent development of particular note is the *π-calculus* [Mil99], which introduces the notion of process mobility, where the set of processes in a system may vary over time.

Of these, CSP has been particularly successful in the industrial context, often applied successfully in areas as varied as distributed databases, parallel algorithms, train control systems [BS99], fault-tolerant systems [BPS98], and security protocols [RSG+01].

Fixing one syntax, CSP offers a number of approaches to semantics. A process written in CSP may be understood in terms of *operational* semantics (where the process is transformed to a *labelled transition system*, with transitions representing communications); or in terms of *algebraic* semantics (where properties of a process — such as equivalence to some other process — may be deduced by syntactic transformations on the process text following a set of algebraic laws); or in terms of *denotational* semantics (where the process corresponds to a value in some mathematical model, typically a *complete partial order* or a *complete metric space*). Of these three approaches, the latter is dominant; a number of denotational models exist, each of which is dedicated to particular verification tasks, having varying notions of equivalence on processes. We consider some of these models further in section 3.3.

3.2 Survey of CSP features and syntax

This section presents a survey of the key features and operators of CSP, with some examples. Note that this is by no means exhaustive; rather, we aim to provide sufficient context for the discussions of CSP-CASL which follow in chapters 4 and 8. Our explanations largely follow [Ros98].

3.2.1 Basic features

Processes and communications: The basic units of abstraction in CSP are *processes* and *communication events*, or *actions*. Processes are long-lived named entities existing in some environment; communications represent instantaneous (i.e. atomic) synchronisations between processes, usually carrying some semantic content (i.e. a value — a 'message') encoded in the communication's name. A communication may be seen as *a synchronisation the process is willing to engage in*: the various operators on processes are then concerned with defining and modifying the communications which a process offers to the environment.

Channels: Communication names may be prefixed, conceptually representing *channels* 'over which' communication takes place. e.g. the event $login.andy$ conceptually represents communication of the value *andy* over the channel *login*; really, we are just communicating the value $login.andy$. Thus, channels provide a syntactic sugaring — but a very useful one, and they are used widely.

Alphabet: A set of communications is called an *alphabet*; in particular, 'the alphabet of communications' Σ refers to the 'total' alphabet of communications over which a system of processes is defined.

Named and parametrised processes: a *process equation* binds a process to a name, which may be referenced by other processes. Such names may be parametrised; strictly speaking, a parametrised process represents a family of processes, one for each possible set of parameter values. Named processes also allow for recursion (see examples below), and systems of mutually recursive equations.

Primitive processes: CSP defines some primitive processes with fixed behaviour. *SKIP* represents *successful termination*: it never communicates anything, but nobody minds. Conversely,

$STOP$ represents *deadlock*: it represents a process which has entered a state of perpetual noncommunication, where this is *not* the desired outcome. We also have DIV, representing *divergence*: this is similar to $STOP$ in that it is perpetually noncommunicative, from the environment's point of view; however, DIV represents *livelock*: it is engaging in an infinite sequence of internal, non-observable actions. Some models of CSP are capable of distinguishing between $STOP$ and DIV; others are not.

3.2.2 Prefix operators

Action prefix: The most fundamental communication operator is action prefix. The process $a \rightarrow P$ offers the communication a, and then behaves like the process P. For example, the process YES:

$$YES = yes \rightarrow YES$$

is always willing to communicate the value yes. A simple parametrised process example is $TICK$:

$$TICK(0) = 0 \rightarrow TICK(0)$$
$$TICK(n) = n \rightarrow TICK(n-1)$$

For any $n > 0$, $TICK(n)$ will communicate n then behaves like $TICK(n-1)$; the special case $TICK(0)$ will communicate 0 but then just behaves like $TICK(0)$ again. Thus, $TICK(n)$ counts down from n to 0, then stays at 0. Note that we have not explicitly stated that $n \in \mathbb{N}$, leaving it 'obvious' in this case. Of course, while this is fine 'on the blackboard', for tool processing more rigour is required; the problem of defining the data types communicated by processes, and doing so in a rich, flexible, and well-founded manner, is exactly the problem CSP-CASL is intended to solve.

Prefix choice: If X is a set of communications, then $?x : X \rightarrow P(x)$ is a process which will communicate any value $x \in X$ and then behave like $P(x)$; thus, this operator allows a choice of values to be communicated. Then $c?x \rightarrow P(x)$ is the same thing but 'over a channel' (i.e. we communicate $c.x$). In cases where we are nominally 'sending' a value over a channel (strictly all CSP communications are bidirectional synchronisations: properly, the 'sending' process is *choosing* which value is to be synchronised on), we write $c!x \rightarrow P(x)$ as a synonym for $c.x \rightarrow P(x)$.

3.2.3 Choice operators

External choice: The process $P \,\Box\, Q$ offers the environment the choice of the first communications of P and Q, and then behaves accordingly. For example, consider:

$$ECH = a \rightarrow SKIP \,\Box\, b \rightarrow c \rightarrow ECH$$

If the environment offers a, ECH will communicate a then terminate. Conversely, if the environment offers b, ECH will communicate b, then offer *only* c, after which it behaves like ECH again (i.e. offers a choice of a and b). Thus, ECH in some sense 'recognises' the regular expression $(bc)^*a$. n.b.: if both sides offer the same communication, the choice of which side is taken is *nondeterministic*.

Internal choice: The process $P \sqcap Q$ can behave either like P or like Q, where the choice is made internally to the process (and is thus, from the point of view of the environment, nondeterministic). Consider:

$$ICH = a \rightarrow SKIP \sqcap b \rightarrow c \rightarrow ICH$$

If the environment offers b, ICH may choose to behave like $a \rightarrow SKIP$, and not engage in the b communication offered (and vice versa); only if the environment offers both a and b is ICH obliged to communicate, because it must choose one of the alternatives.

3.2.4 Parallel operators

Interleaving: The process $P \,|||\, Q$ is a process where P and Q run in parallel, independent of one another; if the environment offers a communication which both P and Q could engage in, exactly one does so, the choice being nondeterministic. Consider:

$$AB = a \rightarrow b \rightarrow AB$$
$$CA = c \rightarrow a \rightarrow CA$$
$$INT = AB \,|||\, CA$$

INT may initially engage in a or c; supposing it engages in a, it *may still* engage in c subsequently: it presents a choice of b and c. Conversely, if it initially engages in c, then it can subsequently only engage in a, but after that it might offer a and b, or a and c, depending on 'which a' was (nondeterministically) chosen.

Synchronous parallel: The process $P \parallel Q$ is a process which behaves like P and Q in total synchrony with each other; that is, it only communicates events on which they both agree.

Generalised parallel: If A is a set of communications then $P \,[\![X]\!]\, Q$ behaves like $P \,|||\, Q$ except for events in X, on which P and Q must synchronise. For example, supposing $X = \{b, c\}$:

$$AC = a \rightarrow AC \sqcap c \rightarrow AC$$
$$ACAC = AC \,[\![X]\!]\, AC$$

AC can choose internally whether to communicate a or c; two copies are placed in parallel but must synchronise on everything in X; if one copy chooses to communicate c, which is in X, they must both do so. Clearly, then, $ACAC$ can deadlock, if one side offers a, and the other offers c.

Alphabetised parallel: If A and B are sets of communications then $P \,[\![A \mid B]\!]\, Q$ behaves like $P \,|||\, Q$ except for events in $A \cap B$, on which P and Q must synchronise.

3.2.5 Other operators

Sequential composition: The process $P \,;\, Q$ is a process which behaves like P then behaves like Q. It is useful for composing named processes, promoting modularity. For example:

$$A = a \rightarrow SKIP$$
$$B = b \rightarrow SKIP$$
$$AB = A \,;\, B$$

Here AB is equivalent to $a \rightarrow b \rightarrow SKIP$.

Conditional choice: if b is some boolean valued expression, then *if* b *then* P *else* Q behaves like P if b is true, and like Q otherwise — just as one would expect.

Hiding: if X is a set of communications then $P \setminus X$ is a process which behaves like P except that any event in X is not observable from outside $P \setminus X$ (though it may still take place).

Renaming: if f is an injective function $f : \Sigma \to \Sigma$ then $P[f]$ is a process which behaves like P except that all its communications are subjected to the transformation f before being made observable to the environment. (Note that there is also a *relational renaming* form, not shown here.) For example, with $\Sigma = \mathbb{N}$ and $f(x) = 2x$, consider:

$$COUNTUP(n) = n \to COUNTUP(n+1)$$
$$EVENS = COUNTUP(0)[f]$$

Here, $COUNTUP(0)$ would communicate the sequence $0, 1, 2, 3, \ldots$ whereas $EVENS$, which applies the renaming f, communicates the sequence $0, 2, 4, 6, \ldots$

Run and **Chaos**: [Ros98] also presents two further processes "with very basic behaviour that it is useful to have standard names for", namely RUN and $CHAOS$:

> For a set of events $A \in \Sigma$, RUN_A can always communicate any member of A desired by the environment.
>
> For a set of events $A \in \Sigma$, $CHAOS_A$ can always choose to communicate or reject any member of A.

3.2.6 Machine readable CSP

CSP as presented in [Ros98] and outlined above is a 'blackboard' language whose "intended audience is human" [Sca98], i.e. its syntax is more like 'general mathematics' than a machine readable language. Naturally this leads to rich expressivity and supports the development of related theory. However, CSP's practical success is also founded on well developed tool support, in particular the process animation tool ProBE [FSE03] and the model checker FDR [FSE06]. In order to support such tools, [Ros98] introduces the formal language CSP-M, *Machine Readable* CSP, which is described in detail in [Sca98] (with a formal syntax, but no static semantics).

Data in CSP-M is defined using a purely functional programming language with a strong static type system, requiring explicit type declarations for channels and data types. We refrain from presenting here a survey of its syntax, but it should be noted that it forms an important reference point for our design of the concrete syntax of CSP-CASL: where possible, and where questions arose, we have kept an eye on tool interoperability between CSP-M and CSP-CASL, and attempted to conform to this 'conventional form' familiar to CSP practitioners everywhere.

For a concrete example, consider the specification `simple.csp`, in figure 3.1. This example, taken from the FDR[1] distribution, specifies a single-place buffer implemented over two channels, `left` and `right`; indeed, it includes two specifications of such a buffer, and asserts that they are equivalent. Note the order of items: data types, channels, then processes — in CSP-CASL, by design, all specifications follow this order.

[1] http://www.fsel.com/software.html

```
-- First, the set of values to be communicated
datatype FRUIT = apples | oranges | pears

-- Channel declarations
channel left,right,mid : FRUIT
channel ack

-- The specification is simply a single place buffer
COPY = left ? x -> right ! x -> COPY

-- The implementation consists of two processes communicating over
-- mid and ack
SEND = left ? x -> mid ! x -> ack -> SEND
REC = mid ? x -> right ! x -> ack -> REC

-- These components are composed in parallel and the internal comms hidden
SYSTEM = (SEND [| {| mid, ack |} |] REC) \ {| mid, ack |}

-- Checking "SYSTEM" against "COPY" will confirm that the implementation
-- is correct.
assert COPY [FD= SYSTEM

-- In fact, the processes are equal, as shown by
assert SYSTEM [FD= COPY
```

Figure 3.1: CSP-M example: `simple.csp`

- First we define `FRUIT`, an enumerated type — which in CSP-CASL we would specify as a free type in the data part; (in CSP-CASL we would also have the option to keep `FRUIT` *loosely specified*). Note the use of '`--`' for a comment.
- Channels `left`, `right`, and `mid` communicate values of type `FRUIT`; the channel `ack` is singleton-typed.
- The abstract specification `COPY`, states that a buffer's behaviour is to repeatedly read some value on channel `left`, bind it to the local variable `x`, and then write it out on channel `right`,
- Then `SYSTEM` is a more concrete specification, describing one possible implementation of `COPY`; here the 'receive from left' and 'send to right' parts are separate processes, synchronising on the channel `mid` and using the channel `ack` to proceed in lockstep; the channels `mid` and `ack` are then *hidden* from everything outside `SYSTEM`:
- Finally, we assert first that `SYSTEM` refines `COPY`, and then the reverse, so that they are in fact entirely equivalent in behaviour; these are *proof obligations*: it is precisely assertions like these which FDR exists in order to check.

3.3 The Models $\mathcal{T}$, $\mathcal{F}$ and $\mathcal{N}$

Here we briefly introduce the three dominant models of CSP's denotational semantics; each model has varying views on two interesting questions regarding processes, namely 'are two processes equal?' and 'what properties does a process possess?' (e.g. does it deadlock?).

The **Traces model**, model $\mathcal{T}$, denotes a CSP process according to its finite *traces*, which is the set of finite sequences of communications in which it may engage; this corresponds to observing a system solely in terms of the communication events which occur.

$traces(P)$ may be computed recursively using a set of rules, a few of which are:

$$\begin{aligned} traces(STOP) &= \{\langle\rangle\} \\ traces(a \rightarrow P) &= \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid s \in traces(P)\} \\ traces(P \,\Box\, Q) &= traces(P) \cup traces(Q) \\ traces(P \sqcap Q) &= traces(P) \cup traces(Q) \\ traces(P \parallel Q) &= traces(P) \cap traces(Q) \end{aligned}$$

Thus: $STOP$ never communicates anything: its set of traces consists only of the empty trace $\langle\rangle$; the traces of an action prefix process are the traces of the prefixed process P, each prefixed with the event a first communicated; etc.

Two processes P and Q are then *traces-equivalent*, $P =_{\mathcal{T}} Q$, if $traces(P) = traces(Q)$. In this sense, the model $\mathcal{T}$ is the weakest of the three presented here. It considers equivalent many processes which are distinguished by the other models. To give a simple example, consider:

$$\begin{aligned} ECHY &= a \rightarrow STOP \,\Box\, b \rightarrow STOP \\ ICHY &= a \rightarrow STOP \sqcap b \rightarrow STOP \end{aligned}$$

Here we have $ECHY =_{\mathcal{T}} ICHY$, as $traces(ECHY) = traces(ICHY) = \{\langle\rangle, \langle a\rangle, \langle b\rangle\}$ — the model $\mathcal{T}$ cannot distinguish between internal and external choice.

The **Stable Failures** model, model $\mathcal{F}$, denotes a process by its *failures*. A failure is a pair (s, X) where s is some member of $traces(P)$ and X (called a *refusal*) is a set of events which the process can refuse after s. So, after trace s, P can refuse do anything from the set X. $failures(P)$ is the set of P's failures.

The stable failures model can distinguish between internal and external choice (and much else besides), and in particular, allows us to detect *deadlock*. In practice, however, the stable failures model is mainly used to prove Failures-Divergence refinement for processes which are known to be divergence free — see below.

Finally, the **Failures-Divergence** model, model $\mathcal{N}$, allows us to detect processes which might **livelock** (enter a state in which they can perform an infinite sequence of internal actions, which are not visible from outside the process).

In this model, a process is represented by its failures and also its set of *divergences*, where a divergence is a trace after which the process may perform an infinite sequence of consecutive internal actions. The model $\mathcal{N}$ is used for proving safety and liveness properties of processes (such as freedom from *starvation* as illustrated by the Dining Philosophers example in section 3.4). It is the 'standard' model of CSP, since it provides the most detailed and useful information about a process' possible behaviour.

The name of the CSP tool FDR stands for 'Failures-Divergence Refinement', reflecting the central rôle of the model $\mathcal{N}$ in its operation. Here *refinement* refers to the question of whether a process P is 'better' than some other process Q in the sense that its behaviour is indistinguishable (within a given model), but that it is more concrete, i.e. is less nondeterministic. Then we might write $Q \sqsubseteq P$, pronounced 'P refines Q'.

We generally want processes to be divergence free, in practical situations at least. Where that is the case, Failures-Divergence refinement and Stable Failures refinement are identical.

3.4 Example: the Dining Philosophers

[Ros98, §2.2] formulates the well known *Dining Philosophers Problem* [Dij71] in CSP; we end this chapter by presenting the same example here, exploring its behaviour in some detail.

To briefly review: five (or, in general, n, though the problem is trivial with $n < 5$) philosophers are sat about a round table on which is a bowl of food; between each pair of adjacent philosophers is a fork — each fork is shared between two philosophers; the philosophers alternate unpredictably between thinking and eating; to eat, a philosopher must first pick up the two forks on either side of them; when they have finished eating, both forks are returned to the table; critically, only one fork can be picked up or put down at once.

The n-philosopher version may be modelled as a system of n processes modelling the philosophers, and n processes modelling the forks. The operators $\oplus$ and $\ominus$ represent addition and subtraction, respectively, *modulo* n.

A fork, then, is something which is repeatedly picked up and put down, either by the philosopher to its right, or the one to its left. This may be modelled in CSP as:

$$\begin{aligned} FORK_i = &(picksup.i.i \rightarrow putsdown.i.i \rightarrow FORK_i) \\ &\Box\ (picksup.i\ominus 1.i \rightarrow putsdown.i\ominus 1.i \rightarrow FORK_i) \end{aligned}$$

Consider the first branch of the choice, i.e. $picksup.i.i \rightarrow putsdown.i.i \rightarrow FORK_i$:

1. Offer the value $i.i$ on the *picksup* channel; this represents philosopher i (to the right of fork i) picking up fork i.
2. Offer the value $i.i$ on the *putdown* channel; this represents philosopher i putting down fork i.
3. Behave as the process $FORK_i$ (i.e. loop, represented recursively).

Now, $\Box$ is external choice: $FORK_i$ might behave as just described, or it might behave like the second branch, i.e. as $picksup.i\ominus 1.i \rightarrow putsdown.i\ominus 1.i \rightarrow FORK_i$. This behaves very similarly to the above, except that it communicates $i\ominus 1.i$ rather than $i.i$ — representing being picked up and put down by the philosopher *to the left*.

$FORK_i$ has no say about which branch is taken: the outcome depends entirely on its environment. It offers both $picksup.i.i$ *and* $picksup.i\ominus 1.i$, then behaves according to the choice made externally. It is, then, the philosophers who make the choice — of course.

There are many ways the philosophers might behave: at its heart the Dining Philosophers Problem is about finding the right algorithm so that *deadlock* (where every philosopher just waits forever for one of their neighbours to put down a fork) and starvation (where some particular philosopher never gets to eat) are avoided.

An obvious and naïve algorithm, which can indeed lead to deadlock is as follows: when a philosopher wants to eat, they first attempt to pick up the fork to their left, then the one to their right; they eat, then replace the forks in the reverse order. To encode this algorithm, the i^{th} philosopher is represented as the following CSP process:

$$\begin{aligned} PHIL_i = &thinks.i \rightarrow picksup.i.i \rightarrow picksup.i.i\oplus 1 \rightarrow \\ &eats.i \rightarrow putsdown.i.i\oplus 1 \rightarrow putsdown.i.i \rightarrow PHIL_i \end{aligned}$$

1. Offer the value i on the channel *thinks*; this represents philosopher i starting to think.

2. Offer the value $i.i$ on the channel *picksup*; as noted above, this represents philosopher i picking up fork i, the fork to their left. This will only occur if $FORK_i$, the only fork process capable of engaging in this communication, does so; it need not: it might have been picked up recently by philosopher $i \ominus 1.i$, the philospher to *its* left. In that case $PHIL_i$ waits for it.
3. Offer the value $i.i\oplus 1$ on the channel *picksup*; this communication represents philosopher i picking up fork $i\oplus 1$, the fork to their right. Again, $PHIL_i$ might need to wait, this time for $FORK_{i\oplus 1}$.
4. Offer the value i on the channel *eat*; this represents philosopher i starting to eat.
5. Offer the value $i.i\oplus 1$ on the channel *putsdown*; as noted above, this represents philosopher i putting down fork $i\oplus 1$, the fork to their right. In this case $PHIL_i$ should never have to wait for $FORK_{i\oplus 1}$, at least.
6. Offer the value $i.i$ on the channel *putsdown*; this represents philosopher i putting down fork i, the fork to their left. Again, this should not involve any waiting.
7. Behave as the process $PHIL_i$ (i.e. loop, represented recursively).

Having read the above, the reader will surely agree that the CSP process $PHIL_i$ is a pleasingly compact representation of the behaviour described here: easier (when familiar with the formalism) to grasp and manipulate — in other words, a good abstraction.

Now, the complete system is formed by putting all of these processes in parallel with each other (not shown here)[2]. For $n = 5$ this forms a system of 10 processes, where each $FORK$ process communicates only with 2 particular $PHIL$ processes, and each $PHIL$ process communicates only with 2 particular $FORK$ processes. A trace leading directly to deadlock is then:

$$\langle thinks.0, picksup.0.0, thinks.1, picksup.1.1, thinks.2, picksup.2.2, thinks.3, picksup3.3, thinks.4, picksup.4.4\rangle$$

After this trace, $PHIL_1$ is offering (only) $picksup.1.2$, but $FORK_2$, the only fork process capable of communicating this value, is offering (only) $putsdown.2.2$, which can only be communicated by $PHIL_2$, which is waiting for $picksup.2.3$, only possible from $FORK_3$, which is offering only $putsdown.2.3$, only possible from $PHIL_3$, which is waiting for ..., etc. — in a circular 'deadly embrace'.

[2]Properly, we also need environment processes capable of engaging in *thinks* and *eats* communications with the philosophers, but these may be abstracted away without loss of generality.

Chapter 4

CSP-CASL

Contents

In this chapter we introduce CSP-CASL, the combination of CASL and CSP with which this thesis is concerned. We present a brief overview of the language's main features (following [Rog03, GRS05]), present and discuss some simple examples from [Rog06], and conclude with a larger example, from an industrial case study [GRS05]. A full description of the language may be found in chapter 8.

4.1 Overview

CSP-CASL is a comprehensive language which combines the specification of *data types* in CASL (see chapter 2) with *processes* written in CSP (see chapter 3). The general idea of this language combination is to describe reactive systems in the form of processes based on CSP operators, but where the communications between these processes are the values of data types, which are loosely specified in CASL. All standard CASL features are available for the specification of these data types, namely many-sorted First Order Logic with equality, sort-generation constraints, partiality, and subsorting. Furthermore, the various CASL structuring constructs can be used to describe data types within CSP-CASL. For the description of processes, the typical CSP operators are included in CSP-CASL: there are for instance internal choice and external choice; the various parallel operators such as interleaving, alphabetized parallel, and generalised parallel; communication over channels is included, where channels are a category of symbols with sorts defined in CASL. Similarly to CASL, CSP-CASL specifications can be organised in libraries. Indeed, it is possible to mix CASL specifications and CSP-CASL specifications in one library, separating the development of data types in CASL from their use within CSP-CASL — see chapter 5.

Syntactically, a CSP-CASL specification with name N consists of a data part Sp, which is a structured CASL specification, an (optional) channel part Ch to declare channels, which are typed according to the data specification, and a process part P, written in our CSP dialect,

within which CASL terms are used as communications; CASL sorts denote sets of communications; renaming is described by a binary CASL predicate and functions; and the CSP conditional construct uses CASL formulae as conditions.

In the process part, recursive process definitions may be written using systems of process equations, binding processes to process names. Processes can also be parametrised with variables typed by CASL sorts. In general, this combination of recursion and parameterisation leads to an infinite system of process equations.

As a consequence of CASL's loose semantics, *semantically* a CSP-CASL specification is a family of process denotations for CSP processes, where each model of the data part Sp gives rise to one process denotation. The definition of CSP-CASL is generic in its choice of CSP semantics. For example, all denotational CSP models mentioned in [Ros98] are possible parameters.

4.2 CSP-CASL examples

We now consider four simple example CSP-CASL specifications, all taken from [Rog06]. Note that each of these examples has been parsed, statically analysed, and pretty-printed to the LaTeX source rendered below, by our tool — as have all of the CSP-CASL examples in this thesis.

4.2.1 Example 1: TCS1

The specification TCS1 is a particularly simple example, but nonetheless illustrates several key features:

spec TCS1 =
  data sorts S, T
    ops $c : S; d : T$
  process $tcs1 : S, T$;
    $tcs1 = c \rightarrow SKIP \;||\; d \rightarrow SKIP$
end

As noted in section 4.1, syntactically we have a *data part* and a *process part*, indicated by the **data** and **process** keywords, respectively. The data part contains a CASL basic specification defining two sorts, S and T, and two constants of those sorts, c and d. The process part contains a *process declaration* and a *process equation*. The declaration declares that $tcs1$ is the name of a process which takes no parameters, and communicates values in the sorts S and T. The equation then binds the process $c \rightarrow SKIP \;||\; d \rightarrow SKIP$ to that name. Thus, $tcs1$ is a process which potentially could communicate c or d. As terms of different sorts are considered unique, however, $tcs1$ is in an immediate deadlock, i.e. it is equivalent to $STOP$.

A key point to note here, with respect to the discussion of CSP in chapter 3, is that whereas CSP processes are not associated with particular alphabets (see in particular [Ros98, §2.7]), CSP-CASL processes *are*; this point is discussed further in section 14.3.10.

4.2.2 Example 2: TCS2

The next example, TCS2, illustrates CASL subsorting in the CSP-CASL context:

spec TCS2 =
  data sort $S < T$
    ops $c : S; d : T$
    • $c = d$
  process *tcs2* : T ;
    $tcs2 = c \rightarrow SKIP \mid\mid d \rightarrow SKIP$
end

This is very similar to the TCS1 example in section 4.2.1, except that in the data part, S is now declared to be a subsort of T (see section 2.2.1.2), with an axiom restricting c and d to the same value. The most interesting thing to note from our point of view is that in the process part, the alphabet of $tcs2$ contains only T, even though $tcs2$ may communicate c, of sort S; this is fine because S is implicitly also in the alphabet, because it is a subsort of T. We will see later that subsorting has a profound effect on our treatment of process alphabets (in particular, see section 10.2.4.1). In terms of the meaning of $tcs2$, it will engage in one event c and then terminate.

4.2.3 Example 3: TCS3

The next example, TCS3, illustrates partiality in the data part.

spec TCS3 =
  data sorts S, T
    op $f : S \rightarrow ? T$
    • $\forall x : S \bullet \neg\, def\, f(x)$
  process *tcs3* : S, T ;
    $tcs3 = \Box\, x :: S \rightarrow f(x) \rightarrow SKIP \,[\![T]\!]\, \Box\, y :: T \rightarrow$ **if** *def* y **then** *SKIP* **else** *STOP*
end

The operation f is defined as *partial*, i.e. not necessarily defined for all values of sort S; in fact, an axiom declares that it is undefined for *all* values of S. With respect to CSP-CASL parsing and static semantics, this is irrelevant in this example — though it has a significant effect in the model semantics, of course. The process $tcs3$ might first engage in an arbitrary element of sort S, then communcate the undefined element of sort T, and finally end in a deadlock situation. From our point of view, then, the points of interest here are:

- A generalised parallel operator: $[\![T]\!]$ — the processes on either side must synchronise on any element of T they wish to communicate;
- Two external prefix choice operators: $\Box\, x :: S \rightarrow \ldots$ and $\Box\, y :: T \rightarrow \ldots$

  These correspond to *prefix choice* in CSP — see section 3.2.2. The first one engages in an element of sort S with the environment; the second synchronises with the $f(x)$ on the left hand side.
- A prefix operator, communicating a value expressed as a CASL term: $f(x) \rightarrow \ldots$

  Note two things: the CASL term refers to a (partial) function, and we pass as a parameter the locally bound S-sorted value x.
- A conditional choice operator: **if** *def* y **then** $SKIP$ **else** $STOP$

  The conditional is controlled by a CASL formula, in this case one checking the definedness of the CASL term y (a simple term in this case, consisting only of the T-sorted value

y) — an undefined y value might be produced by an application of the partial function f, for example. See section 10.2.4.2 for further discussion of issues relating to definedness.

4.2.4 Example 4: TCS4

The final example from [Rog06], by comparison, is once again very straightforward in the process part, and mainly of interest from a model semantics point of view; however, as we shall see, there is in fact a new and interesting static semantics requirement present here:

spec TCS4 =
  data sorts $A, B, C < S$
    ops $a : A; b1, b2 : B; c : C;$
      $f : A \to ? A; g : C \to ? C$
    • $a = b1$ • $b2 = c$
    • $\forall x : A \bullet \neg\, def\, f(x)$ • $\forall x : C \bullet \neg\, def\, g(x)$
  process $tcs4 : A, C$;
    $tcs4 = f(a) \to SKIP \;||\; g(c) \to SKIP$
end

The specification illustrates subsorting and partiality together, with sorts A, B and C all subsorts of S, and $f(x)$ and $g(x)$ always undefined. The purpose of the example is to consider synchronisation of undefined values: $f(a)$ and $g(c)$ are undefined, so the question is do they ever synchronise? According to CSP-CASL's model semantics, undefined values synchronise only if they belong to sorts having a common supersort; in this case, they do, so the process $tcs4$ will terminate. On the theoretical side, this gives rise to the requirement for *local top elements*, which may be checked statically, and which is explored in depth in chapter 11.

4.3 Case study: EP2

We conclude this chapter with a larger example, drawn from an industrial scale case study. In [GRS05] (from which this section draws heavily) we described the formal specification of a banking system using CSP-CASL. The system in question is called EP2, which stands for *EFT/POS 2000*, short for 'Electronic Fund Transfer/Point Of Service 2000'; it is a joint project established by a consortium of (mainly Swiss) financial institutes and companies in order to define EFT/POS infrastructure for credit, debit, and electronic purse terminals in Switzerland[1]. EP2 builds on a number of other standards, most notably EMV 2000 (the Europay/Mastercard/Visa Integrated Circuit Card standard[2]) and various ISO standards.

An overview of EP2 is shown in figure 4.1. The system consists of seven autonomous entities centred around the EP2 *Terminal*, which is a hardware device concerned with processing card details and authorising financial transactions. The other entities are the *Cardholder* (i.e. customer), *Point of Service/POS* (i.e. cash register), *Attendant*, *POS Management System/PMS*, *Acquirer*, *Service Center*, and *Card*. These entities communicate with the Terminal and, to a certain extent, with one another via XML messages in a fixed format over TCP/IP. These messages contain information about authorisation, financial transactions, and initialisation and status data. The state of each component heavily depends on the contents of the exchanged

[1] www.eftpos2000.ch
[2] www.emvco.com

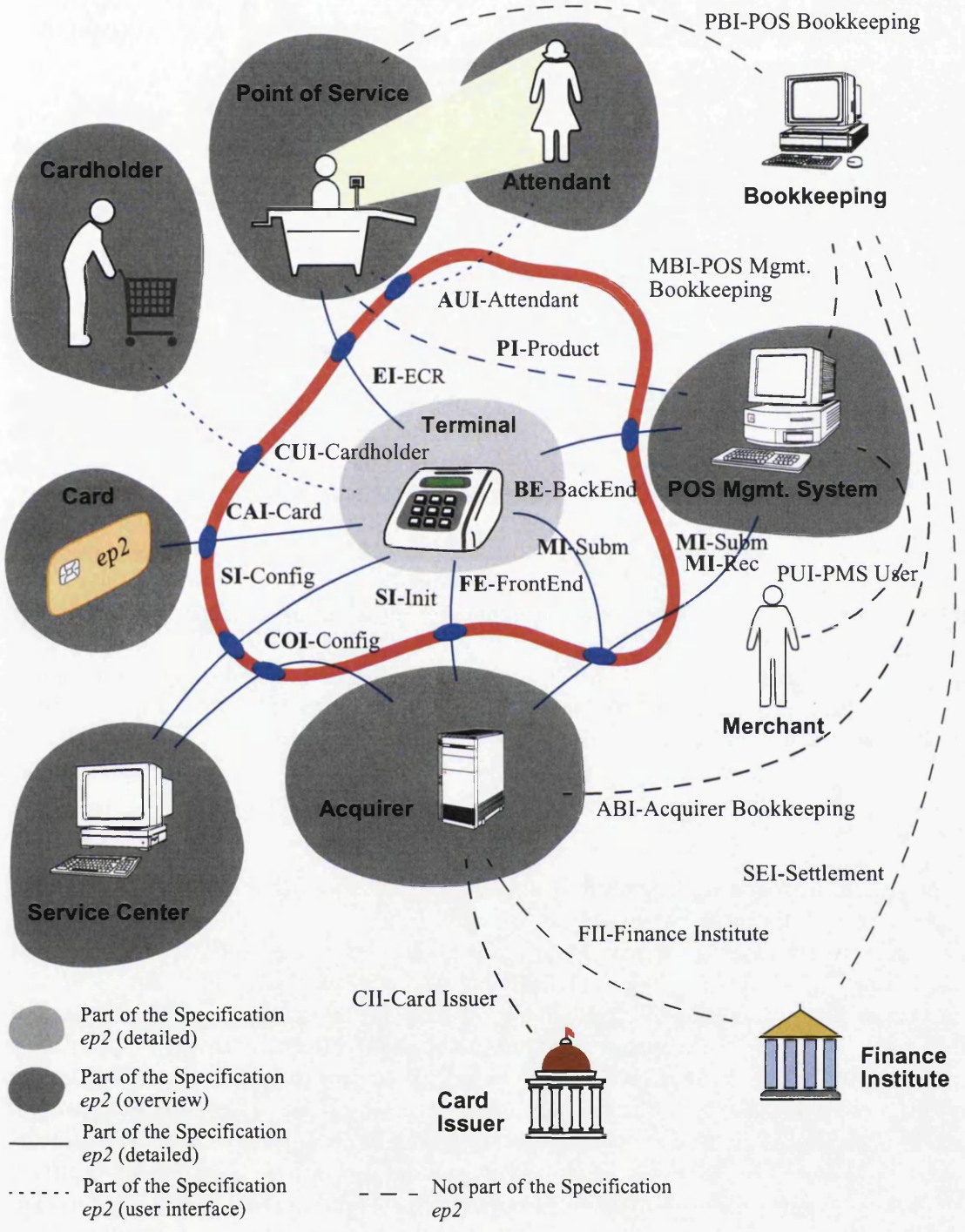


Figure 4.1: Overview of the EP2 system, from [EP202].

data. Each component is thus a reactive system, and there are both reactive parts and data parts which need to be modelled, with these parts heavily intertwined.

The specification is available (for a fee) to anyone who wants to build a Terminal, which would then be tested and certified by Zühlke Engineering AG. The EP2 project began in October 2000, and was officially completed on May 31 2003, in that Terminals with full EP2 functionality had been built and tested, and released for use in the field.

Now, the EP2 specification consists of twelve documents, each of which either considers some particular component of the system in detail, or considers some aspect common to many or all components. The Terminal, Acquirer, POS, PMS and Service Center components all have specification documents setting out *general*, *functional*, and *supplementary* requirements, where the functional requirements carry the most detail, and consist mainly of use cases discussing how that particular component behaves in various situations. As well as the specifications of particular components, there is a *Security Specification*, an *Interface Specification* and a *Data Dictionary*. The Security Specification discusses the various security mechanisms used within EP2, many of which are inherited from EMV and other related specifications. The Interface Specification describes every interface between components in terms of the messages allowed on the interface, and in what order they may flow. The Data Dictionary describes in more detail the data elements found in messages and, in some cases, in the state of some component. It includes brief discussion of every data element occurring in the system.

One obvious characteristic of such a document structure is that, when considering some aspect of the system, the information required to understand that aspect is contained in several different documents, each of which has its own things to say about the situation in question. For example, in order to gather all information about the `SI-Init` interface between Terminal and Acquirer, one has to examine the Terminal Specification, the Acquirer Specification, the Interface Specification, and the Data Dictionary. The documents are written using a number of different specification notations: plain English; UML-like graphics (use cases, activity diagrams, message sequence charts, class models, etc.); pictures; tables; lists; file descriptions; encoding rules. In these regards it is entirely typical of a modern industrial strength system specification.

In general, as we found, this approach results in a specification which is difficult to understand, and which easily leads to inconsistencies and ambiguities. As such, this is exactly the kind of system we would like to specify formally, with all the associated benefits of tool support that this brings. At the time at which [GRS05] was written, such tool support for CSP-CASL was unavailable; nonetheless, we were able to specify in CSP-CASL a number of aspects of the system, at varying levels of abstraction, and for example perform (relatively simple) proofs of properties such as deadlock-freedom 'on paper'. The major part of the motivation for the efforts described in this thesis is to work towards a tool allowing such analyses to be automated, and thus performed on a larger scale — as is required in order to be of real benefit in the industrial context. Nonetheless, even without such tool support, we were able to identify a number of ambiguities and even contradictions in the EP2 specification: the very act of writing specifications in a formal language required a level of rigour at which such problems were forced into the light.

In figure 4.2 we see a CSP-CASL specification of the top-level 'architectural' view of EP2— the same view of the system illustrated in figure 4.1. In particular, it specifies at a high level each of the nine interfaces represented by solid blue lines in that diagram (CAI-Card, SI-Config, COI-Config, SI-Init, FE-FrontEnd, MI-Subm, MI-Rec, BE-BackEnd, EI-ECR).

The first thing to note is that data and processes are specified separately: we have a CASL specification describing the data, which is then referenced by the process part (see section 5.4).

The data specification defines sorts describing the data communicated on each of the interfaces listed above; this is loose specification, where in fact all we are doing is defining a name for each sort. Later, we would expect to refine this specification, writing a new one where these data types are constrained, and proving that it refines this specification; a chain of such refinements leads to a specification of a concrete system.

There are actually two interfaces between the Acquirer and the PMS; this is modelled in the specification as a single interface carrying two data types, whose sort is then a free type, *MI_Subm_or_Rec* enumerating these possibilities. An alternative strategy would be to treat the two interfaces as two separate channels, each with their own sort.

The process specification begins by declaring channels representing each of the interfaces; each channel's sort comes from the data part.

This is followed by declarations of processes for each of the six system components involved in these communications, giving each process an alphabet consisting of the channels over which it may communicate. For example, the *ServiceCenter* process may communicate on the *cSI_Config* and *cCOI_Config* channels.

This is followed by process equations defining the behaviour of each component's process as a CSP-CASL process term. Operating, like the data part, at almost the highest level of abstraction possible, we use the *RUN* process to simply allow each process to communicate any value over any of its channels, freely, forever. For example, the *Card* process can communicate all values of channel *cCAI_Card*; the *ServiceCenter* process communicates on two channels, which we model as two processes, one per channel, running freely in parallel using the interleaving operator. A slightly more abstract view here would be $Run(cSI\_Config, cCOI\_Config)$, which does not model the fact that these two interfaces really are disjoint.

Finally, we declare and define a process, *ep2*, representing the entire system. Its communication alphabet consists of all the channels we have defined. Its process is slightly complicated, however. The most abstract view would simply put *Card*, *ServiceCenter*, etc. in parallel with each other using interleaving; this would represent a broadcast environment in which each component ignores messages on channels associated with interfaces it does not use. Instead, we have encoded aspects (though not all) of the point-to-point communication structure of the system explicitly, as follows.

The *Terminal*, sitting at the centre of the system, can in fact communicate, on any channel *except COI_Config* and *MI_Rec*, with the rest of the system. 'The rest of the system' is then modelled as three procsses interleaved: the Card, the POS, and a process in which the communications between the ServiceCenter, Acquirer, and PMS are restricted; for example, here the ServiceCenter cannot communicate with the PMS whatsoever.

This is, then, a very abstract view of the system — but certainly not a trivial one, and one on which we could perform, for example, deadlock analysis. For another example specification from our work on EP2, see section 13.4.

logic CASL
spec EP2_DATA =
sorts *CAI_Card*; *SI_Config*; *SI_Init*; *FE_FrontEnd*; *MI_Subm*; *BE_BackEnd*; *EI_ECR*;
COI_Config; *MI_Rec*
free type *MI_Subm_or_Rec* ::= *subm*(*select_subm* :? *MI_Subm*) | *rec*(*select_red* :? *MI_Rec*)
end

logic CSPCASL
spec EP2 =
data EP2_DATA
channels *cCAI_Card* : *CAI_Card*; *cSI_Config* : *SI_Config*; *cSI_Init* : *SI_Init*;
cFE_FrontEnd : *FE_FrontEnd*; *cMI_Subm* : *MI_Subm*; *cBE_BackEnd* : *BE_BackEnd*;
cEI_ECR : *EI_ECR*; *cCOI_Config* : *COI_Config*; *cMI_Subm_or_Rec* : *MI_Subm_or_Rec*
process *Card* : *cCAI_Card* ;
ServiceCenter : *cSI_Config*, *cCOI_Config* ;
Acquirer : *cCOI_Config*, *cSI_Init*, *cFE_FrontEnd*, *cMI_Subm*, *cMI_Subm_or_Rec* ;
PosMgmtSystem : *cBE_BackEnd*, *cMI_Subm_or_Rec* ;
PointOfService : *cEI_ECR* ;
Terminal : *cCAI_Card*, *cSI_Config*, *cSI_Init*, *cFE_FrontEnd*, *cMI_Subm*,
cBE_BackEnd, *cEI_ECR* ;

Card = *RUN* (*cCAI_Card*)
ServiceCenter = *RUN* (*cSI_Config*) ||| *RUN* (*cCOI_Config*)
Acquirer = *RUN* (*cCOI_Config*) ||| *RUN* (*cSI_Init*) ||| *RUN* (*cFE_FrontEnd*)
||| *RUN* (*cMI_Subm*) ||| *RUN* (*cMI_Subm_or_Rec*)
PosMgmtSystem = *RUN* (*cBE_BackEnd*) ||| *RUN* (*cMI_Subm_or_Rec*)
PointOfService = *RUN* (*cEI_ECR*)
Terminal = *RUN* (*cCAI_Card*) ||| *RUN* (*cSI_Config*) ||| *RUN* (*cSI_Init*)
||| *RUN* (*cFE_FrontEnd*) ||| *RUN* (*cMI_Subm*)
||| *RUN* (*cBE_BackEnd*) ||| *RUN* (*cEI_ECR*)

ep2 : *cCAI_Card*, *cSI_Config*, *cSI_Init*, *cFE_FrontEnd*, *cMI_Subm*, *cBE_BackEnd*,
cEI_ECR, *cCOI_Config*, *cMI_Subm_or_Rec* ;
ep2 = *Terminal*
[| *cCAI_Card*, *cSI_Config*, *cSI_Init*, *cFE_FrontEnd*, *cMI_Subm*,
cBE_BackEnd, *cEI_ECR* |]
(*Card*
||| (*ServiceCenter*
[*cCOI_Config* || *cCOI_Config*, *cMI_Subm_or_Rec*]
Acquirer
[*cCOI_Config*, *cMI_Subm_or_Rec* || *cMI_Subm_or_Rec*]
PosMgmtSystem)
||| *PointOfService*)

Figure 4.2: Part of the EP2 specification in CSP-CASL

Chapter 5

The Hetereogeneous Context

Contents

CSP-CASL, as described in this thesis, is designed and implemented within a framework of *heterogeneous specification.* We start this chapter by introducing heterogeneous specification, in the setting of the CASL language family. Then we briefly describe the language HETCASL, and introduce the tool HETS. Finally, we describe informally how CSP-CASL fits into that world. As we will see, this has important implications for the design, formalisation, and implementation of the language — the subject of this work.

5.1 Heteterogeneous specification

> "Heterogeneous specification becomes more and more important because complex systems are often specified using multiple viewpoints, involving multiple formalisms. Moreover, a formal software development process may lead to a change of formalism during the development." [MML07a]

As its name suggests, heterogeneous specification allows specification using an heterogeneous mix of specification languages. (We ignore here *informal* heterogenous methods such as UML.) Historically, using multiple specification languages on a single project has been difficult, and poorly supported by the languages and tools in question; with a few exceptions, homogeneous specification — i.e. using a single specification language — has been the only real option. This has a number of obvious drawbacks:

- The language used may be insufficiently rich to describe what is desired.
- The language used may be too rich to use in conjunction with the tools desired (see the discussion of CASL sublanguages in section 2.3.1).
- More generally, only the tools directly supported by the language may be used; using combinations of tools is difficult or impossible.

One approach to this problem might be to attempt to create an all-encompassing specification language which solves all problems for all people. A recognition of the unlikelihood of success of such an effort leads to the conclusion that what is required is the ability to 'mix and match' various specification languages as appropriate: heteregeneous specification.

As noted in chapter 2, CASL is at the centre of a family of specification languages of varying capabilities; in such an environment, it is natural to wish to combine specifications written in those languages heterogeneously. Recent work enables this and even, in fact, "integration of logics that are completely different from the CASL logic", by formalising mechanisms for relating specifications written in a mix of languages together. For a full treatment, see [Mos03, Mos05].

The fundamental ideas, whose details are beyond the scope of this work, are that a specification language has underlying it a *logic* (CASL's, for example, is $SubPCFOL^{=}$, see section 2.3.1), that logics may be formalised using the category-theoretical notion of an *institution* [GB92], and that *institution morphisms* and *comorphisms* formalise translations between logics and thus between specification languages. This leads to the idea of a *logic graph*, describing possible translations between languages according to the morphisms in place; for example, figure 2.1 on page 8, displaying part of the CASL family of languages, is in fact a logic graph taken from [Mos04a].

5.2 HETCASL

Thus, while a CASL structured specification may consist of multiple (homogenous) specifications, a heterogeneous specification over the logic graph of CASL-family languages is a structured specification consisting not only of multiple specifications, but of specifications written in multiple logics. The language HETCASL [Mos04a] implements this notion in the CASL family context, with tool support provided by HETS (section 5.3).

It turns out that CASL's structuring constructs (see section 2.2.2) are independent of the logic used in the basic specifications being structured. HETS thus extends CASL's structuring mechanism, adding in particular, support for declaring the logic to use in a given context, and mechanisms for declaring relationships between logics (e.g. reductions, translations, renamings). A key point here is that HETCASL extends CASL *only* at the structuring level: basic specifications written in CASL (or any other language in its family) are essentially independent of the HETCASL structuring mechanisms (though as we shall see, at least for CSP-CASL, just operating within the HETCASL context has language design implications).

We now ignore (as out of the scope of this work) HETCASL's inter-logic features, and concentrate on the features of importance to CSP-CASL. In particular, we have the following syntactic extensions at the structuring level (examples follow, section 5.4):

- *Logic qualification* — this addition allows the declaration that a particular logic is to be used in a given context. A logic qualification is written:

  logic *L SP*

  where *L* denotes the logic to use and *SP* is a specification, whose *local environment* (see below) is the empty environment for that logic.

- *Data specification* — this addition pertains specifically to CASL extensions implementing 'process logics' (e.g. and in particular CSP-CASL). A data specification is written:

data SP_1 SP_2

This allows for the declaration of a 'data part' in SP_1, interpreted as a CASL basic or structured specification, whose signature is then *coerced* into the process logic, the result of which coercion is used as the *local environment* for SP_2, itself written in the language of the process language. See section 10.2.1 for further discussion of this mechanism and the notion of 'local environment' in the CSP-CASL context.

5.3 HETS, the Heterogeneous Toolset

HETS [MML07a, MML07b] provides "parsing, static analysis and proof management for heterogeneous multi-logic specifications by combining various tools for individual specification languages". Parsing and static analysis are the areas most relevant to this thesis, and are considered at length elsewhere; in order to provide a rounded context however, we shall briefly consider HETS' *proof management* capabilities.

At time of writing, HETS supports (at least one of parsing, static analysis and full logic implementation for) CASL [BM04, CoF04c], COCASL [MRRS03, MRS03], MODALCASL [Mos04b], HASCASL [SM02], HASKELL [SM02], CSP-CASL, OWL-DL (Web Ontology Language, OWL), and CASL_DL (a strongly-typed OWL-DL variant). Having written specifications involving one or more of these languages, we might reasonably want to say something more interesting than just 'the specification is well formed'[1]. For example:

- 'specification A is a refinement of specification B';
- 'the system described in specification A cannot deadlock';

and more generally:

- 'theorem X is a consequence of specifications $SP_1, \ldots, SP_n$'.

Proving things in HETS is achieved via *heterogeneous development graphs*. Given a structured (HETCASL) specification, such a graph encodes and represents the structure of the specification and any *open proof obligations* which have been automatically generated by the tool or explicitly specified by the user as part of the input. HETS then uses a *proof calculus* to decompose open obligations to *local proof goals* which can be discharged using an appropriate theorem prover. At present, HETS supports SoftFOL (automatic, for first-order logic) and Isabelle (assisted, for higher order logic).

HETS is currently used via a GUI, in which a specification's development graph can be visualised and manipulated, and through which the user may interact with the theorem provers. A fully-featured command-line interface, allowing greater automation of this process, is currently being implemented.

It is interesting to note that HETS is generic in terms of the logic graph used. It has been developed as a tool for heterogeneous specification over the CASL family so, naturally, the CASL family logic graph and logics are those which have been implemented so far. However, the parts of HETS implementing heterogeneous specification are independent of (and may be compiled separately to) those actually implementing the CASL logic graph and the individual logics. It would thus be possible to use HETS as a framework for heterogeneous specification

[1] This thesis is predicated, of course, on the notion that that is an interesting question of itself.

logic CSPCASL
spec TCS1 =
data sorts S, T
ops $c : S; d : T$
process $tcs1 : S, T$;
$tcs1 = c \rightarrow SKIP \,||\, d \rightarrow SKIP$
end

(a) CSP-CASL logic only

logic CASL
spec D =
sorts S, T
ops $c : S; d : T$
end

logic CSPCASL
spec TCS1 =
data D
process $tcs1 : S, T$;
$tcs1 = c \rightarrow SKIP \,||\, d \rightarrow SKIP$
end

(b) Using named process D in CASL logic

Figure 5.1: Two versions of `tcs1.cspcasl` in HETCASL

on a completely different set of languages, provided the appropriate logics and logic graph are implemented.

HETS is implemented in Haskell (see chapter 7). We describe HETS' internal structure and aspects of its implementation in chapter 12.

5.4 CSP-CASL in the heterogeneous setting

In order to fully integrate CSP-CASL into HETS for heterogeneous specification, it is necessary to formulate its underlying logic as an institution; discussion of this is beyond the scope of this work — see [Rog06, MR07, MR08]. There are, however, some interesting issues relating to syntax and semantics, which we now explore with the support of an illustrative example.

In section 4.2.1 we considered the example CSP-CASL specification `tcs1.cspcasl` from [Rog06]. In figure 5.1 we see two versions of the same example, in a HETCASL context. These examples serve to illustrate and introduce the key points regarding CSP-CASL's language design which arise as a result of interoperation with HETCASL.

The most important thing to note in figure 5.1(a) is the use of the data specification keyword **data**, described in section 5.2. As noted in chapter 4, this keyword has historically always been part of CSP-CASL, and its inclusion in this form in HETCASL is of great benefit for the integration of CSP-CASL into the CASL family: what we must now formalise is the *other part* of the specification, i.e. the SP_2 in '**data** SP_1 SP_2'. This has an important consequence, explored further in chapter 8 and subsequent chapters: a CSP-CASL basic specification then consists only of the *process part*. Syntactically, semantically and programmatically, the data part is essentially already taken care of:

- CASL's syntax is defined in [CoF04b], and HETCASL's in [Mos04a]. We need only define the syntax of the process part — though this will refer to the CASL syntax where we reuse CASL items such as `FORMULA` and `TERM`.
- The CASL static semantics is defined in [BCH+04]. We need only define the static semantics of the process part — though this will interact with the CASL static semantics,

both in terms of the 'input' we receive from the data part, and the semantics of items we reuse.

- Parsing and static analysis for CASL and HETCASL is already implemented in HETS. We need only implement them for the process part — though we will call CASL parsers and static analysers for items we reuse, and must ensure our implementation fits into the wider HETS machinery properly.

The other point to note from figure 5.1(a) is that it is identical to the version seen in section 4.2.1 *except* for the addition of the logic qualification '**logic** CSPCASL'. In fact this could (in this case at least) be omitted: by default HETS uses the CASL logic as its initial logic, but this may be overridden with a command-line option and thus at least for specifications written using only a single logic, no logic qualification is required in the input file, provided it is supplied on the command-line. However, truly heterogeneous specifications *require* logic qualifications, since only the initial logic may be overridden in this manner – so it is arguably a good idea to be explicit by default.

Finally, consider figure 5.1(b). This contains (semantically) exactly the same specification; however, instead of embedding the text of the data part in the CSP-CASL specification, we use HETCASL's *structuring* mechanisms in order to separately specify our data and process parts. We start by defining a basic CASL specification describing our data; this is then reused in the CSP-CASL specification, by referring to the named specification D in its **data** part. Such separation is clearly a desirable capability; it is worth noting that the data part could be defined in a completely different source file, and could even come from an off-site library.

Chapter 6

Natural Semantics

Contents

6.1 Introduction

Natural Semantics [Kah87] is a framework for writing operational semantics. Originally targetting the operational semantics of programming languages (notably Standard ML in [MTH90]), it has since been used to specify semantics of specification languages, in particular the static and model semantics of CASL [BCH+04]. It is thus of interest to us as the canonical choice for presenting the static semantics of CSP-CASL, for which see chapter 10. (CSP-CASL's model semantics is outside the scope of this document, but see, e.g., [Rog06].)

The aim of this chapter is to introduce Natural Semantics in order to support chapter 10's presentation of CSP-CASL's static semantics. In the rest of section 6.1 we describe the formalism in general, then focus on aspects relating to static semantics, since this is our main area of interest. Here we set the scene for the examples in the following two sections: in section 6.2 we look at the static semantics of Mini-ML as presented in [Kah87]; in section 6.3, we consider an example from the static semantics of CASL, namely local libraries.

Kahn [Kah87] introduces and names the framework, noting that it builds on earlier work of Plotkin [Plo81]. The stated aim of natural semantics in that paper is to "...present several aspects of programming language semantics in a unified manner...". (Note the restriction here to *programming* languages — application to the semantics of specification languages such as CASL came later.) These 'several aspects' are:

- static — well-typedness of expressions, declarations/scope;
- dynamic — execution behaviour;
- translational — translation of expressions between languages, e.g. interpretation.

The approach is proof-theoretic: generation of new facts (proof trees) from existing ones. The formalism's visual style is heavily influenced by Gentzen's Natural Deduction [vD04], hence

the name.

It should be noted that Natural Semantics is a *framework* or *style* rather than a 'closed' formalism and that, for example, a rule from CASL's static semantics looks rather different to a rule from Mini-ML's dynamic semantics. The framework provides basic mechanisms for writing semantic definitions, but the exact contents of such definitions will vary greatly depending on:

- The kind of semantics we are defining — e.g., static semantics is concerned with *types*, whereas dynamic semantics is concerned with *values* and *states* (both are concerned with *expressions*, however).
- The kind of language whose semantics we are defining — e.g. Mini-ML and CASL are *very* different languages: their elements, syntaxes and – at heart – semantic contents are just different things.

Thus, rather than try to provide a framework containing all possibilities, Natural Semantics instead requires that we accompany our semantic rules with definitions of the actual semantic objects under discussion, leaving the style of these definitions open. Natural Semantics then provides us with a mechanism for formally describing relationships between these objects, allowing proofs to be built. As we will see, it is generally the context-specific machinery which complicates a definition in the Natural Semantics style; the formalism itself is quite simple.

6.1.1 The formalism

Here we provide a top-down overview of the elements of the Natural Semantics formalism. Having done this, we'll look briefly at how static, dynamic, and translational semantics are defined, concentrating on static semantics as our main area of interest.

A **semantic definition** is a collection of rules. A **rule**, like a rule in natural deduction, has a numerator and a denominator. The numerator is a collection of formulae: the rule's **premises**; the denominator is just one formula: the rule's **conclusion**. A rule is written:

$$\frac{premise_1 \quad \cdots \quad premise_n}{conclusion}$$

A rule's meaning is that from proof trees yielding *all* of the premises we obtain a new tree yielding the conclusion. **Formulae** (the premises and conclusions) consist of sequents and conditions. **Sequents** have the basic form:

$$antecendent \vdash consequent$$

the content and interpretation of which vary with context, though the consequent is always a predicate, whose first argument is generally the subject of the sequent. Sequents are, largely, where the 'action' is. A **condition** is a boolean predicate which restricts the applicability of the rule (sometimes written to the right of the bar separating the premises from the conclusion — though not in the case of CASL). An **axiom** is a rule with no sequent in its premises. For examples of axioms and conditions, see rules (1) and (2) in section 6.2.2.

6.1.2 Static semantics and static analysis

Having introduced the formalism in general, we now consider its application to static semantics. Consider the sequent:

$$\rho \vdash E : r$$

This is a typical style for sequents in a static semantics definition, and can be read as:

"The expression E has type r in the context of environment ρ."

Here, then, 'expression' and 'type' have their normal meanings. The **environment** is an artifact of the static semantics: it is a list of *bindings* of which we are already aware, associating expressions with types.

Given the above sequent, two questions we might reasonably ask are:

1. Given ρ, can E be assigned the type r such that $\rho \vdash E : r$? (type checking)
2. Given E and r, does there exist an environment ρ such that $\rho \vdash E : r$? (typability)

In the CSP-CASL context, we are most interested in questions of the first kind, though our semantic objects are not restricted to types of expressions, and our semantic rules build a variety of semantic objects. Nonetheless, the basic operation of constructing semantic objects and checking that they relate 'sanely' to one another is the central question in our task of static analysis: specifications which violate the rules are ill-formed, and must be rejected with an appropriate and hopefully useful error message.

The sequent form given above is not the only possibity. Commonly, we might have:

- $\rho_1 \vdash DECL : \rho_2$ a declaration, expanding *environment* ρ_1 to ρ_2
- $\rho \vdash E : r$ expression E has type r in environment ρ
- $\rho \vdash E$ expression E is well-typed in environment ρ

A static semantic definition then consists of a collection of rules, some of which concern building environments (e.g. rules containing sequents of the first type above), and the rest of which concern determining/checking the types of expressions within environments (e.g. rules containing sequents of the second and third types above).

We will see and discuss examples of such rules later in this chapter, but here is a simple example to whet the appetite:

$$\frac{\rho \vdash E_1 : r_1 \qquad \rho \vdash E_1 : r_2}{\rho \vdash (E_1, E_2) : r_1 \times r_2}$$

We can read this as "the pair (E_1, E_2) has the type $r_1 \times r_2$ in environment ρ provided E_1 and E_2 have types r_1 and r_2 respectively in environment ρ". The proof of the premises involves further rules, leading eventually to 'base' rules dealing with constants, literals, and declarations. Similarly, the conclusion might be used when checking some other expression involving (E_1, E_2), in the context of environment ρ.

6.2 Semantics of Mini-ML

Our first example of Natural Semantics in action comes from [Kah87], in which Kahn exemplifies static, dynamic, and translational semantics using a small functional language, Mini-ML. Since we are primarily concerned with static semantics, we concentrate on this aspect.

6.2.1 Mini-ML

Mini-ML is "a simple typed λ-calculus with constants, products, conditionals and recursive function definitions" [Kah87]. Before we can discuss its static semantics, we need to introduce the language, though we omit the details. As a simple example of its concrete syntax, here is the mandatory *factorial*:

letrec *factorial*= $\lambda\, x.$ **if** $x = 0$ **then** 1 **else** $x*$ *factorial*$(x - 1)$
in *factorial* 4

The abstract syntax of Mini-ML is fairly simple. The sorts are expressions, identifiers and patterns, which occur in lambda abstractions and let-expressions (in the example above, the first occurences of *factorial* and x are patterns; subsequent occurrences are identifiers).

In order to discuss Mini-ML's static semantics, we must first consider its *type language*. In Mini-ML, a type r is one of the following:

- a basic type (*int* or *bool*);
- a type variable α;
- a functional type $r \rightarrow r'$ (with r, r' types);
- a product type $r \times r'$ (with r, r' types).

We noted earlier that an environment consists of a list of bindings, which associate expressions with types. In Mini-ML, a binding associates an expression with one of the following:

- a type r;
- a type-scheme $\forall\, \alpha.\sigma$ where α is a type variable and σ is a type-scheme.

Type variables and quantified type schemes allow us to define types of polymorphic functions, e.g. $\forall\, \alpha.\alpha \rightarrow \alpha$, the type of the polymorphic identity function.

6.2.2 Static semantics

The full static semantics is beyond the scope of this chapter. Instead, we will examine a few example rules which give us a good idea of the flavour and meaning of the semantic definition. Our first two example rules deal with declarations:

$$\vdash \text{ident x}, r : \text{ident x: } r \tag{1}$$

$$\frac{\vdash P_1, r_1 : \rho_1 \qquad \vdash P_2, r_2 : \rho_2}{\vdash (P_1, P_2), r_1 \times r_2 : \rho_1 + \rho_2} \quad (\rho_1 \cap \rho_2 = \emptyset) \tag{2}$$

Here $\vdash P, r : \rho$ means "declaring P with type r creates the environment ρ". Rule (1) allows for the declarations of identifiers; rule (2) for the declaration of patterns, which are essentially binary trees of identifiers. Here $+$ is a 'union-like' operation on environments (with some extra rules, omitted here).

Further examples then concern the types of expressions. Here we see the more familiar judgement $\rho \vdash E : r$. Now we are no longer just building environments; we are checking types of expressions against them.

$$\frac{\rho \vdash E_1 : \mathit{bool} \qquad \rho \vdash E_2 : r \qquad \rho \vdash E_3 : r}{\rho \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : r} \tag{3}$$

Rule (3) is fairly simple, and shows how to determine the type of a conditional expression: it is necessarily the type of either (both) of the branch expressions; the condition expression itself must be boolean, of course. Note that it explicitly introduces the constraint that E_2 and E_3 must have the same type.

$$\frac{\vdash P, r' : \rho' \qquad \rho + \rho' \vdash E : r}{\rho \vdash \lambda P.E : r' \to r} \tag{4}$$

Rule (4) determines the type of a λ-abstraction: a function type, of course. Here P has type r' in ρ', and E has type r in the concatenation of ρ' and ρ, the environment in which the λ-abstraction occurs. The point here is that P need not exist in the environment ρ in which we consider the $\lambda\, P.E$. In essense, we add it to that environment when checking E.

$$\frac{\rho \vdash E_1 : r' \to r \qquad \rho \vdash E_2 : r'}{\rho \vdash E_1\ E_2 : r} \tag{5}$$

Rule (5) determines the type of a function application. It checks that the function is applied to an expression of the correct type (r').

At this point, hopefully the meaning of the rule given at the end of section 6.1.2 is clear.

6.3 Static semantics of CASL local libraries

As a domain-specific example, we now consider the static semantics of CASL local libraries, as defined in [BCH+04, §III:6.2]. We have chosen this example because it is small enough to discuss fairly briefly, while adequately illustrating the key features of CASL's static semantic rules, and in particular some interesting aspects (namely *linear visibility* and *qualified rules*) which also appear in the CSP-CASL semantics.

The rules in this context have a slightly different structure to those for Mini-ML considered above. In particular, whereas the Mini-ML rules are concerned with determining the type of an expression, the CASL static semantics are somewhat more general in character: a given rule will 'yield' one or more of various kinds of semantic objects — not just a type. In the case of local libraries, the main yielded semantic objects are *static global environments*, described below. However, one of the rules yields not only a static global environment but also a *library name*. In general, CASL static semantic rules yield whatever semantic objects are required in the given context; the CSP-CASL rules presented in chapter 10 are similarly varied. They are thus richer — and correspondingly harder to decode — than the Mini-ML rules; none of this is particularly surprising given CASL's (and CSP-CASL's) greater scope and complexity.

Removal of aspects relating only to distributed libraries

Note that the rules presented in this section are in fact *modified* versions of those found in [BCH+04, §III:6.2]. CASL supports not only local libraries but also *distributed* ones, and the rules for distributed libraries involve semantic objects called *static universal environments* and *global directories*. As the rules for distributed libraries extend those for local libraries, the latter must (in their full form) 'pass through' these extra semantic objects. However, the objects are not actually used in the local library rules; as such, as we do not consider distributed libraries here, we have chosen to simplify and clarify our discussion of the rules for local libraries by omitting those parts of no immediate relevance.

6.3.1 Semantic objects

As noted in section 6.1, it is necessary when writing rules using Natural Semantics to define the domain-specific objects to which the rules refer. In the case of the CASL local library rules, we have the following semantic objects:

- *Static global environments* $\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)$, which consist of finite functions from names to static denotations of generic specifications, views, architectural specifications and unit specifications.

  The details of $\mathcal{G}_s$, $\mathcal{V}_s$, $\mathcal{A}_s$, and $\mathcal{T}_s$ are irrelevant to this discussion, and thus omitted. However, the main point is that the static global environment is a container for the semantic objects corresponding to the contents of the library. As such, this is *the* key semantic object related to local libraries, and the rules presented below are essentially concerned with building these.

- *Library names* and *library identifiers.* A library identifier is either a *URL* pointing directly to the library, or a *path* facilitating indirect lookup via a global directory (see note regarding distributed libraries, above). A library name is then a library identifier and a version number.

$$\begin{aligned} LI &\in LibId = Url \uplus Path \\ LN &\in LibName = LibId \times Version \end{aligned}$$

  The internal structure of URLs, paths, and version numbers is irrelevant to this discussion, and thus omitted.

6.3.2 The rules

The presentation of CASL's static semantics in [BCH+04, §III:6.2] adheres to a particular format throughout. The first part of any collection of rules is the abstract syntax excerpt for the syntactic elements in the rules' conclusions. In this case, we are interested in `LIB-DEFN` and `LIB-ITEM`:

```
LIB-DEFN ::= lib-defn LIB-NAME LIB-ITEM*
LIB-ITEM ::= SPEC-DEFN | VIEW-DEFN | ARCH-SPEC-DEFN | UNIT-SPEC-DEFN
```

This is followed by a highlighted summary of the key points of the rules which follow. In the case of CASL local libraries, it reads:

> "A library definition `LIB-DEFN` provides a collection of specification (and perhaps also view) definitions. It is well-formed only when the defined names are distinct, and not referenced until (strictly) after their definitions. The global environment for each definition is that determined by the preceding definitions. Thus a library in CASL provides linear visibility, and mutual or cyclic chains of references are not allowed."

This is followed by some optional discussion, and then the rules themselves. Here we have a header summarising the format of the rule or rules following, and the actual rule/rules — often interspersed with and followed by further discussion/clarification of the rule's contents and meaning.

6.3.2.1 LIB-DEFN

The first rule for local libraries is as follows:

$$\boxed{\vdash \texttt{LIB-DEFN} \rhd (LN, \Gamma_s)}$$

$$\frac{\vdash \texttt{LIB-NAME} \rhd LN \qquad \emptyset \vdash \texttt{LIB-ITEM*} \rhd \Gamma_s}{\vdash \texttt{lib-defn LIB-NAME LIB-ITEM*} \rhd (LN, \Gamma_s)}$$

The header summarises the rule:

1. It is the rule for the syntactic element LIB-DEFN.
2. The required context is empty (there is nothing to the left of the $\vdash$ symbol).
3. The semantic object yielded is a pair consisting of a library name LN and a static global environment Γ_s.

Thus, given a library definition, we may compute a library name and static global environment for that library definition.

Now consider the rule itself; there are two premises and (of course) one conclusion. The conclusion is essentially identical to the header, except that the syntactic element LIB-DEFN has been expanded according to one rule from the abstract syntax. This introduces the syntactic elements which are referred to in the premises. Where the grammar contains multiple productions for a particular syntactic element, there will be a set of rules in the semantics — see the discussion of 'qualified rules', below.

The premise $\vdash$ LIB-NAME $\rhd LN$ refers to the rule for LIB-NAME; this rule is not shown here, but as it has no context we may reliably infer that it simply turns a syntactic name (whatever is contained in LIB-NAME) into a semantic one (i.e. LN). Such premises are typical at the level of names. Note that the library name LN yielded by this premise is then the first part of the pair yielded by the LIB-DEFN rule's conclusion.

The premise $\emptyset \vdash$ LIB-ITEM* $\rhd \Gamma_s$ refers to the rule for LIB-ITEM*, considered below. To this rule is passed an empty set which, as we shall see, forms the basis of the yielded static global environment Γ_s. That Γ_s is then the second part of the pair yielded by the LIB-DEFN rule's conclusion.

6.3.2.2 LIB-ITEM*

The next rule defines the static semantics of a sequence of LIB-ITEMs. It is probably the most interesting rule of the three presented here, as it encodes linear visibility.

$$\boxed{\Gamma_s \vdash \texttt{LIB-ITEM*} \rhd \Gamma'_s}$$

Thus: the rule deals with a sequence of LIB-ITEMs; the context consists of a static global environment; the rule then yields a *different* static global environment. In particular, as we shall see, the 'input' static global environment Γ_s is expanded upon by each LIB-ITEM in the sequence to produce the final one, Γ'_s.

$$\frac{\begin{array}{c}(\Gamma_s)_0 \vdash \texttt{LIB-ITEM}_1 \rhd (\Gamma_s)_1 \\ \cdots \\ (\Gamma_s)_{n-1} \vdash \texttt{LIB-ITEM}_n \rhd (\Gamma_s)_n\end{array}}{(\Gamma_s)_0 \vdash \texttt{LIB-ITEM}_1 \ \ldots \ \texttt{LIB-ITEM}_n \rhd (\Gamma_s)_n}$$

In the rule's conclusion, LIB-ITEM* is expanded to an indexed list of LIB-ITEMs, each of which has a corresponding sequent in the rule's premises. The LIB-ITEM rule is presented below, but the particular thing to note here is the chain of static global environments being passed along the rules: the rule for $\texttt{LIB-ITEM}_1$ receives the first, $(\Gamma_s)_0$ in its context, and produces $(\Gamma_s)_1$, which is passed to $\texttt{LIB-ITEM}_2$, which produces $(\Gamma_s)_2$, etc., until finally $\texttt{LIB-ITEM}_n$ produces the final static global environment $(\Gamma_s)_n$ which is in turn yielded by the rule's conclusion.

This arrangement implements *linear visibility*, whereby later library items may refer to items defined in earlier ones, *but not the other way round.* We also find linear visibility in the CSP-CASL static semantics: for example, process declarations and process equations may be interspersed, but a process equation may only refer to a process whose declaration has already been seen (see section 10.3.3). *Nonlinear* visibility requires a somewhat more complicated arrangement of rules; for a CASL example see datatype declarations [BCH+04, §III:2.3.4].

Finally, there is a small stylistic point to make here. The rule shows how to expand a static global environment Γ_s with a sequence of library items to produce a new static global environment Γ'_s. When we 'call' it from the LIB-DEFN rule we pass an empty set to Γ_s, so that the net effect is to produce the static global environment for the entire library *and only the library.* Now, note that an alternative and equivalent rule would receive *no* static global environment as input, and simply build up from the empty set internally by passing $\emptyset$ in the $\texttt{LIB-ITEM}_1$ premise:

$$\frac{\begin{array}{c}\emptyset \vdash \texttt{LIB-ITEM}_1 \rhd (\Gamma_s)_1 \\ \cdots \\ (\Gamma_s)_{n-1} \vdash \texttt{LIB-ITEM}_n \rhd (\Gamma_s)_n\end{array}}{\vdash \texttt{LIB-ITEM}_1 \ \ldots \ \texttt{LIB-ITEM}_n \rhd (\Gamma_s)_n}$$

There seems to be no technical reason for not using this style: no other rule in the CASL semantics refers to this LIB-ITEM* rule, so no other rule wants to pass in a non-empty starting environment. We speculate, then, that the rule has been written in this form because CASL has been designed to be extended, and some extension might in fact wish to treat libraries differently. In the CSP-CASL rules we tend towards the other, more minimal, rule form where possible, favouring compactness over generality — see for example the CHAN-DECLS rule in section 10.3.2.

6.3.2.3 LIB-ITEM

The final rule deals with an individual library item:

$$\boxed{\Gamma_s \vdash \texttt{LIB-ITEM} \rhd \Gamma'_s}$$

Thus, the LIB-ITEM rule takes a static global environment as context, and yields a different one:

$$\frac{\Gamma_s \vdash \texttt{SPEC-DEFN} \rhd \Gamma'_s}{\Gamma_s \vdash \texttt{SPEC-DEFN qua LIB-ITEM} \rhd \Gamma'_s}$$

The main thing to note here is that this is a *qualified rule*. Qualified rules correspond to elements of the abstract syntax containing disjoint unions of syntactic categories. Consider again the abstract syntax for LIB-ITEM:

```
LIB-ITEM ::= SPEC-DEFN | VIEW-DEFN | ARCH-SPEC-DEFN | UNIT-SPEC-DEFN
```

Thus, a LIB-ITEM devolves to one of four possibilities, and we need one rule for each of these cases. These, then, are qualified rules, and are indicated by the presence of the text 'qua' in the rule's conclusion. There is only ever one sequent in the premises of a qualified rule: a reference to the rule for the syntactic category to which we are devolving — in this case, SPEC-DEFN. The rule for SPEC-DEFN is not shown: it is part of the static semantics of structured specifications [BCH+04, III:4]. Recall that semantically, a library is nothing more than a container for semantic denotations of entire specifications; detailed knowledge of those semantic objects is not required in order to understand the rules for local libraries. In other words, CASL's static semantics is structured in a helpfully modular fashion.

Similar qualified rules exist for devolving LIB-ITEM to VIEW-DEFN, ARCH-SPEC-DEFN, and UNIT-SPEC-DEFN; however, it is almost always the case that where we have qualified rules covering the disjoint union of syntactic categories, only the first rule is shown — the rest are elided. In such cases it is implicit that the elided rules are entirely similar to the shown one, differing only in the syntactic category to which the rule refers. As such, we can easily infer the three missing rules in this example.

Chapter 7

Haskell and Parsec

Contents

In this chapter we introduce the programming language *Haskell*, in which our tool is implemented, and the library *Parsec*, which is the key technology in the implementation of the parser part of the tool. Note that the use of both Haskell and Parsec in our implementation is a constraint introduced by the context of our work, i.e. extension of the HETS toolset: HETS is written in Haskell, and uses Parsec throughout for parsing — see chapters 5 and 12.

7.1 Haskell

Haskell [BW88, Pey03, HHJW07] is a polymorphically-typed, lazy purely functional programming language, based on the λ-calculus. In this section we introduce its key features with some small examples, in particular concentrating on aspects of relevance to our implementation.

7.1.1 Main features

Haskell is functional in the sense introduced by Backus in [Bac78]; a Haskell program is a collection of **functions**, and is **pure** in that a function has *no side effects*: it takes some input and produces some output, and that is *all it does*. Thus, Haskell programs have no implicit state, as modification of state violates purity. Indeed, Haskell functions are *referentially transparent*: every application of a given function with a certain input produces the same output, just like a function in the mathematical sense; as such, a function application can always be replaced by the value thus produced.

Our first example of a Haskell function is the function which doubles integers:

```
double :: Int -> Int
double x = x + x
```

The first line is the function's *type signature*, and states that this is a function from `Int` to `Int`; the second line defines the function's behaviour, equationally, in an obvious manner. It is

almost always possible to omit the type signature, and have the Haskell compiler infer it from the function definition using Hindley-Milner type inference [Mil78].

A function call, or *application* is written without parentheses; e.g. `double 5`, which according to the above definition of `double`, has the value 10. *Anonymous functions* may be written in the form of λ-abstractions, as `\x -> f x` (for the λ-term $\lambda x.f(x)$), binding a single input variable to the name `x` within the scope of `f`.

Haskell is **strongly typed** (all values have a particular type), and **statically typed** (the type of a value never changes). Furthermore, and critically, Haskell is **higher-order**: functions are 'first class citizens', have types (as demonstrated above), and may be passed as parameters to, and returned from, other functions. For example, consider:

```
twice :: (Int -> Int) -> (Int -> Int)
twice fn x = fn (fn x)
```

This higher-order function takes and returns a function from `Int` to `Int` — the function it returns has the effect of applying the input function twice. We can go further: Haskell is **polymorphic**; rather than restrict `twice` to functions from `Int` to `Int`, we can generalise it as follows:

```
twice :: (a -> a) -> (a -> a)
twice fn x = fn (fn x)
```

Here `a` is a *type variable* referring to 'some type'. Supposing we also define:

```
repStr :: String -> String
repStr a = a ++ a
```

which repeats a string, then we have:

```
(twice double) 5 = 20
(twice repStr) "hi " = "hi hi hi hi"
```

Haskell has a number of other built-in types, including `Int`s, `String`s, `Char`s, lists (in fact, a `String` is just a list of `Char`s, written `[Char]`), etc. Naturally, we may define our own types based on these — primarily via **algebraic data types**. These are aggregated types, whose values consist of data of various types wrapped in a *constructor*; an algebraic datatype may have multiple constructors, and their definitions may be recursive. For example, the following defines a polymorphic datatype of binary trees:

```
data Tree a = Leaf a
            | Tree (Tree a) (Tree a)
```

The datatype is called `Tree a` ('tree of as'), and has two constructors: `Leaf` wraps a value of type `a` in a `Tree a` value representing a leaf of the tree; `Tree` takes two values of type `Tree a` and wraps them in a `Tree a` value representing an inner node. n.b., we have a constructor with the same name as the type, and we have a recursive definition. Then the following:

```
nTree = Tree (Tree (Leaf 1) (Leaf 2)) (Tree (Leaf 3) (Leaf 4))
sTree = Tree (Leaf "andy") (Tree (Leaf "markus") (Leaf "basheera"))
```

defines two values: `nTree` of type `Tree Int`, and `sTree` of type `Tree String`, illustrated in figure 7.1.

Algebraic datatypes are used widely in our tool, in particular to represent abstract syntax trees produced by the parser (see section 12.2.1), and various semantic objects produced by the static semantics phase (see section 12.4).

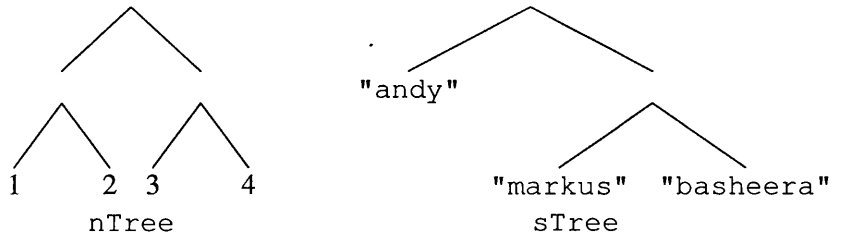


Figure 7.1: Visualisation of `nTree :: Tree Int` and `sTree :: Tree String`

When a value of an algebraic datatype is referenced, its members are typically unpacked using *pattern matching* on constructors. The following function walks a `Tree a` from left-to-right, depth first:

```
walk (Leaf x) = show x
walk (Tree left right) = (walk left) ++ " " ++ (walk right)
```

`show` converts a value to a `String` — see below. Then, `walk` takes a `Tree a` as input, and returns a string of the tree's leaves, e.g.:

```
walk nTree = "1 2 3 4"
```

The definition of `walk` consists of a case distinction over the datatype's constructors; if the `Tree a` value is a leaf node, then the `a` value is bound locally to the name `x`, used on the right side; similarly, the case for an inner node binds the child values to the names `left` and `right`. In general, pattern matching is used extensively in Haskell function definitions.

Type classes are a novel feature of Haskell, somewhat akin to interfaces in, say, Java, in that they define a *contract* regarding functions which must be defined for a particular type. For example, the type signature of `walk`, above, is:

```
walk :: Show a => Tree a -> String
```

The '`Show a =>`' part of this signature enforces a constraint, namely that this function can only be called on values of type `Tree a` where the type `a` is a member of the type class `Show`. This is a standard type class concerned with converting values to strings. In particular, it guarantees the existence of the function `show :: a -> String` used in the `Leaf a` case of `walk`.

Clearly, we should be able to state that our own data types are members of particular type classes. One convenient method is a *derived instance*, where we request a default implementation to be inferred by the compiler; e.g. we might write our `Tree a` declaration as:

```
data Tree a = Leaf a
            | Tree (Tree a) (Tree a)
  deriving (Show, Eq)
```

Then `Tree a` automatically 'inherits' implementations of `show` and == (to check for equality). However, this constrains `Tree a` values to types `a` which are instances of both `Show` and `Eq` (almost all built-in types are). Where such a derivation cannot be made or is unsuitable, we may explicitly declare a type to be a member of a type class, and provide implementations of the required functions; for example, this is necessary when we implement pretty-printing of CSP-CASL — see section 12.3.

A Haskell program is organised into **modules** providing namespaces — this is Haskell's key method for controlling scope of functions. Modules may import other modules, in which case the imported module's exported types and functions are in scope in the importing module. By default, all of the contents of a module are exported; modules may hide contents by providing an explicit export list. Conversely, an import may be restricted by listing the names to import, or may be *qualified* by some name, in which case any references to the imported items must be prefixed by the qualification name.

Haskell has a large standard library of modules providing types, type classes and functions for a wide range of standard and exotic purposes. In particular, there are modules for: datatypes such as sets, relations, lists, mappings, monads, functors, and various representations of primitive types such as integers and strings; common tasks such as filesystem access, networking, pseudorandom number generation, graphics, sound processing, and user interfaces; and facilities for distributing Haskell packages, and for interfacing with code written in other languages via the *Foreign Function Interface*. Furthermore, there is a large and ever-growing collection of third-party libraries[1] for an astonishing range of purposes.

7.1.2 Monads and monadic I/O

In section 7.1.1 we asserted that Haskell programs consist of functions which have no side effects. This purity is at the heart of Haskell and results in many distinctive and useful properties; however, in general, we sometimes *want* side effects and non-purity, e.g. for input and output. Reading from and writing to the console, for example, must clearly modify some state *somewhere* in the computer.

There are a number of ways in which this problem might be solved; Haskell uses **monads** to control side effects of this nature [Wad92, HHJW07]. Monads, a category-theoretical construct, neatly encapsulate the concept of order of execution, and in fact allow the Haskell programmer to define their own execution strategy for a given situation; in particular, in Haskell we have **monadic input/output** via the `IO` monad, as well as a plethora of other monads. Due to Haskell's purity, order of execution is in general unpredictable, particularly as Haskell is **lazy** and only evaluates an expression when its value is required; clearly this is also problematic for I/O. Using the `IO` monad, we represent and manipulate computations which have side effects and a strictly sequential order of execution; it is important to realise that in such functions, *we are not performing the I/O*, but are *computing functions which perform the I/O* and which are called at runtime as required according to lazy evaluation of I/O values. *All* Haskell functions, including ones operating on the `IO` monad, are pure — see the example below.

From the programmer's point of view, the key benefit of this approach is a clear separation between code which can give rise to side-effects, and pure code where this is not the case: a function's type signature clearly indicates which 'world' we are in, and the type system rigourously enforces this separation. Most Haskell programs consist of an outer 'interface shell' which performs I/O, often written in a more imperative style, and a pure functional 'core' with no side effects. Indeed, the type signature of a Haskell program's `main` function is `main :: IO ()` — that is, `main` takes no parameters, performs some I/O as a side effect, and returns nothing (here `()` represents the empty type, and also values of that type). Then we have, e.g., to read and write a line from/to the console:

getLine :: IO String

[1] `http://hackage.haskell.org/`

```
putStrLn :: String -> IO ()
```

A monad is any type which implements the `Monad` type class, `Monad m`, with polymorphic type constructor `m` (`IO` in the above), and four particular functions, such that they obey certain laws; the functions are:

```
(>>=) :: m a -> (a -> m b) -> m b
return :: a -> m a
(>>) :: m a -> m b -> m b
fail :: String -> m a
```

Of these, `>>` and `fail` have default implementations in terms of the first two functions; these defaults usually suffice but may be overridden if necessary. Thus, it is minimally necessary to define `(>>=)` (pronounced 'bind') and `return`. Together, these functions control the sequencing of computations occuring 'in the monad'. `return` specifies how a value of type `a` is 'placed inside' the monadic value `m a`, and `(>>=)` and `(>>)` specify how to chain computations performed within the monad.

For the implementation details of `(>>=)`, etc., see [Pey03]; the key point to note is that the user of the monad simply calls these functions to 'chain together' operations to be performed within the monad, however is appropriate for the monad in use: the actual control of execution order is thus abstracted away, into these functions.

This turns out to be such a central idea (and in particular, I/O is such a critical aspect of any programming language) that Haskell blesses monads (which are, after all, just another type class in the library) with a particular syntactic sugar called 'do notation'. This can be seen as defining an imperative-looking sublanguage where chains of monadic function calls may be created without explicitly chaining them using `(>>=)`, `(>>)`, etc. To give a trivial example, here is a complete Haskell program for asking a user their name and then printing it back to them:

```
module GetName where

main :: IO ()
main = do putStrLn "What is your name?"
          name <- getLine
          putStrLn ("Hello, " ++ name)
          return ()
```

The 'do-block' in `main` chains together four calls, binding the result of the second to the name `name`, in scope for the remainder of the block. We emphasise again that this imperative-appearing notation is just syntactic sugaring for pure Haskell. In this case, the body of `main` is desugared to the following:

```
main = (putStrLn "What is your name?") >> (getLine) >>=
          (\name -> (putStrLn ("Hello, " ++ name)) >> (return ()))
```

Note how the `(>>=)` operater passes the result of `getLine` to a λ-abstraction defining the scope of `name`.

Now, while monads were introduced to address the I/O issue, they are not restricted to I/O, nor to encoding strictly sequential execution. For example, the `Maybe` monad encapsulates a simple notion of exception handling, where we have a chain of dependent computations which can be interrupted at any point by the failure of one of them. Of particular interest

to us, however, are mechanisms for simulating *statefulness* (see section 12.4), and Parsec's `GenParser` monad, which encapsulates the execution model of a recursive descent parser with arbitrary *backtracking*, which we now examine in context.

7.2 Parsec — monadic combinator parsing in Haskell

7.2.1 Recursive Descent Parsing

Parsec, the subject of this section, is a library for writing *recursive descent parsers*; we should briefly outline what this means.

Preliminary definitions: An *alphabet* is a finite set of symbols, typically atomic characters; a *string* is a finite sequence of symbols taken from an alphabet; a *formal language* is a set of strings over some alphabet; a *grammar* is a description of a formal language, manifested as a set of rules concerning how strings in the language may be generated; in that context, a *terminal* is an entity in the grammar corresponding to a symbol in the alphabet, whereas a *non-terminal* is a different kind of entity, not corresponding to a symbol in the alphabet; finally, a *context-free grammar* is a grammar having a particular structure. [Sud05]

For example, here is a context-free grammar:

$$S \rightarrow A$$
$$A \rightarrow T \mid A + T$$
$$T \rightarrow b \mid (A)$$

Here we have four terminals (b, $+$, (and)), three non-terminals (S, A, and T) and five *productions* showing how non-terminals may be transformed (the '|' symbol indicates choice, so e.g. there are two productions for A, namely $A \rightarrow T$, i.e. 'A may be replaced by T', and $A \rightarrow A + T$, i.e. 'A may be replaced by $A + T$'). This grammar defines a language, namely the set of strings of non-terminals derivable by repeated applications of the production rules starting, by convention, with a single instance of the non-terminal S.

For example, the follow demonstates a *derivation* of the string $(b) + b$ in this language:

$$S \rightarrow A \rightarrow A + T \rightarrow T + T \rightarrow (A) + T \rightarrow (A) + b \rightarrow (T) + b \rightarrow (b) + b$$

Now, "parsing is the process of determining how a string of terminals can be generated by a grammar" [ALSU06]. That is, given some input consisting of members of an alphabet, and a grammar defining a language we are interested in, we wish to determine if it is possible to generate that input from that grammar — and (critically) if so, *how*? This question is of central importance for processing texts written in formal languages, in particular programming languages and specification languages.

There are a number of ways to approach this problem, but a common view on them is that of traversing the *graph of a grammar*; this is a directed graph, where:

- leaf nodes are strings in the corresponding language;
- inner nodes are 'partially-derived' strings, containing some non-terminals;
- the root node is the grammar's start symbol;
- at every node, we have an arc for every possible production leading from that node (possibly labelling the arc with the derivation).

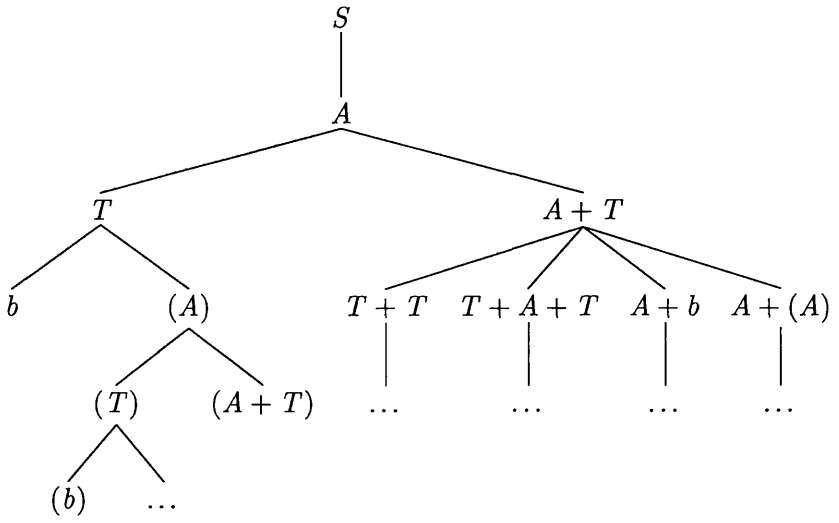


Figure 7.2: Part of the graph of a grammar

For any non-trivial grammar, such a graph will be infinite, of course. For example, the start of the graph of the grammar seen above is shown in figure 7.2.

Recursive descent parsing, then, is an approach to parsing characterised by exploration of this graph, top-down, one derivation at a time until a non-terminal is reached; the exploration is directed by the input string being parsed: in the above example, at the inner node labelled T, if the next symbol of input is '(' then we clearly take the right-hand branch and ignore the other. In general this approach requires backtracking, so that 'dead end' branches may be explored and disregarded: the maximum number of symbols one must read before possibly being forced to backtrack is the *lookahead* of the grammar. There is a danger of exploring an infinitely long branch; with finite input this can only occur without consuming all input, thus, if exploring the tree top-down and left-to-right, an infinite branch is the result of *left-recursion* in the grammar, where the leftmost symbol of a production is identical to the non-terminal acted on by that production[2]. Such grammars may be transformed mechanistically into non-left-recursive ones, however.

7.2.2 Parsec

Parsec [LM01] is a library for writing parsers in Haskell, based on earlier work by Wadler, Hutton, and others [Wad85, Hut92, Fok95, HM96]. The library provides a polymorphic type `GenParser tok st a`, representing a backtracking recursive descent parser with unbounded lookahead which parses tokens of type `tok`, with a user supplied state `st`, returning a value of type `a` on success. This may then be specialised, e.g. Parsec also provides `Parser a`, a type synonym for `GenParser Char () a`, for statelessly parsing a stream of characters to a value of type `a`. Building on this, the library also provides a number of basic parsers for parsing characters, words, upper case characters, whitespace, etc., and (critically) a range of **combinators**, i.e. higher order functions, for combining parsers (table 7.1 summarises a few interesting combinators, including all mentioned in this chapter and in chapter 12). For

[2]Strictly, this is *direct* left-recursion; *indirect* left recusion involves intermediate symbols but is handled similarly.

Combinator	Meaning
`many p`	Apply parser `p` zero or more times, returning a list of results.
`many1 p`	Like `many` but requires at least one successful application of `p`.
`sepBy p s`	Apply parser `p` zero or more times, separated by application of parser `s`.
`sepBy1 p s`	Like `sepBy` but requires at least one successful application of `p`.
`endBy p s`	Like `sepBy` but also requires a successful application of `s` at the end.
`endBy1 p s`	Like `endBy` but requires at least one successful application of `p`.
`skipMany p`	Apply parser `p` zero or more times, discarding its results.
`skipMany1 p`	Like `skipMany` but requires at least one successful application of `p`.
`option x p`	Try to apply parser `p`; if it fails w/o consuming input, return value `x`.
`choice ps`	Try to apply each parser in list `ps` until one succeeds, then return its value.
`p <|> q`	Try to apply parser `p`; if it fails w/o consuming input, attempt `q`.
`try p`	Try to apply parser `p`; if it fails, backtrack so no input was consumed.

Table 7.1: Some simple Parsec combinators

example[3], given a parser:

```
letter :: Parser Char
```

that is, a stateless parser over a stream of characters which returns a `Char` and, as its name suggests, accepts only letters, we might define:

```
word :: Parser [Char]
word = many1 letter
```

to parse a word up until whitespace; this uses the `many1` combinator, with type signature:

```
many1 :: GenParser tok st a -> GenParser tok st [a]
```

This combinator takes a parser which returns a value of type `a`, and returns a parser which returns a value of type `[a]`, i.e. a list of `a` values; in particular, `many1` applies its input parser as many times as possible before failure, and succeeds provided there is at least one application. In general, parsec's combinators, and the ability to write new ones, provide much of its power, and are a superb example of the power of higher-order programming.

7.2.2.1 Example runs

Consider the following short session in `ghci`, an interactive environment for Haskell:

```
*Parses> parse word "" "hello"
Right "hello"
```

Here `*Parses>` is the interactive environment's prompt, and displays the name of the module defining our current scope (in which we have defined `word`, above). `parse` is a Parsec function for running a parser: its first parameter is the parser to run; its second (blank here) is a filename, nominally the file where the parser's input originated (for error messages); its third parameter is the input to parse.

The result of the call `parse word "" "hello"` is the value `Right "hello"`. This is a value of type `Either ParseError String`, where `Either` is a polymorphic type used where a computation may have two possible outcomes, with different types:

[3]Examples in this section are mostly drawn directly from the Parsec documentation.

```
data Either a b = Left a
                | Right b
    deriving (Eq, Ord)
```

In this context, `Right "hello"` means "successful parse; parse result is the string `"hello"`".

A similar example:

```
*Parses> parse word "" "hello there"
Right "hello"
```

If we try to parse the string `"hello there"`, we get the same result, demonstrating that `many1 letter` does indeed stop when it reaches a non-letter (i.e. when `letter` fails), in this case a space.

An error manifests as a `Left ParseError` result from `parse`:

```
*Parses> parse word "" ""
Left (line 1, column 1):
unexpected end of input
expecting letter
```

In this example, we have attempted to parse an empty string as a word; this is rejected because the `many1` combinator only succeeds if it is able to apply its argument parser at least once.

Finally, we see a similar error with input not consisting entirely of letters:

```
*Parses> parse word "" "1234"
Left (line 1, column 1):
unexpected "1"
expecting letter
```

(Note that if this input had prefixed with letters, `word` would have successfully parsed those letters, until reading the first non-letter — as in the second example, above.)

Now, the nature of a recursive descent parser is that it attempts to parse sequences of symbols, backtracking and trying alternatives on failure. Let us now examine each of these aspects.

7.2.3 Sequencing parsers

Monads provide a mechanism for controlling order of execution in a flexible manner, encoded in the monad's implementation of `(>>=)` and `return`, but in practice we can generally abstract this away using do-notation (see section 7.1.2). Consider the following function:

```
openClose :: Parser Char
openClose = do char '('
               char ')'
```

Here the `char` parser succeeds if it reads the specified character, and fails otherwise; `openClose` thus succeeds if it reads a '`(`' followed by a '`)`'. In general, then, a do-block in a Parsec monad represents sequential application of a number of parsers, and succeeds if each of those parsers succeeds; as soon as one fails, the whole block fails — so in the above, if the first character is *not* '`(`', the second `char` parser is not attempted.

7.2.4 Backtracking and lookahead

Parsec's backtracking and unbounded lookahead is based on control of what happens when a parser fails; in particular, we have two combinators, `<|>` (pronounced 'predictive') and `try`, which work as follows:

- `a <|> b` attempts to apply parser `a`, but if it fails *without consuming any input*, instead attempts to apply parser `b`.
- `try a` attempts to apply parser `a`, but if it fails, it *backtracks* to the point before the attempt was begun.

Thus, we use `try` to control backtracking, and `<|>` to control choice between alternatives.

Let us consider a larger example, though one with no backtracking:

```
separator :: Parser ()
separator = skipMany1 (space <|> char ',')

sentence :: Parser [String]
sentence = do words <- sepBy1 word separator
              oneOf ".?!"
              return words
```

Here `space` attempts to parse the single character ' ' (a space), so `(space <|> char ',')` attempts to parse a space or a comma; there is no `try` here because if `space` fails, no backtracking is necessary: we haven't consumed any input while trying to match that single character. `skipMany1` applies its argument as many times as possible and at least once, discarding the results; thus, `separator` reads (but discards) at least one space or comma.

This allows us to write the `sentence` parser: we use `sepBy1` to read a sequence of words (at least one), separated by spaces and commas, and terminated by either a full stop, a question mark or an exclamation mark; then we return the list of words just read.

An instructive example regarding backtracking is:

```
testOr = string "(a)"
       <|> string "(b)"
```

`string` creates a parser which attempts to parse a specified string. Despite initial appearances, this will fail to parse the string `'(b)'`, because with that input, the parser `string "(a)"` successfully reads a `'('` character then fails on the `'b'`, *having consumed input* — so the second branch of `<|>` is not taken.

The traditional fix would be to refactor the grammar to reduce the lookahead, giving:

```
testOr1 = do char '('
             char 'a' <|> char 'b'
             char ')'
```

However, thanks to the `try` combinator we may encode our intended meaning more directly:

```
testOr2 = try (string "(a)")
        <|> string "(b)"
```

Here we have wrapped the `string "(a)"` parser in a `try` call, so if it fails at any point, the second alternative will be attempted.

In summary: Parsec provides a *domain specific embedded language* [Hud96] for writing backtracking recursive descent parsers in Haskell; as outlined here, it allows very direct encoding of a concrete grammar as Haskell functions, and allows the full power of higher-order programming in Haskell to be used in manipulating and combining the parser functions; when compared with more traditional parsing technologies such as Lex and Yacc [ALSU06], it is thus much more direct and readable, and may be freely extended by the programmer as required by context — which is a much harder proposition in that more traditional setting.

Part II

Design & Implementation

Chapter 8

The Language Implemented

Contents

In this chapter we describe CSP-CASL as formalised and implemented in this project. We provide an overview of its features, survey the various process types and process operators, and summarise various language design decisions made. This serves as an informal but comprehensive introduction to the following chapters which describe in detail the syntax and static semantics we have implemented, and the implementation itself.

8.1 CSP-CASL overview

8.1.1 Introduction

A CSP-CASL specification consists of a data part and a process part (see chapter 4); however, a CASL specification may be reused as the data part of an heterogeneous CSP-CASL specification (see section 5.4). Thus, *we* need define neither the data part of a CSP-CASL specification, nor its 'coarse structure' (e.g. the starting `spec` keyword, specification name, etc.): they are defined in [CoF04c] and [Mos04a][1], respectively. Thus, our primary concern is the process part.

The process part of a CSP-CASL specification consists of an optional *channel declarations* part, followed by the keyword `process`, followed by a sequence of *process declarations* and *process equations*, interspersed with one another and subject to linear visibility. Drilling down a little further:

- a channel declaration declares a list of channel names and their associated sort;
- a process declaration declares a process name, the sorts of any parameters it takes, and its *alphabet* of permitted communications;

[1] Actually, [Mos04a] defines HETCASL's syntax but not its static semantics; still, the latter is essentially that of CASL structured specifications defined in [BCH+04, III:4].

- a process equation then associates a previously-declared process name with a *process term*, built from primitive processes via process operators.

A key feature of a process algebra is of course communication between processes; in CSP-CASL, we have individual communication *events* having a particular sort and possibly occurring over a channel, and *event sets*, which describe collections of possible communications. In general, collections of possible and actual communications are characterised in CSP-CASL by *communication alphabets*, which are essentially sets of sort and channel names.

For the rest of this section, we describe each of these elements in more depth, and informally introduce syntax and aspects of the static semantics as appropriate. As we are introducing the concrete syntax, all of the examples in this section are given in machine readable form (i.e. ASCII).

8.1.2 Communication alphabets

A number of CSP-CASL process operators involve communication between processes. In particular:

- The various prefix operators each communicate an *event*, having a particular sort, possibly over a channel.
- Run, chaos, generalised parallel, and alphabetised parallel involve communications chosen from an *event set*. An event set is not, as the name suggests, simply a set of events. Rather, it is a set of sort names and channel names, where the communication event engaged in must have a sort found in the event set (modulo channels & subsorting).
- Furthermore, the hiding operator involves event sets: hiding hides sorts/channels named in an event set.

A process' possible and actual communications may, thus, be characterised by a *communication alphabet*, which describes a set of communications. Specifically, it is a set containing:

1. CASL sort names — the sorts of non-channel communications in the alphabet;
2. CSP-CASL channel names — the channels of channel communications[2].

(Clearly, then, an event set is then simply a communication alphabet, albeit in a particular context.)

Specifically, we refer to a process' possible communications as its *alphabet of permitted communications*; these are declared in process declarations (see section 8.1.4). We refer to a process' actual communications as its *constituent alphabet*; these arise recursively from process terms as described in sections 8.2 and 10.2.3.

Syntactically, a communication alphabet (or event set) is a list of comma-separated sort and channel names, freely mixed. For example, in:

```
RUN(a,b,c,d)
```

the list `a,b,c,d` is an event set, and is valid provided `a`, `b`, `c` and `d` are sort names or channel names (syntactically, at least, they are: sort names and channel names are CASL `SIMPLE-IDs`).

[2]Or rather, *typed channel names*, which are (channel name, sort) pairs — in order to deal properly with subsorting: see section 10.2.3 for a proper treatment.

Repetition is allowed and of no consequence, as the (syntactic) list is transformed to a (semantic) set.

Semantically, we require that the names in a communication alphabet are known, i.e. have been declared previously in the data part or a channel declaration. We distinguish between a communication alphabet's sort names and channel names at the semantic level, by performing this lookup; this introduces the restriction that channel names and sort names must be distinct. This is discussed further in section 8.3.

Communication alphabets are checked at two points in CSP-CASL's static semantics, namely in considering alphabetised processes, and process equations. This is described further in section 10.2.3.

8.1.3 Channel declarations

Conceptually, a channel is an entity 'over which' a communication may occur (though a channel is not *required* for a communication to occur). In both CSP and CSP-CASL, a channel is in fact nothing more than a 'tag' attached to a communication — written as a prefix.

Now, a CSP-CASL channel has a sort, which is just a CASL sort defined in the specification's data part: only communications whose sort matches that of the channel (modulo subsorting) may occur over that channel. A *channel declaration* introduces new channel names, and binds each of them to a CASL sort. Attempting later to use a channel which has not been declared is an error.

Syntactically, a channel declaration is a comma-separated list of channel names (CASL `SIMPLE-IDs`), followed by a colon, followed by a sort name. A channel declarations section of a CSP-CASL specification is then a semicolon-separated sequence of channel declarations, preceded by the keyword `channel` or `channels`. For example:

```
channels
  a, b, c : d;
  e, f, g : h
```

There are three semantic restrictions on channel declarations:

1. The sorts must be known, i.e. must have been declared in the data part.
2. A channel may not have the same name as a sort; we require distinct names in order to be able to properly distinguish members of a communication alphabet (see section 8.1.2).
3. A channel name may not be declared more than once with different sorts, for obvious reasons. Multiple declarations of a channel name with the same sort are accepted, but raise a warning in our implementation.

8.1.3.1 Channels in CSP vs channels in CSP-CASL

CSP supports two channel forms which are not supported directly in CSP-CASL:

- *Recursive* — e.g. we can have `d.c.b.a` where `a` is the original communication, and `b`, `c` and `d` are all channel names.

- *Indexed* — e.g. we can have `c[i].a`, where `a` is the original communication, and `c[i]` represents an indexed family of channels.

In CSP-CASL we make the design decision that only 'simple' channel names are supported; we do not *directly* support recursive or indexed channels: rather, a CSP-CASL communication event is either a simple event or an event sent across a single channel.

The rationale behind this decision is that the structure introduced by a recursive or indexed channel name may (and arguably *should*) be defined using CASL. That is, while CSP-CASL does not directly support these channel forms, a channel's sort is a defined in CASL, and CASL is strong enough to emulate these structures in a sort definition. Conversely, supporting these extra channel forms directly in CSP-CASL would complicate the syntax and static semantics. A future version might possibly support these forms (either as 'syntactic sugar' or fully integrated) — however, a strong argument could be made that CASL is the right place to define such structure anyway.

8.1.4 Process declarations

A *process declaration* declares a new process name, which may subsequently be used in process equations. In the declaration, the process name has associated with it the sorts of the process' expected parameters (if any), and the sorts and channel names constituting the process' *alphabet of permitted communications.*

Syntactically, a process declaration consists of the following, in this order:

1. a process name (a simple identifier);
2. an optional, parenthesised, comma-separated list of CASL sort names — the sorts of the process' parameters;
3. a colon;
4. an optional comma-separated list of CASL sort names and CSP-CASL channel names — the process' alphabet of permitted communications (see section 8.1.2);
5. a semicolon.

For example, the following are both syntactally valid process declarations:

```
P:;
Q(a,a,b): a,b,c;
```

Semantic restrictions on a process declaration are as follows.

- A parameter sort list must contain only sorts which are known in the data part. Similarly, as described in section 8.1.2, the permitted communication alphabet must contain previously-declared sort names and channel names.

  Thus, in the example above, we require that `a` and `b` are known CASL sort names, and that `c` is either a known CASL sort name or a known CSP-CASL channel name.

Note that a process declaration's parameter list and alphabet of permitted communications may both be empty, as in the `P` example above. Obviously processes need not have any parameters; similarly, while an empty communication alphabet appears initially strange, it is in fact perfectly valid. For example, the process `STOP` engages in no communication, and so

could legitimately be bound, in a process equation, to `P`. (See section 8.1.5 for more on process equations.)

8.1.5 Process equations

A process equation binds a process name to a process term. Syntactically, it consists of a *parametrised process name* followed by an 'equals' sign, followed by a process term.

A parametrised process name consists of a process name followed by an optional parenthesised list of variable names — the *process-global variables* of the process (see section 8.1.7). If the variable name list is present and contains more than one element, the elements are comma-separated. Process terms are described further in section 8.1.6 — they correspond to what we might loosely think of as 'a process' such as `STOP`, `x -> P`, `P [| a,b |] y -> Q`, etc.

For example, the following three lines each contain a single process equation:

```
P = STOP
Q(a,b,c) = a -> P |~| (b -> Q ; c -> Z(c))
Z(d) = [] x :: s -> Q(x,x,d)
```

Semantic requirements on process equations are as follows:

- The process name must have already been declared in a process declaration (see section 8.1.4).
- The length of the parameter list must be the same as that of the parameter sort list in the corresponding process declaration.
- Each variable name in the parameter list must be unique within the list.
- The constituent alphabet of the process term on the right hand side of the process equation must be contained in the permitted alphabet of the process name found on the left hand side (see section 10.2.3 for further discussion of this important aspect of the static semantics).

8.1.6 Process terms

Process terms are constructed recursively from primitive processes, (references to) named processes, and applications of process operators on process terms, as follows:

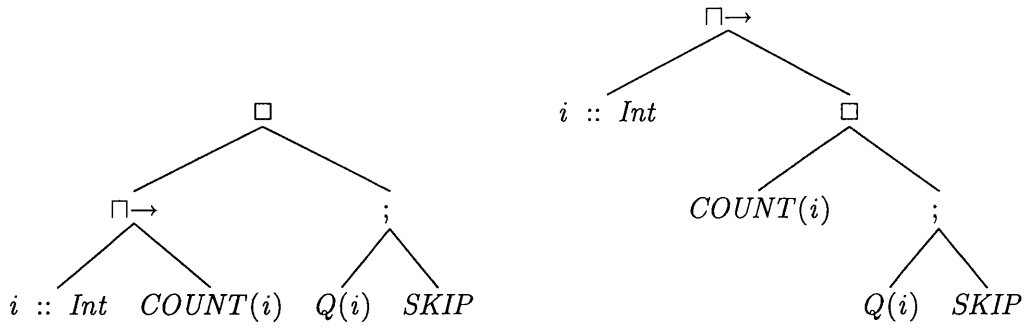
- A reference to a **named process** is a process term — see section 8.2.2. Examples:

  ```
  Q
  COUNT(i+1)
  ```

- A **primitive process** is a process term — see section 8.2.3. Examples:

  ```
  SKIP
  RUN(a,b)
  ```

- A **process operator** applied to one or two process terms (as appropriate for the operator) is a process term — see sections 8.2.4 to 8.2.10. Examples:



(a) Normal operator precedence (b) Enforced left-to-right operator precedence

Figure 8.1: Tree representations of process term $\sqcap i :: Nat \rightarrow COUNT(i) \,\Box\, Q(i)\ ;\ SKIP$.

```
x -> Q
x -> y -> Q
|~| i :: Nat -> COUNT(i+1) [] Q(i) ; SKIP
```

Figure 8.1(a) illustrates the process term in this final example as a tree whose inner nodes are process operators (see section 8.2), and whose leaves are either parameters of those operators or named processes. The structure of this tree is dictated by the process operator precedence rules presented in section 8.2.1.

- A **parenthesised process term** is a process term. Parentheses may be used to indicate desired structure of a term where it varies from that arising automatically from the process operator precedence rules presented in section 8.2.1. Examples:

```
(|~| i :: Nat -> COUNT(i+1)) [] (Q(i) ; SKIP)
|~| i :: Nat -> (COUNT(i+1) [] (Q(i) ; SKIP))
```

The first of these has the same tree structure as in the previous example, explicitly represented using parentheses; again, its structure is illustrated in figure 8.1(a). The second example uses parentheses to enforce a linear 'left-to-right' reading of the process operators; its structure is illustrated in figure 8.1(b).

As mentioned in section 8.1.2, a process term has a constituent alphabet describing the communications appearing in that process; how these are computed is described in section 10.2.3.1.

8.1.7 Process variables: global and local

CSP-CASL processes, like CSP processes, may make use of *variables*. In CSP-CASL, however, unlike in CSP, we explicitly acknowledge the distinction between *process-global* and *process-local* variables. CSP takes a declarative view on variables declared locally to a process: when the process has terminated, all information on such variables vanishes; this is complemented by recursive process definitions, where a variable serves to define (potentially infinitely) many processes, naming one process for every value the variable can take [Ros98] [MR07, §3.2]. Setting this in the context of CSP-CASL process terms, then, we have the following:

- A **process-global** variable is one which arises from the parameter list of the parametrised process name part of a process equation (see section 8.1.5). For example, in:

```
P(x) = x -> SKIP
```

the variable x is process-global.

The scope of a process-global variable is the entire process term on the right hand side of the process equation.

- A **process-local** variable is one which arises as the result of a prefix term introducing a new binding, i.e. in an internal prefix choice, an external prefix choice, a channel nondeterministic send, or a channel receive (see section 8.2.4). For example, in:

```
P(x) = |~| i :: Nat -> (COUNT(i+1) [] (Q(i) ; SKIP))
```

the variable i is process-local, whereas x is, as above, process-global (and not actually used on the right hand side in this example).

A process-local variable arises in a prefix term; its scope is the remainder of that prefix term *until* a sequential composition operator is encountered — sequential composition causes all process-local variable bindings to be erased (see section 8.2.5). Considering our example above, we see that the right hand side of the process equation is exactly the process term illustrated in figure 8.1(b); thus, we see that the scope of i includes the terms `COUNT(i+1)` and `Q(i)` but *not* the `SKIP`.

8.2 Survey of process types and operators

In this section, we survey the various processes and process operators which may occur in CSP-CASL process terms in more detail. Examples are given in pretty-printed LaTeX throughout — refer to table 8.1 for a summary of machine-readable and pretty-printed syntax, as well as operator precedences (see section 8.2.1).

While we introduce a few aspects of the static semantics, we do not describe the computation of a term's constituent alphabet here, however: see section 10.2.3.1.

8.2.1 Precedence of process operators

The abstract syntax for process terms presented in section 9.1.2 is quite ambiguous; since, in practice, an unique parse tree is required for each process term, we define levels of precedence for the various process operators as follows, starting with the highest precedence ('binds most tightly'). These levels have been chosen to match those seen in CSP [FSE06], and are also summarised in table 8.1.

- 0 — named processes and primitive processes (including $RUN()$ and $CHAOS()$) are atomic;
- 1 — hiding and renaming;
- 2 — prefix operators of all kinds;
- 3 — sequential composition;
- 4 — external and internal choice;
- 5 — parallel operators of all kinds;
- 6 — conditional.

Element	Machine readable syntax	Pretty printed	Precedence
Named process	P `(`t_1`, ..., `t_n`)`	$P(t_1, \ldots, t_n)$	0
Skip	`SKIP`	*SKIP*	0
Stop	`STOP`	*STOP*	0
Div	`DIV`	*DIV*	0
Run	`RUN (`es`)`	$RUN(es)$	0
Chaos	`CHAOS (`es`)`	$CHAOS(es)$	0
Term prefix	t`->`p	$t \rightarrow p$	1
Internal prefix choice	`|~|` x`::`s`->`p	$\sqcap x :: s \rightarrow p$	1
External prefix choice	`[]` x`::`s`->`p	$\Box x :: s \rightarrow p$	1
Channel send	c`!`t`->`p	$c!t \rightarrow p$	1
Channel nondeterministic send	c`!`x`::`s`->`p	$c!x :: s \rightarrow p$	1
Channel receive	c`?`x`::`s`->`p	$c?x :: s \rightarrow p$	1
Sequential composition	p`;`q	p ; q	2
Internal choice	p`|~|`q	$p \sqcap q$	3
External choice	p`[]`q	$p \Box q$	3
Interleaving	p`|||`q	$p \mid\mid\mid q$	4
Synchronous	p`||`q	$p \parallel q$	4
Generalised parallel	p`[|`es`|]`q	$p [\![es]\!] q$	4
Alphabetised parallel	p`[`es_1`||`es_2`]`q	$p [\![es_1 \| \| es_2]\!] q$	4
Hiding	p`\`es	$p \setminus es$	5
Renaming	p`[[`R`]]`	$p[R]$	5
Conditional	`if` φ `then` p `else` q	if φ then p else q	6

Table 8.1: Summary of CSP-CASL process types.

The precedence levels are encoded in the structure of the concrete grammar presented in section 9.3. Naturally, and as described in section 8.1.6, these precedences may be over-ridden using parentheses in the usual manner.

For operators at the same level of precedence, and where it is not otherwise clear (i.e. for the choice and parallel operators), we read from left to right in the absence of parentheses. Thus, for example, $X \Box Y \sqcap Z$ is parsed as $(X \Box Y) \sqcap Z$.

8.2.2 Named processes

A **named process** $P(t_1, \ldots, t_n)$ is a reference to a process name P previously declared in a process declaration, passing the vales of the CASL terms $t_1, \ldots, t_n$ to its parameters.

If the process expects no parameters, the parentheses must be omitted.

In the static semantics, we require the term list to be the same length as the declaration's parameter list, and that each term's overall sort is compatible with the corresponding CASL sort in the parameter list declaration (see section 10.2.4.2 for a discussion of what we mean by 'compatible').

8.2.3 Primitive processes

Skip: The process

$SKIP$

represents immediate successful termination. It never engages in any communication.

Stop: The process

$STOP$

represents deadlock: it never communicates and never terminates.

Div: The process

DIV

represents livelock: it does nothing except diverge — an infinite sequence of internal, non-observable actions.

Run: Let es be an event set. Then

$RUN(es)$

can always communicate any member of es desired by the environment.

Chaos: Let es be an event set. Then

$CHAOS(es)$

can always choose to communicate some member of es or to reject everything.

8.2.4 Prefix

CSP-CASL has six prefix operators, where a process engages in a communication then behaves like some other process; three of these operators are essentially 'channel versions' of the other three:

Term prefix: Let t be a CASL term, and P a process. Then

$$t \rightarrow P$$

communicates the value of t, then behaves like P.

Internal prefix choice: Let x be a variable name, s a CASL sort, and P a process. Then

$$\sqcap x :: s \rightarrow P$$

communicates a particular s-sorted value v, then behaves like P with x bound to v as a process-local variable.

External prefix choice: Let x be a variable name, s a CASL sort, and P a process. Then

$$\Box x :: s \rightarrow P$$

communicates any s-sorted value v, then behaves like P with x bound to v as a process-local variable.

Channel send: Let t be a CASL term of sort s, c a channel whose sort is a supersort of s, and P a process. Then

$$c!t \rightarrow P$$

communicates the value of t over the channel c, then behaves like P. This can be seen as the 'channel version' of term prefix, above.

In the static semantics, we require that c is known and that the overall sort of t is a subsort of the sort of c.

Channel nondeterministic send: Let x be a variable name, s a CASL sort, c a channel, and P a process. Then

$$c!x :: s \rightarrow P$$

communicates a particular s-sorted value v over the channel c, then behaves like P with x bound to v as a process-local variable. This can be seen as the 'channel version' of internal prefix choice, above.

In the static semantics, we require that s and c are known, and that s is a subsort of the sort of c. Thus, the value communicated need not be of the exact same sort as the channel: the specifier may *restrict* the values communicated, provided they do so in line with CASL subsorting.

Channel receive: Let x be a variable name, s a CASL sort, c a channel, and P a process. Then

$$c?x :: s \rightarrow P$$

communicates any s-sorted value v over the channel c, then behaves like P with x bound to v as a process-local variable. This can be seen as the 'channel version' of external prefix choice, above.

In the static semantics, we require that s and c are known, and that s is a subsort of the sort of c. Thus, the value communicated need not be of the exact same sort as the channel: the specifier may *restrict* the values communicated, provided they do so in line with CASL subsorting.

8.2.5 Sequential composition

Sequential composition: Let P and Q be processes. Then

$$P \; ; \; Q$$

first behaves like P, then when P terminates successfully, it behaves like Q. If P never terminates successfully, neither does $P \; ; \; Q$.

As discussed in section 8.1.7, any process-local variables in scope in P fall out of scope in Q.

8.2.6 Internal and external choice

Internal choice: Let P and Q be processes. Then

$$P \sqcap Q$$

behaves either like P or Q, where the choice between them is made arbitrarily, without the knowledge or the control of the environment.

External choice: Let P and Q be processes. Then

$$P \Box Q$$

offers the environment the choice of the first events of P and Q and then behaves accordingly.

8.2.7 Parallel operators

Interleaving: Let P and Q be processes. Then

$$P \;|||\; Q$$

is a process where P and Q run independently of each other: any event which $P \;|||\; Q$ communicates arises in precisely one of P and Q; if they could both have communicated the same event then the choice of which one actually did so is nondeterministic.

Synchronous: Let P and Q be processes. Then

$$P \;||\; Q$$

is a process where P and Q agree on all events that occur; any event communicated by $P \;||\; Q$ is communicated by P and Q simultaneously.

Generalised parallel: Let P and Q be processes, and es an event set. Then

$$P \;[\![es]\!]\; Q$$

is a process where all values in es are synchronized (if communicated, must be communicated by both P and Q) , and values outside es are independent (may be communicated by either P and Q, but never both in sychrony).

In the static semantics, we require es to be a valid event set (i.e. contain only known sorts and channels).

Alphabetised parallel: Let P and Q be processes, and es_1 and es_2 event sets. Then

$$P \,[\![es_1 || es_2]\!]\, Q$$

is a process in which P may communicate any event in es_1, and Q may communicate any event in es_2, but they must synchronize on any communications in the intersection of es_1 and e_2.

In the static semantics, we require es_1 and es_2 to be valid event sets (i.e. contain only known sorts and channels), and we require them to be contained in the constituent alphabets of P and Q, respectively.

8.2.8 Hiding

Hiding: Let P be a process, and es an event set. Then

$$P \setminus es$$

behaves like P except that any communications in es are hidden; i.e. while they may still occur inside P, they are invisible to and uncontrollable by anything outside P.

In the static semantics, we require es to be a valid event set (i.e. contain only known sorts and channels).

8.2.9 Renaming

Renaming: Let P be a process and $r_1, \ldots, r_n$ be CASL IDs. Then

$$P[r_1, \ldots, r_n]$$

behaves like P except that any communications it engages in are subjected to conversion by each of the renaming items $r_1, \ldots r_n$ in turn.

Syntactically, the renaming items r_i are just CASL IDs. Semantically, we require that each r_i is be either a known CASL unary (total or partial) function

$$r_i : s_{i,1} \rightarrow s_{i,2}$$

or a known CASL binary predicate

$$r_i : s_{i,1} \times s_{i,2}$$

Syntactically, the two cases are indistinguishable; we distinguish between the two, and determine the sorts involved, at the semantic level.

8.2.10 Conditional

Conditional: Let P_1 and P_2 be processes, and φ a CASL formula. Then

$$\text{if } \varphi \text{ then } P \text{ else } Q$$

behaves like P_1 if φ evaluates as true, and otherwise behaves like P_2.

8.3 Some design decisions

Here we collect and summarise some notable design decisions made during the development of CSP-CASL.

8.3.1 Communication alphabets distinguished at semantic level

Communication alphabets (and event sets) are undistinguished lists of sort and channel names; we distinguish between the two at the semantic level, by looking up sort names and channel names in the signature. This simplifies the syntax (the alternative approach would be to list first one kind, then a separator, then the other kind) and gives the specifier greater freedom, while introducing the requirement that sort names and channel names are distinct; this seems to be a quite reasonable requirement, however.

8.3.2 No recursive / indexed channels

As noted in section 8.1.3.1, recursive and indexed channels are not directly supported by CSP-CASL, as these structures can (and arguably should) be simulated in CASL sorts definitions.

8.3.3 Treatment of multiple channel declarations

We reject multiple declarations of the same channel with different sorts, for obvious reasons — see section 8.1.3. An alternative approach would be to allow them and only kept the last, so that later declarations shadow earlier ones; we argue, however, that this could lead to subtle errors which were harder to find later on, and that the extra burden on the specifier at this stage is small and reasonable.

In line with CASL, we allow multiple declarations with the same sort, since that will not lead directly to errors later — but we raise a warning in our implementation.

8.3.4 Explicit process declarations vs implicit inference

We require that processes are declared before use, and that their permitted communication alphabets are declared and 'match' the actual constituent alphabets arising from process terms. An alternative approach would have been to *not* require process declarations, and instead infer the required process names, parameter sort lists, and communication alphabets from their usage in process terms.

Our approach does admittedly place an added burden on the specifier, and has largely been chosen in order to simplify the task of formalising and implementing CSP-CASL's static semantics at this stage. A future version of the language might relax these requirements.

However, note that even in the presence of inference, one would not wish to do away with process declarations altogether. Not only do they provide useful documentation and make a specification easier to understand, they also allow the specifier to introduce tighter constraints on processes than those arising from the inference algorithm. In particular, they are essential in order to support *loose* specification of processes in a structured setting.

Consider, for example, the specification LOOSE, below. This has been written in a (currently) fictional dialect of CSP-CASL with inference of process alphabets, and no process declarations. Here, the process Q is loosely specified. Conceptually, all the specifier wishes to say about it is that it occurs after P has communicated a then b then c.

spec LOOSE =
  data sorts S, T, U
    ops $a : S; b : T; c : U$
  process $P = a \rightarrow b \rightarrow c \rightarrow Q$
end

Now consider the task of inferring process alphabets for P and Q. We would expect an algorithm to produce the smallest possible 'safe' answer. For P, such an algorithm would produce the answer $\{S, T, U\}$ (consider the sorts of a, b and c). For Q, it would be the empty set.

However, what if the specifier's intention is that Q's alphabet should be $\{S\}$, say — or indeed anything other than $\emptyset$? With inference but no declarations, this is impossible to specify. Within the scope of LOOSE this is unimportant, but in a structured setting, properly specifying the alphabet of Q might be of utmost importance (i.e. elsewhere). Hence, process declarations are required, even in the presence of inference.

8.3.5 Equational vs let-expression style

We write processes in an equational style rather than in the '*let*-expression' style seen in earlier CSP-CASL papers, in particular [GRS05]. That is, systems of processes are written as, for example:

$X = STOP$
$Y = SKIP$
$Z = a \rightarrow X \,\Box\, Y$

rather than (say):

let $X = STOP$
  $Y = SKIP$
in $a \rightarrow X \,\Box\, Y$

Note that the first of these defines a system of several processes, whereas the second really describes just one process. Equational style is thus more general. If, in practice, we wish to hide some processes while specifying 'in the large', this can be done using HETCASL's (CASL-inherited) structuring mechanisms — in particular the **local** ... **within** mechanism (see [BM04, §6.4]).

A potentially interesting future extension to CSP-CASL would be the addition of let-expressions at the process operator level; this would provide a fully generalised process equation scoping mechanism similar to that seen for functions in Haskell (say). Then one could write, e.g.:

$X = STOP$
$Z =$ **let** $Y = SKIP$
  in $a \rightarrow X \,\Box\, Y$

to restrict the scope of Y in a more pleasantly localised manner than is possible at present using **local** ... **within**.

8.3.6 Use of : : in prefix process operator syntax

The syntax $x :: s$ (meaning 'x is a variable of sort s') in certain of the prefix operators was chosen because the more obvious $x : s$ form also parses as a CASL term and so is unsuitable — a channel nondeterministic send could be confused for a channel send. We chose a double colon because it is used to declare an entity's type in Haskell, and so seemed a reasonable second choice here.

8.3.7 Renaming items distinguished at semantic level

Similarly to the situation with communication alphabets noted above: renaming items are undistinguished lists of function and predicate names; we distinguish between the two at the semantic level, by looking up the names in the specification's CASL signature. Again, this simplifies the syntax, since we don't have to 'artificially' distinguish the two categories. Doing so would be particularly onerous in this case, because the order of renaming items matters, so we couldn't just 'list the functions first, then a separator, then the predicates', for example. However, note that CASL does not enforce distinct function and predicate names (though the HETS tool does raise a warning when a clash occurs), so the order in which we perform our lookup is significant. In CSP-CASL we adopt the convention that if a renaming item has a name which is both a binary function and a predicate, we interpret it as the function (a function of the wrong profile may be ignored).

8.3.8 Variable names in parametrised process names must be unique

We reject multiple declarations of the same parameter name in a parametrised process name. While this seems an obvious restriction, it could (possibly) be usefully relaxed in the future. Consider the following specification:

spec XX =
  data sort $Nat < Rat$
    op $n : Nat$
  process $P(Nat, Rat) : Rat$;
    $Q(Rat) : Rat$;
    $P(x, x) = x \rightarrow SKIP \,\Box\, Q\,(\,x\,)$
end

This is currently invalid, due to the repetition of x in P's variable list. However, looking at P's parameter sort list we see that it might in fact be reasonable to use the same variable in two slots: in the first, it is treated as sort Nat, in the second as Rat. Since Nat is a subsort of Rat, this is fine *provided* the value passed in is not in $Rat \setminus Nat$.

It is unclear at this stage how useful such a relaxation would be, or its greater implications for the semantics; as such, the restriction remains in place but we note that future work would investigate this further.

Chapter 9

CSP-CASL Syntax

Contents

In this chapter, we formally present the syntax of CSP-CASL, following closely the style of [CoF04b]: the three grammars here are presented in the order seen there, and the grammars and production rules are written using syntax seen there (see in particular [CoF04b, §II:2]).

- In section 9.1 we present the 'normal grammar' of the abstract syntax, which forms the basis of the CSP-CASL static semantics; each syntactic category appearing in the normal grammar has a corresponding rule or set of rules in chapter 10.
- In section 9.2, again following [CoF04b], we present the 'abbreviated grammar' of the abstract syntax, providing a more concise view of the syntactic categories — one which corresponds more closely to the Haskell algebraic data types used to represent the abstract syntax in our implementation (see chapter 12).
- Finally, in section 9.3, we present the concrete syntax of CSP-CASL as implemented lately in HETS using Parsec.

In each of these grammars, a number of non-terminals refer to members of the corresponding CASL grammar; we note these in each case.

9.1 CSP-CASL abstract syntax — normal grammar

The following nonterminal symbols correspond to nonterminals in CASL, and are left unspecified in the CSP-CASL abstract syntax: `FORMULA`, `ID`, `SIMPLE-ID`, `SORT`, `TERM`, `VAR`.

9.1.1 Top-level elements

```
CSP-BASIC-SPEC ::= csp-basic-spec CHAN-DECLS PROC-ITEMS

CHAN-DECLS     ::= chan-decls CHAN-DECL*
```

```
CHAN-DECL       ::= chan-decl CHAN-NAMES CHAN-TYPE
CHAN-NAMES      ::= chan-names CHAN-NAME+
CHAN-NAME       ::= chan-name SIMPLE-ID
CHAN-TYPE       ::= chan-type SORT

PROC-ITEMS      ::= proc-items PROC-ITEM+
PROC-ITEM       ::= PROC-DECL | PROC-EQ

PROC-DECL       ::= proc-decl PROC-NAME PROC-ARGS PROC-ALPHA
PROC-NAME       ::= proc-name SIMPLE-ID
PROC-ARGS       ::= proc-args SORT*
PROC-ALPHA      ::= proc-alphabet COMM-TYPE*

PROC-EQ         ::= proc-eq PARM-PROCNAME PROCESS
PARM-PROCNAME   ::= parm-procname PROC-NAME PROC-VARS
PROC-VARS       ::= proc-vars PROC-VAR*
PROC-VAR        ::= proc-var VAR
```

9.1.2 Processes

```
PROCESS         ::= NAMED-PROC | BASIC-PROC | PREFIX-PROC
                  | INTPRE-PROC | EXTPRE-PROC | SEQ-PROC
                  | INTCHOICE-PROC | EXTCHOICE-PROC | INTPAR-PROC
                  | SYNPAR-PROC | GENPAR-PROC | ALPHAPAR-PROC
                  | HIDING-PROC | RENAMING-PROC | CONDITION-PROC
NAMED-PROC      ::= named-proc PROC-NAME TERM*
BASIC-PROC      ::= SKIP-PROC | STOP-PROC | DIV-PROC | RUN-PROC
                  | CHAOS-PROC
SKIP-PROC       ::= skip-proc
STOP-PROC       ::= stop-proc
DIV-PROC        ::= div-proc
RUN-PROC        ::= run-proc EVENT-SET
CHAOS-PROC      ::= chaos-proc EVENT-SET
PREFIX-PROC     ::= prefix-proc EVENT PROCESS
INTPRE-PROC     ::= intpre-proc SVAR-DECL PROCESS
EXTPRE-PROC     ::= extpre-proc SVAR-DECL PROCESS
SEQ-PROC        ::= seq-proc PROCESS PROCESS
INTCHOICE-PROC  ::= intchoice-proc PROCESS PROCESS
EXTCHOICE-PROC  ::= extchoice-proc PROCESS PROCESS
INTPAR-PROC     ::= intpar-proc PROCESS PROCESS
SYNPAR-PROC     ::= synpar-proc PROCESS PROCESS
GENPAR-PROC     ::= genpar-proc EVENT-SET PROCESS PROCESS
ALPHAPAR-PROC   ::= alphapar-proc EVENT-SET EVENT-SET PROCESS PROCESS
HIDING-PROC     ::= hiding-proc PROCESS EVENT-SET
RENAMING-PROC   ::= renaming-proc PROCESS RENAMING
CONDITION-PROC  ::= condition-proc FORMULA PROCESS PROCESS
```

9.1.3 Supporting items

```
EVENT-SET       ::= event-set COMM-TYPE*

COMM-TYPE       ::= comm-type SIMPLE-ID
```

```
EVENT          ::= term-event TERM
                 | chan-send CHAN-NAME TERM
                 | chan-nondet-send CHAN-NAME VAR SORT
                 | chan-recv CHAN-NAME VAR SORT

SVAR-DECL      ::= svar-decl VAR SORT

RENAMING       ::= renaming RENAMING-ITEM+

RENAMING-ITEM  ::= renaming-item ID
```

9.2 CSP-CASL abstract syntax — abbreviated grammar

This abbreviated grammar defines the same language as the one in section 9.1. It was obtained by eliminating each nonterminal that occurs only once as an alternative. The categories in this grammar correspond closely to the Haskell algebraic datatypes used to represent the abstract syntax in our implementation (see section 12.2.1).

As in section 9.1, the following nonterminal symbols correspond to CASL nonterminals in CASL, and are left unspecified in the CSP-CASL abstract syntax: FORMULA, ID, SIMPLE-ID, SORT, TERM, VAR.

9.2.1 Top-level elements

```
CSP-BASIC-SPEC ::= csp-basic-spec CHAN-DECLS PROC-ITEMS

CHAN-DECLS     ::= chan-decls CHAN-DECL*
CHAN-DECL      ::= chan-decl CHAN-NAME+ SORT
CHAN-NAME      ::= chan-name SIMPLE-ID

PROC-ITEMS     ::= proc-items PROC-ITEM+
PROC-ITEM      ::= proc-decl PROC-NAME SORT* SIMPLE-ID*
                 | proc-eq PROC-NAME VAR* PROCESS
PROC-NAME      ::= proc-name SIMPLE-ID
```

9.2.2 Processes

```
PROCESS        ::= named-proc PROC-NAME TERM*
                 | skip-proc
                 | stop-proc
                 | div-proc
                 | run-proc EVENT-SET
                 | chaos-proc EVENT-SET
                 | prefix-proc EVENT PROCESS
                 | intpre-proc SVAR-DECL PROCESS
                 | extpre-proc SVAR-DECL PROCESS
                 | seq-proc PROCESS PROCESS
                 | intchoice-proc PROCESS PROCESS
                 | extchoice-proc PROCESS PROCESS
                 | intpar-proc PROCESS PROCESS
                 | synpar-proc PROCESS PROCESS
```

```
                | genpar-proc EVENT-SET PROCESS PROCESS
                | alphapar-proc EVENT-SET EVENT-SET PROCESS PROCESS
                | hiding-proc PROCESS EVENT-SET
                | renaming-proc PROCESS ID+
                | condition-proc FORMULA PROCESS PROCESS
```

9.2.3 Supporting items

```
EVENT-SET       ::= event-set SIMPLE-ID*

EVENT           ::= term-event TERM
                  | channel-send CHAN-NAME TERM
                  | channel-nondet-send CHAN-NAME VAR SORT
                  | channel-recv CHAN-NAME VAR SORT

SVAR-DECL       ::= svar-decl VAR SORT
```

9.3 CSP-CASL concrete syntax

Here is the concrete grammar for the process part of a CSP-CASL specification.

As in section 9.1, the following nonterminal symbols correspond to CASL nonterminals in CASL, and are left unspecified in the CSP-CASL abstract syntax: FORMULA, ID, SIMPLE-ID, SORT, TERM, VAR.

```
CSP-BASIC-SPEC  ::= CHAN-PART PROC-PART
                  | PROC-PART

CHAN-PART       ::= channel CHAN-DECLS
                  | channels CHAN-DECLS

CHAN-DECLS      ::= CHAN-DECL
                  | CHAN-DECL ; CHAN-DECLS

CHAN-DECL       ::= CHAN-NAMES : SORT

CHAN-NAMES      ::= CHAN-NAME
                  | CHAN-NAME , CHAN-NAMES

CHAN-NAME       ::= SIMPLE-ID

PROC-PART       ::= process PROC-ITEMS

PROC-ITEMS      ::= PROC-ITEM
                  | PROC-ITEM PROC-ITEMS

PROC-ITEM       ::= PROC-DECL
                  | PROC-EQ

PROC-DECL       ::= PROC-NAME ( SORTS ) : PROC-ALPHABET ;
                  | PROC-NAME : PROC-ALPHABET ;
```

```
PROC-NAME       ::= SIMPLE-ID

SORTS           ::= SORT
                  | SORT , SORTS

PROC-ALPHABET   ::= COMM-TYPES

COMM-TYPES      ::= COMM-TYPE
                  | COMM-TYPES , COMM-TYPE

COMM-TYPE       ::= SIMPLE-ID

PROC-EQ         ::= PARM-PROCNAME = PROCESS

PARM-PROCNAME   ::= PROC-NAME
                  | PROC-NAME ( VAR-LIST )

VAR-LIST        ::= VAR
                  | VAR , VAR-LIST

EVENT-SET       ::= COMM-TYPES

PROCESS         ::= COND-PROC
                  | PAR-PROC

COND-PROC       ::= if FORMULA then PROCESS else PROCESS

PAR-PROC        ::= CHOICE-PROC
                  | PAR-PROC "||" CHOICE-PROC
                  | PAR-PROC "|||" CHOICE-PROC
                  | PAR-PROC [ EVENT-SET ] CHOICE-PROC
                  | PAR-PROC [ EVENT-SET "||" EVENT-SET ] CHOICE-PROC

CHOICE-PROC     ::= SEQ-PROC
                  | CHOICE-PROC [] SEQ-PROC
                  | CHOICE-PROC "|~|" SEQ-PROC

SEQ-PROC        ::= PREF-PROC
                  | SEQ-PROC ; PREF-PROC

PREF-PROC       ::= HID-REN-PROC
                  | EVENT -> PREF-PROC
                  | [] SVAR-DECL -> PREF-PROC
                  | "|~|" SVAR-DECL -> PREF-PROC

EVENT           ::= TERM
                  | CHAN-NAME ! TERM
                  | CHAN-NAME ! VAR :: SORT
                  | CHAN-NAME ? VAR :: SORT

SVAR-DECL       ::= VAR :: SORT

HID-REN-PROC    ::= PRIM-PROC HID-REN-PROC-W

HID-REN-PROC-W  ::= \ EVENT-SET
                  | [[ RENAMING ]]
```

```
RENAMING          ::= RENAMING-ITEM
                    | RENAMING-ITEM + RENAMING

RENAMING-ITEM     ::= ID

PRIM-PROC         ::= ( PROCESS )
                    | run ( EVENT-SET )
                    | chaos ( EVENT-SET )
                    | div
                    | stop
                    | skip
                    | NAMED-PROC

NAMED-PROC        ::= PROC-NAME
                    | PROC-NAME ( TERMS )
```

9.3.1 Analysis of concrete syntax

We have used an online context-free grammar checking tool[1] in order to investigate this grammar's properties.

The first thing to note about our grammar, however, is that it is incomplete: as noted above, various non-terminals correspond to CASL nonterminals. When analysing the CSP-CASL concrete grammar, it would be a mistake to treat the CASL nonterminals as 'black boxes', i.e. as terminals in the CSP-CASL grammar. Thus, our analysis has been performed on a larger grammar than the one shown here, extended with aspects of the CASL concrete syntax as required. As these rules appear in [CoF04b] (and are quite extensive) we omit them here.

We now note some interesting aspects of our analysis.

9.3.1.1 The CSP-CASL concrete grammar is left recursive

Our grammar is left-recursive: see, e.g., the rule for CHOICE-PROC:

```
CHOICE-PROC       ::= SEQ-PROC
                    | CHOICE-PROC [] SEQ-PROC
                    | CHOICE-PROC "|~|" SEQ-PROC
```

This is not a problem, however, because the grammar may be automatically transformed to an equivalent non-left-recursive grammar using standard techniques (see, eg [ALSU06]). For example, the CHOICE-PROC rule becomes:

```
CHOICE-PROC       ::= SEQ-PROC CHOICE-PROC1

CHOICE-PROC1      ::= [] SEQ-PROC CHOICE-PROC1
                    | "|~|" SEQ-PROC CHOICE-PROC1
                    | epsilon
```

[1] http://smlweb.cpsc.ucalgary.ca

where `epsilon` indicates an empty parse. The rules for `CHOICE-PROC` and `CHOICE-PROC1` shown above are then translated very directly into Parsec parsing functions in our implementation (see chapter 10).

Similar comments may be made regarding a number of the rules.

9.3.1.2 The CSP-CASL concrete grammar is not LL(n) for any n

Unlike the CASL concrete grammar, our CSP-CASL grammar is not LL(1), or indeed LL(n) for *any* n. The problem is an unbounded lookahead while parsing a `PROC-ITEM`: after reading a `SIMPLE-ID` for the process name, in order to differentiate between the cases `PROC-DECL` and `PROC-EQ` we need to look ahead for either a colon or an equals sign (respectively). As we may first need to parse a list of parameter sorts or names (respectively), the size of this lookahead is in fact unbounded.

In practice, however, this is again no problem, and is handled beautifully in the implementation by Parsec's `try` combinator: we *attempt* to parse the `PROC-ITEM` as a `PROC-DECL` but if we fail (because, say, we hit an equals when we expected a colon), we 'rewind' and attempt instead to parse it as a `PROC-EQ`.

The alternative would have been to transform our grammar to one in which the choice between `PROC-DECL` and `PROC-EQ` is deferred until the last moment: we would parse the process name and parameter list and *then* make a choice between the two cases depending on whether the next symbol is a colon or an equals.

The advantage of our method is that our grammar remains simple and comprehensible, and our implementation remains a fairly direct translation of that grammar. An apparent but in fact negligible disadvantage is performance: a failed lookahead is wasted processing time, for sure, but in practice all reasonable specifications will have processes with negligibly short parameter lists, and the lookahead will be unnoticeable. If this ever turns out to not be the case, of course, we can transform the grammar to fix the problem.

Other parts of the grammar also involve small lookaheads, also implemented using the `try` combinator, but this is the one unbounded one of which we are aware.

9.3.1.3 Notes on use of online grammar analysis tool

The online tool requires input in a more restricted format than that used in [CoF04b] and this chapter; as such, we have written a small utility in Haskell for parsing a grammar in our format and transforming it to something suitable for the online tool .

For example, the rule for `EVENT`:

```
EVENT            ::= TERM
                   | CHAN-NAME ! TERM
                   | CHAN-NAME ! VAR :: SORT
                   | CHAN-NAME ? VAR :: SORT
```

is translated to:

```
EVENT          ->  CHAN_NAME  sym_bang  TERM
                 | CHAN_NAME  sym_bang  VAR  sym_dcolon  SORT
                 | CHAN_NAME  sym_ques  VAR  sym_dcolon  SORT
```

```
| TERM
.
```

The required translation is in fact fairly simple:

1. Rewrite '`::=`' as '`->`' in each rule.
2. Rewrite various symbols that appear as terminals, e.g. '`::`' becomes '`sym_dcolon`' (the online tool has a very restricted input alphabet).
3. Rewrite nonterminal names, replacing '`-`' (which is rejected by the online tool) with '`_`'.
4. Ensure every rule ends with a '`.`' symbol.

It's fairly clear that these are reasonably simple textual transformations which could be performed (albeit with care) without parsing the grammar — however, doing so was a useful exercise in its own right, and could have been the first stage of writing a tool to perform grammar analysis/transformation ourselves (though in the end this proved unnecessary).

Chapter 10

CSP-CASL Static Semantics

Contents

10.1 Introduction

In this chapter we present the formal static semantics of CSP-CASL, building on those of CASL [BCH+04]. In this first section we introduce the required concepts, semantic objects, and some notation. In section 10.2, we closely examine some areas of particular interest, *viz.* signature structure, local vs global variables, communication alphabets, and the implications of CASL subsorting. Finally, in sections 10.3, 10.4, and 10.5 we present the actual rules, using the Natural Semantics formalism described in chapter 6; here we follow the style of the CASL static semantics, as explored in section 6.3. Each of the syntactic categories in CSP-CASL's abstract syntax (section 9.1) has a rule or set of rules in one of these three sections.

We reuse various concepts, definitions, and rules from [BCH+04]; in general this is made clear at point of usage, but note in particular that many of the rules given in sections 10.3 to 10.5 refer without note to the CASL static semantic rules for `FORMULA`, `ID`, `SIMPLE-ID`, `SORT`, `TERM` and `VAR`.

10.1.1 Overview

The fundamental purpose of the static semantics is to define a transformation from terms in CSP-CASL's abstract syntax to various corresponding semantic objects, and to define conditions and requirements on those transformations and objects. The (input) abstract syntax terms are produced by our parser; the (output) representations of semantic objects may then be processed by tools outside the scope of this thesis, e.g. automated or interactive theorem provers — see section 5.3.

The ultimate product of the static semantics is a pair consisting of a CSP-CASL signature encoding channel and process declarations and a set of semantic objects representing CSP-CASL process equations. The latter are essentially similar to syntactic process equations as seen in the abstract syntax, in that they each bind a parametrised process name to a process term; however, there are important differences.

- A process term's semantic object is decorated at various points with extra information; in particular, syntactic CASL terms and variables are transformed to *fully qualified* versions, tagged with their sorts.
- In some cases we transform undistinguished identifiers into particular distinguished semantic objects by lookup; for example, a `RENAMING-ITEM` is syntactically just a CASL `ID`, which is then transformed to an id and a pair of sort names by performing lookups against the specification's data part (see section 10.5.4).
- There are some other general structural differences between the syntactic and semantic worlds. e.g. syntactic lists often become sets in the corresponding semantic object (e.g. `CHAN-NAMES`); in other cases, e.g. `RENAMING`, the semantic object is also a list.

As well as producing the signature and process equation objects, we perform various static checks as side-effects, including:

- Declaration checks: e.g. in a process declaration, we check that each sort in the process' parameter sort list has been declared in the data part.
- Type checks: e.g. in a reference to a named process, we check that the sorts of the parameters passed to the process correspond to those in the process' declaration.
- Alphabet computation and checking: we compute the *constituent alphabet* of a process term, and perform associated checks at the process equation and process term level.

10.1.2 Notation

We re-use a number of notational conventions from [BCH+04, §1.1]. In particular:

- $FinSet(A)$ is the set of all finite subsets of set A.
- $FinSeq(A)$ is the set of all finite sequences of elements of set A.
- $A \rightharpoonup B$ is the set of partial functions from A to B.
- $Dom(f) \subseteq A$ is the domain of $f : A \rightharpoonup B$.
- $A \to B$ is the set of total functions from A to B.

Some rules require the use of further auxiliary functions for fine-grained manipulation of semantic elements. In particular, we use:

- $fst : A \times B \to A$ and $snd : A \times B \to B$ to project the first and second elements, respectively, of a pair $\in A \times B$.
- $sort : FQTerm \to Sort$ to obtain the overall sort of a CASL fully-qualified term — see section 10.2.4.2; definition in [BCH+04, §2.1.3].
- $sorts : Formula \to FinSet(Sort)$ to obtain the sorts referenced in a CASL formula — see section 10.2.3.1.

Symbol	Meaning
Σ	a CASL signature, $\Sigma = (S, TF, PF, P, \leq)$.
S	a set of CASL sorts, $S \subseteq Sort$.
TF	a set of CASL total functions.
PF	a set of CASL partial functions.
P	a set of CASL predicates.
$\leq$	a CASL subsort relation on a set S of sorts.
Λ	a CSP signature, $\Lambda \in ChanSet \times ProcSet$.
CH	a channel definition set, $CH \in ChanSet = ChanName \rightharpoonup Sort$.
PR	a process definition set, $PR \in ProcSet = ProcName \rightharpoonup ProcProfile$.
s	a sort, $s \in Sort$.
c	a channel name, $c \in ChanName$.
C	a set of channel names, $C \subseteq ChanName$.
(c, s)	a typed channel name, $(c, s) \in TypedChanName = ChanName \times Sort$.
ε	a communication type $\varepsilon \in CommType = ChanName \cup Sort$.
α	a communication alphabet, $\alpha \in CommAlpha = FinSet(CommType)$.
P	a process name, $P \in ProcName$.
$args$	a sequence of process argument sorts, $args \in FinSeq(Sort)$.
p, q	process terms.
i	a CASL identifier, $i \in Id$.
x, y	variable names, $x, y \in Var$.
V	a set of variable names, $V \subseteq Var$.
X	a set of CASL variables, $X \in Variables = Sort \xrightarrow{\text{fin}} FinSet(Var)$.
G	a set of process-global variables, $G \in Variables$.
L	a set of process-local variables, $L \in Variables$.
t	a (CASL) term, $t \in FQTerm$.
es	an event-set, $es \in CommAlpha = FinSet(CommType)$.
e	a communication event.
R	a renaming, $R \in FinSeq(Id \times Sort \times Sort)$; a sequence of renaming items.
r	a renaming item, $r \in Id \times Sort \times Sort$.
φ	a (CASL) formula.

Note that the symbol P is used both for CASL predicates and CSP-CASL process names. In practice, it is always clear from context which is intended; in particular, the predicates part of a CASL signature is in practice projected as Σ_P, as noted in section 10.1.2, in every case except the rule for `RENAMING-ITEM` (section 10.5.4) — and that rule makes no reference to CSP-CASL process names.

Table 10.1: Summary of naming conventions used in this chapter.

Type	Members	Definition
Sort	CASL sort names.	[BCH+04]
Id	CASL identifiers.	[BCH+04]
Var	CASL variable names.	[BCH+04]
FQTerm	CASL fully-qualified terms.	[BCH+04]
Formula	CASL formulae.	[BCH+04]
FunSet	CASL functions.	[BCH+04], 10.2.1
FunProfile	CASL function profiles.	[BCH+04], 10.2.1
FunName	CASL function names.	[BCH+04], 10.2.1
PredSet	CASL predicates.	[BCH+04], 10.2.1
PredProfile	CASL predicate profiles.	[BCH+04], 10.2.1
PredName	CASL predicate names.	[BCH+04], 10.2.1
Variables	Finite maps from sort names to sets of variable names	[BCH+04], 10.2.2
ChanSet	Partial functions from channel names to sorts.	10.2.1
ChanName	Channel names.	10.2.1
ProcSet	Partial functions from process names to process profiles.	10.2.1
ProcName	Process names.	10.2.1
ProcProfile	(Parameter sort sequence, communication alphabet) pairs.	10.2.1
CommAlpha	Sets of communication types.	10.2.1
CommType	Sorts and typed channel names.	10.2.1
TypedChanName	(Channel name, sort name) pairs.	10.2.1

Table 10.2: Summary of types used in this chapter.

- Given a CASL subsorted signature $\Sigma = (S, TF, PF, P, \leq)$ we project its individual components as S_Σ, TF_Σ, PF_Σ, P_Σ, $\leq_\Sigma$.

We adopt a number of naming conventions for instances of semantic objects. These are summarised in table 10.1. That table refers to a number of sets defining the types of those semantic objects (e.g. *ChanName*); table 10.2 summarises these sets, their contents, and where to find their definitions.

The internal structure of identifiers used to identify sorts, variables, channels, renaming-items and processes is insignificant for the static semantics of CSP-CASL specifications. See [BCH+04, §2.6] for the structure of *Sort*, *Var* and *Id*.

Finally, note that in some of the rules we use abbreviated versions of names of syntactic elements in order to save space (e.g. `P-I` for `PROC-ITEM` in the rule for `PROC-ITEMS` in section 10.3.3). We use the full forms corresponding to the abstract syntax presented in section 9.1 wherever possible, but otherwise we note the use of abbreviated forms in the rule's accompanying text.

10.2 Signatures, variables, alphabets and subsorting

In this section we consider some areas of particular interest, *viz.*: the contents and meaning of CASL and CSP-CASL *signatures*; the treatment of *variables*; the contents and treatment of

communication alphabets; and the implications for the CSP-CASL static semantics of CASL subsorting.

10.2.1 CSP-CASL signatures

A CASL subsorted signature [BCH+04, §2.1.1] is a 5-tuple $\Sigma = (S, TF, PF, P, \leq)$ with:

$$\begin{aligned} S \in \mathit{SortSet} &= \mathit{FinSet}(\mathit{Sort}) \\ TF, PF \in \mathit{FunSet} &= \mathit{FunProfile} \rightharpoonup \mathit{FinSet}(\mathit{FunName}) \\ P \in \mathit{PredSet} &= \mathit{PredProfile} \rightharpoonup \mathit{FinSet}(\mathit{PredName}) \\ \mathit{FunProfile} &= \mathit{FinSeq}(\mathit{Sort}) \times \mathit{Sort} \\ \mathit{PredProfile} &= \mathit{FinSeq}(\mathit{Sort}) \end{aligned}$$

- $\leq \in \mathit{Sort} \times \mathit{Sort}$ is a preorder of *subsort embedding* on the set S of sorts.
- *Sort*, *FunProfile*, *FunName* and *PredName* are as defined in [BCH+04, §2.1.1].

For example, consider the following simple CASL fragment:

sorts a, b, c
ops $f : a \times b \to c; g : a \times b \to c;$
$h : a \to b$
pred $p : a \times c$

The corresponding signature is $\Sigma = (S, TF, PF, P)$, with:

$$\begin{aligned} S &= \{a, b, c\} \\ TF &= \{(\langle a, b\rangle, c) \mapsto \{f, g\}, (\langle a\rangle, b) \mapsto \{h\}\} \\ PF &= \emptyset \\ P &= \{\langle a, c\rangle \mapsto \{p\}\} \end{aligned}$$

A CSP-CASL signature (Σ, Λ) is a pair consisting of a CASL signature Σ as described above and a CSP signature $\Lambda = (CH, PR)$, containing information on *channels* and *processes*, specifically: a family of sets of sort names indexed by channel name; and a family of sets of process profiles indexed by process name:

$$\begin{aligned} \mathit{CH} \in \mathit{ChanSet} &= \mathit{ChanName} \to \mathit{Sort} \\ \mathit{PR} \in \mathit{ProcSet} &= \mathit{ProcName} \to \mathit{ProcProfile} \\ \mathit{ProcProfile} &= \mathit{FinSeq}(\mathit{Sort}) \times \mathit{CommAlpha} \\ \alpha \in \mathit{CommAlpha} &= \mathit{FinSet}(\mathit{CommType}) \\ \varepsilon \in \mathit{CommType} &= \mathit{Sort} \cup \mathit{TypedChanName} \\ (c, s) \in \mathit{TypedChanName} &= \mathit{ChanName} \times \mathit{Sort} \\ \mathit{ChanName} = \mathit{ProcName} &= \mathit{Id} \end{aligned}$$

Thus: a channel has a name and a sort; a process has a name and a profile, which consists of a (possibly empty) finite sequence of process parameter sorts, and a set describing its alphabet of possible comunications.

Note an important difference here, between CASL signatures and CSP signatures: in the former, functions are represented as maps from (function) profiles to (function) names; conversely, in a

CSP signature, channels and processess are represented as maps from (channel/process) names to (channel/process) profiles. The distinction arises because CASL allows *overloading* of function names, whereas in CSP-CASL neither channel names nor process names are overloaded (overloading of the former is unnecessary in the presence of CASL subsorting); this simplifies things for us: the (frequent) act of lookup by name is easier if we map *from* names, rather than *to* names.

10.2.2 Variables

[BCH+04, §III:2.1.3] defines $Variables = Sort \xrightarrow{\text{fin}} FinSet(Var)$, i.e. sets of variable names indexed by sort name (note that this structure allows overloading of variable names). A CASL term is statically checked in a context including such a *Variables* set — it is precisely these variables which may occur within that term.

Now, as discussed in section 8.1.7, in CSP-CASL we carefully separate *process-global* variables (declared on the left hand side of a process equation) and *process-local* variables (locally scoped variables arising within a process term). In particular, in the sequential composition operator on process terms, the second process term starts with an empty set of process-local variables.

In our static semantics, then, we write a set of process-global variables as G and a set of process-local variables as L. (An undistinguished set of variables is just written X, following [BCH+04].) The two are unified only when static checks on a CASL term are required, i.e. in the rules for `NAMED-PROC` and `PREFIX-PROC` (although the latter could have been deferred to the appropriate `EVENT` rule equally well).

Note that in rules which build or extend variable sets (e.g. `INTPRE-PROC`), we use the operator $+$ defined in [BCH+04, §III:2.1.3], with type signature:

$$+ : Variables \times (Var \times Sort) \to Variables$$

10.2.3 Communication alphabets

An important aspect of the static analysis of a CSP-CASL specification is the analysis of *communication alphabets*, where:

- A process declaration includes a declaration of the alphabet of *permitted communications* of that process (see section 8.1.2).
- A process term (recursively) produces a *constituent alphabet*, describing the communications occurring in that process term. Items are introduced into this constituent alphabet by particular process operators as discussed below.

Formally, a communication alphabet α is a member of $CommAlpha$:

$$\begin{aligned} \alpha \in CommAlpha &= FinSet(CommType) \\ CommType &= Sort \cup TypedChanName \\ (c, s) \in TypedChanName &= ChanName \times Sort \end{aligned}$$

Thus, a communication alphabet is a finite set of $CommType$ elements, each of which is either a sort name or a (channel name, sort name) pair. Note that this concept covers not only constituent and permitted alphabets, but also *event sets*, as used by various of the process operators.

Note the use of *typed* channel names in communication alphabets; at first this seems unnecessary, because a channel's declaration already associates it with a sort, which can thus be inferred by lookup. CASL subsorting complicates matters, however: in some cases we need to lift CASL subsorts to CSP-CASL communications; we may, e.g., send an s_1-sorted value over a s_2-sorted channel provided $s_1 \leq s_2$. As such, it is necessary to track the actual intended sort of a given channel event, rather than simply inferring it from the channel's declaration. This is discussed further in section 10.2.4.1.

There are two major semantic requirements involving communication alphabets:

1. In a process equation $P = p$, the constituent alphabet α_p of the process term p on the right hand side must be 'included in' the permitted communication alphabet α_P of the process P named on the left hand side. This requirement arises from the formalisation of CSP-CASL's logic as an institution [MR07].
2. In an alphabetised parallel process $p \,[[es_1||es_2]]\, q$, the event sets es_1 and es_2 must be 'included in' the constituent alphabets α_p and α_q of the process terms p and q, respectively. This is a standard requirement inherited from CSP [Ros98].

We formally define 'included in' in section 10.2.4.1.

10.2.3.1 Computation of process term constituent alpabets

The constituent alphabet of a process term is computed recursively over its structure, by computing the constituent alphabets of its components, and combining/modifying them according to the process term's type. At bottom, communication types (sorts and typed channel names) are introduced into the alphabet by process terms which engage in communications, or by hide/rename/conditional processes.

The rules are mainly simple, and are summarised in table 10.3. They largely build directly on the rules for CSP terms presented in [MR07, §3.2]. The rules are implemented in the natural semantics rules later in this chapter, in particular the rules in section 10.4.

In section 10.1.2 we introduced the function $sorts : Formula \to FinSet(Sort)$, to compute the sorts occurring in a CASL formula. A CASL formula is a formula in First Order Logic; as such we define $sorts()$ recursively on a formula's structure, as shown in table 10.4. This definition is, strictly, incomplete, in that CASL formulae have a richer syntax which is encoded in [CoF04a] to the form presented here.

10.2.4 Implications of subsorting

CASL subsorting introduces some interesting challenges to the CSP-CASL context: comparison of communication alphabets needs to be closed under subsort, and CASL terms in CSP-CASL processes may need to be cast, which requires careful attention to subsorting.

10.2.4.1 Closure under subsorting of communication alphabets

In section 10.2.3 we introduced two major semantic requirements on communication alphabets, but deferred discussion of what we mean by one being 'included in' another; we shall now

Process type	**Process Term** p	**Constituent alphabet,** α_p	**Notes**
Named process	$P(t_1, \ldots, t_n)$	Permitted alphabet of P	[1]
Skip	$SKIP$	$\emptyset$	
Stop	$STOP$	$\emptyset$	
Div	DIV	$\emptyset$	
Run	$RUN(es)$	es	
Chaos	$CHAOS(es)$	es	
Term prefix	$t \rightarrow p$	$\alpha_p \cup \{sort(t)\}$	
Channel send	$c!t \rightarrow p$	$\alpha_p \cup \{sort(t), (c, sort(t))\}$	[1]
Channel nondeterministic send	$c!x :: s \rightarrow p$	$\alpha_p \cup \{s, (c, s)\}$	[2]
Channel receive	$c?x :: s \rightarrow p$	$\alpha_p \cup \{s, (c, s)\}$	[2]
Internal prefix choice	$\sqcap x :: s \rightarrow p$	$\alpha_p \cup \{s\}$	[2]
External prefix choice	$\Box x :: s \rightarrow p$	$\alpha_p \cup \{s\}$	[2]
Sequential composition	$p \ ; \ q$	$\alpha_p \cup \alpha_q$	
Internal choice	$p \sqcap q$	$\alpha_p \cup \alpha_q$	
External choice	$p \Box q$	$\alpha_p \cup \alpha_q$	
Interleaving	$p \mid\mid\mid q$	$\alpha_p \cup \alpha_q$	
Synchronous	$p \parallel q$	$\alpha_p \cup \alpha_q$	
Generalised parallel	$p [\![es]\!] q$	$\alpha_p \cup \alpha_q \cup es$	
Alphabetised parallel	$p [\![es_1 \| \| es_2]\!] q$	$\alpha_p \cup \alpha_q$	[3]
Hiding	$p \setminus es$	$\alpha_p \cup es$	[4]
Renaming	$p[r_1, \ldots, r_n]$	$\alpha_p \cup \bigcup_{i=1}^{n} \{s_{i,1}, s_{i,2}\}$ with $r_i : s_{i,1} \times s_{i,2}$ (predicate) or $r_i : s_{i,1} \rightarrow s_{i,2}$ (functional)	[5]
Conditional	if φ then p else q	$\alpha_p \cup \alpha_q \cup sorts(\varphi)$	[6]

Notes

1. See discussion of terms, sorts, and subsorting in section 10.2.4.2.
2. Introduces a new binding on x to the term's local variables.
3. We require $es_1 \subseteq \alpha_p \wedge es_2 \subseteq \alpha_q$.
4. We add es to the constituent alphabet despite the fact that we are hiding its communications, for reasons discussed in [MR07]: without it, CSP-CASL cannot form an institution. Thus, the constituent alphabet is *not* (as it first appears) simply the alphabet of communications in which the process term attempts to engage.
   Note also that we *do not* require $es \in \alpha_p$ as, e.g., we allow $STOP \setminus Nat$.
5. We do not require $\{s_{i,1}, s_{i,2}\} \subseteq \alpha_p$ as, e.g., we allow $STOP[r]$ with $r : Nat \rightarrow Int$.
6. See table 10.4.

Table 10.3: Computation of process term constituent alphabets

φ	$sorts(\varphi)$	
ff	$\emptyset$	falsehood
$P(t_1, \ldots, t_n)$	$\bigcup_{i=1}^{n} sort(t_i)$	predicate
$t_1 = t_2$	$\{sort(t_1)\}$	equality
$t_1 \stackrel{e}{=} t_2$	$\{sort(t_1)\}$	existential equality
$def\, t$	$\{sort(t)\}$	definedness
$\neg t$	$\{sort(t)\}$	negation
$t_1 \wedge t_2$	$\{sort(t_1), sort(t_2)\}$	conjunction
$t_1 \Rightarrow t_2$	$\{sort(t_1), sort(t_2)\}$	implication
$\forall x : s \bullet \varphi$	$\{s\} \cup sorts(\varphi)$	universal qualifier

Table 10.4: Computation of CASL formula sorts

define this properly; the notion is slightly complicated by the need to take account of *closure under* CASL *subsort*.

Consider the process equation $P = p$, and suppose $\alpha_P = \{s_1\}$, and $\alpha_p = \{s_2\}$ for $s_1, s_2 \in Sorts$. Suppose further that $s_2 \leq s_1$, i.e. s_2 is a CASL subsort of s_1. In this case, we would like to say that α_p is indeed 'included in' α_P, since semantically that process equation is perfectly acceptable: anything p attempts to communicate is allowed by P, because $s_2 \leq s_1$. However, a simple check for set inclusion fools us: $\alpha_p \not\subseteq \alpha_P$.

We must, then, lift the CASL subsort relation to communication alphabets. Specifically, we define the *subsort closure* operator $\downarrow : CommAlpha \to CommAlpha$ as follows:

$$\downarrow(\alpha) := \bigcup_{x \in \alpha} \begin{cases} \{y \mid y \leq x\} & x \in Sort \\ \{(c, y) \mid y \leq z\} \cup \{ y \mid y \leq z\} & x = (c, z) \in TypedChanName \end{cases}$$

That is, for any sort name in the alphabet, we include each of its subsorts; and for any typed channel name in the alphabet, we include a corresponding typed channel name for each of its sort's subsorts, *and* we include each of its sort's subsorts.

Then, in the above example, $\alpha_p \subseteq \downarrow(\alpha_P)$ gives us the right answer (i.e. 'true').

Remark: an alternative approach would be to close *CommAlpha* under subsort by definition — then ordinary $\subseteq$ would suffice, but it would be necessary to recompute subsort closure every time we modify a *CommAlpha*. As most of the semantic rules for process terms involve union of communication alphabets, this would be frequent. However, we only *need* the subsort closure when comparing two communication alphabets which, as noted above, happens rarely. Thus, in order to both simplify the definition of *CommAlpha*, and with an eye on performance of the implementation, we choose to enact subsort closure via the $\downarrow$ operator described above, rather than directly in the *CommAlpha* type.

10.2.4.2 Subsorting and terms

In the presence of subsorting, type checking of CASL terms which occur in CSP-CASL process terms requires some attention. Before discussing the CSP-CASL-specific aspects, let us examine how the sorts of CASL terms are treated in the presence of subsorting.

The static semantic rules for TERM given in [BCH+04, §III:2.5.4] all yield a *fully qualified term*

$\in FQTerm$, whose overall sort may then be computed [BCH+04, §III:2.1.3]. Thus, given a syntactic TERM and appropriate context, we may use existing CASL machinery to obtain the corresponding fully qualified term t, and its sort $sort(t)$.

Now, suppose we have two CASL sorts s and s', with $s \leq s'$. In such a setting, there are two kinds of *cast*[1] one might perform on a term t [BCH+04, §III:3.3.2]:

- **upcast** — if t has sort s then the upcast $t : s'$ has sort s', and expands to an application of the implicit operation embedding s in s';
- **downcast** — if t has sort s', then the downcast $t \; as \; s$ has sort s and expands to an application of the implicit operation projecting s' to s.

Clearly, an embedding produced by an upcast will always be total. However, a projection produced by a downcast may (and usually will) be partial.

Consider, for example, the CASL specification CASTS in figure 10.1, which extends INT from the CASL standard libraries [RMS04]. In particular, this imports the sorts *Nat* and *Int*, with *Nat* a subsort of *Int*. The four axioms then illustrate the various casting possibilities, as follows:

- The first axiom illustrates upcast: a, of sort *Nat*, has the value $1 : Nat$ and is upcast to an *Int* (which is bound to the name x). Since *Nat* is a subsort of *Int*, this is well-sorted and always defined.
- The second axiom illustrates downcast: y, of sort *Int*, has the value $1 : Int$ and is downcast to a *Nat* (bound to the name b). Clearly this projection is also well-sorted and defined.
- The third axiom also illustrates a downcast: z, of sort *Int*, has the value $-1 : Int$, and is downcast to a *Nat* (bound to the name c). Clearly, while the term $z \; as \; Nat$ is well-sorted, its value is undefined — however, it is important to note that *this is not prevented by* CASL*'s static semantics.* This could then be checked for elsewhere in the specification using CASL's *definedness assertion* operator, *def* [BM04, §4.2]. For example, we might write an axiom which begins $def \; c \Rightarrow \ldots$
- The final axiom illustrates disallowed casts. Here we have w, of sort *Other*, and we attempt to upcast/downcast it to *Nat* — both of which attempts must necessarily fail because *Nat* and *Other* are in no subsort relationship with each other. Such cast attempts, then, are *not* well-sorted in CASL's static semantics.

Now, in order to properly treat CASL terms occurring in CSP-CASL process terms, and bearing in mind the above, it proves useful to define the function $cast : FQTerm \times Sort \rightarrow FQTerm$ which, given a fully qualified term t and a sort s attempts to cast t to s in the following manner:

$$cast(t, s) := \begin{cases} t & \text{if } sort(t) = s \\ t : s & \text{if } sort(t) < s \text{ (upcast/embedding)} \\ t \; as \; s & \text{if } s < sort(t) \text{ (downcast/projection)} \\ error & \text{otherwise} \end{cases}$$

Let us now consider CASL terms as they occur in CSP-CASL process terms. There are three cases:

1. parameters in a reference to a *named* process, $P(t_1, \ldots, t_n)$

[1] Here we slightly abuse existing terminology for our own purposes: [BCH+04] just uses 'cast' where we use 'downcast', and has no explicit name for what we call 'upcast'.

```
from BASIC/NUMBERS get INT
spec CASTS = INT
then sort Other
     ops  a, b, c, d, e : Nat;
          x, y, z : Int;
          w : Other
     • a = 1 ∧ x = a : Int                          %( upcast )%
     • y = 1 ∧ b = y as Nat                         %( well-sorted downcast )%
     • z = − 1 ∧ c = z as Nat                       %( well-sorted downcast, value undefined )%
     • d = w : Nat ∧ e = w as Nat                   %( disallowed casts )%
end
```

Figure 10.1: Upcasts, downcasts, and disallowed casts in CASL

2. in a *term prefix* process, $t \rightarrow p$
3. in a *channel send* process, $c!t \rightarrow p$

In each case, the term's sort contributes to the constituent alphabet of the process term, as discussed in section 10.2.3; in the first and third cases, however, there are additional requirements:

- The sort of each parameter in a named process reference must be castable to the corresponding sort in the process declaration. For example, given the process declaration $P(S, T)$ and the named process reference $P(a, b)$, we require that $cast(a, S) \neq error$ and $cast(b, T) \neq error$.
- Similarly, the sort of a term in a channel send must be castable to the sort of the channel. For example, in the process term $c!t \rightarrow p$, if c has the sort s, we require that $cast(t, s) \neq error$.

The rules for NAMED-PROC (section 10.4.2) and channel send (section 10.5.2) formalise these requirements.

10.3 Top-level elements

This section contains the static semantics rules for the 'top level' elements of a CSP-CASL specification — i.e. those whose abstract syntax is defined in section 9.1.1.

10.3.1 CSP-CASL basic specifications

A CSP-CASL basic specification CSP-BASIC-SPEC is a sequence of channel declarations and a sequence of process declarations and process equations. It determines a CSP signature and a set of CSP-CASL process equations.

```
CSP-BASIC-SPEC ::= csp-basic-spec CHAN-DECLS PROC-ITEMS
```

$$\boxed{\Sigma \vdash \texttt{CSP-BASIC-SPEC} \rhd \Lambda, \Psi}$$

Given a CASL signature Σ representing the data part of a CSP-CASL specification, and a syntactic CSP-BASIC-SPEC representing the process part, we compute a CSP signature Λ and a set Ψ of semantic objects representing CSP-CASL process equations. We require that the CASL signature has local top elements, as described in chapter 11, deferring the definition of *hasLocalTops* until then.

$$\frac{\mathit{hasLocalTops}(\Sigma) \qquad \Sigma \vdash \texttt{CHAN-DECLS} \rhd CH \qquad \Sigma, CH \vdash \texttt{PROC-ITEMS} \rhd \Lambda, \Psi}{\Sigma \vdash \texttt{csp-basic-spec CHAN-DECLS PROC-ITEMS} \rhd \Lambda, \Psi}$$

10.3.2 Channel declarations

A CHAN-DECLS is a sequence of channel declarations, each of which declares a number of channels as having a particular CASL sort. Multiple declarations of the same channel name must map to the same CASL sort.

```
CHAN-DECLS ::= chan-decls CHAN-DECL*
```

$$\boxed{\Sigma \vdash \texttt{CHAN-DECLS} \rhd CH}$$

Given a CASL signature Σ and a CHAN-DECLS sequence of channel declarations, we compute a *ChanSet* map CH from channel names to sort names. CH is built linearly, starting with the empty set and adding the product of each successive CHAN-DECL rule.

The requirement that multiple declarations of the same channel name must map to the same CASL sort is encoded in the rules for CHAN-DECL and CHAN-NAMES, below. A partially built CH is passed to each CHAN-DECL rule as context in order to enable this.

$$\frac{\begin{array}{c}\Sigma, \emptyset \vdash \texttt{CHAN-DECL}_1 \rhd CH_1 \\ \cdots \\ \Sigma, CH_1 \cup \cdots \cup CH_{n-1} \vdash \texttt{CHAN-DECL}_n \rhd CH_n\end{array}}{\Sigma \vdash \texttt{chan-decls CHAN-DECL}_1 \cdots \texttt{CHAN-DECL}_n \rhd CH_n}$$

A CHAN-DECL is a set of channel names and a channel sort, and declares that each of the named channels has the specified sort.

```
CHAN-DECL      ::= chan-decl CHAN-NAMES CHAN-TYPE
```

$$\boxed{\Sigma, CH \vdash \texttt{CHAN-DECL} \rhd CH'}$$

Given a CASL signature Σ, a *ChanMap* CH, and a syntactic CHAN-DECLS containing a sequence of channel names and a sort name, we compute a channel map CH' containing mappings from each of the channel names to the sort name.

$$\frac{\Sigma \vdash \texttt{CHAN-TYPE} \rhd s \qquad \Sigma, CH, s \vdash \texttt{CHAN-NAMES} \rhd \{c_1, \cdots, c_n\}}{\Sigma, CH \vdash \texttt{chan-decl CHAN-NAMES CHAN-TYPE} \rhd \{c_1 \mapsto s, \cdots, c_n \mapsto s\}}$$

A CHAN-TYPE is just a CASL sort.

```
CHAN-TYPE       ::= chan-type SORT
```

$$\boxed{\Sigma \vdash \texttt{CHAN-TYPE} \rhd s}$$

Given a CASL signature Σ and a syntactic CHAN-TYPE, we (trivially) compute a CASL sort. We require that the sort has been previously declared, i.e. it is known in Σ.

$$\frac{\vdash \texttt{SORT} \rhd s \qquad s \in \Sigma_S}{\Sigma \vdash \texttt{chan-type SORT} \rhd s}$$

A CHAN-NAMES is a sequence of channel names, which are just SIMPLE-IDs. A channel may not have the same name as a previously declared CASL sort.

```
CHAN-NAMES      ::= chan-names CHAN-NAME+
CHAN-NAME       ::= chan-name SIMPLE-ID
```

$$\boxed{\Sigma, CH, s \vdash \texttt{CHAN-NAMES} \rhd C}$$

Given a CASL signature Σ, a *ChanMap* CH, a CASL sort s, and a syntactic CHAN-NAMES sequence of channel names, we compute a set of channel names. This rule rejects repeated declarations of the same channel name with different sorts (see section 8.3.3) by looking for an existing entry for each channel name in the *ChanMap* — if one is found, we require that it maps to the sort name currently under consideration. Note that our implementation raises a warning in the case of multiple declarations with the same sort (see section 12.4).

$$\frac{\begin{array}{cc}\Sigma \vdash \texttt{CHAN-NAME}_1 \rhd c_1 & c_1 \in Dom(CH) \Rightarrow CH(c_1) = s \\ \cdots & \cdots \\ \Sigma \vdash \texttt{CHAN-NAME}_n \rhd c_n & c_n \in Dom(CH) \Rightarrow CH(c_n) = s\end{array}}{\Sigma, CH, s \vdash \texttt{chan-names CHAN-NAME}_1 \cdots \texttt{CHAN-NAME}_n \rhd \{c_1, \cdots, c_n\}}$$

$$\boxed{\Sigma \vdash \texttt{CHAN-NAME} \rhd c}$$

Given a CASL signature and a syntactic CHAN-NAME, we (trivially) compute a channel name. This rule rejects channel names already in use as sort names, in order that we may reliably distinguish between the two when considering communication alphabets (see 8.3).

$$\frac{\vdash \texttt{SIMPLE-ID} \rhd c \qquad c \notin S_\Sigma}{\Sigma \vdash \texttt{chan-name SIMPLE-ID} \rhd c}$$

10.3.3 Process items

A PROC-ITEMS is a sequence of process declarations and process equations.

```
PROC-ITEMS      ::= proc-items PROC-ITEM+
```

$$\boxed{\Sigma, CH \vdash \texttt{PROC-ITEMS} \rhd \Lambda, \Psi}$$

Given a CASL signature Σ, a *ChanSet* CH, and a syntactic PROC-ITEMS sequence of process declarations and process equations, we compute a CSP signature Λ and a set Ψ of process equations.

Each member of the PROC-ITEMS sequence is considered in turn: each PROC-ITEM rule yields a *ProcSet* PR_i and a set Ψ_i of process equations, one of which will be the empty set depending on whether the PROC-ITEM contains a process declaration or process equation. Linear visibility of process declarations is encoded here, by the linear construction of the intermediate *ProcSets* $PR_1, \ldots, PR_{n-1}$ and the presentation of their cumulative unions to the successive PROC-ITEM rules as context.

Note that in this rule we use the abbreviated form P-I for PROC-ITEM.

$$\frac{\begin{array}{c}\Sigma, (CH, \emptyset) \vdash \texttt{P-I}_1 \rhd PR_1, \Psi_1 \\ \cdots \\ \Sigma, (CH, PR_1 \cup \cdots \cup PR_{n-1}) \vdash \texttt{P-I}_n \rhd PR_n, \Psi_n\end{array}}{\Sigma, CH \vdash \texttt{proc-items P-I}_1 \cdots \texttt{P-I}_n \rhd (CH, PR_1 \cup \cdots \cup PR_n), \Psi_1 \cup \cdots \cup \Psi_n}$$

A PROC-ITEM is either a process declaration or a process equation.

```
PROC-ITEM       ::= PROC-DECL | PROC-EQ
```

$$\boxed{\Sigma, \Lambda \vdash \texttt{PROC-ITEM} \rhd PR, \Psi}$$

Given a CASL signature Σ, a CSP signature Λ, and a syntactic PROC-ITEM, we compute a *ProcSet* PR and a set Ψ of process equations.

We have two *qualified rules* devolving PROC-ITEM to the two possibilities. If the PROC-ITEM is a PROC-DECL, PR contains one process declaration object, and Ψ is the empty set; if it is a PROC-EQ, PR is the empty set and Ψ contains one process equation object.

$$\frac{\Sigma, \Lambda \vdash \texttt{PROC-DECL} \rhd PR}{\Sigma, \Lambda \vdash \texttt{PROC-DECL qua PROC-ITEM} \rhd PR, \emptyset}$$

$$\frac{\Sigma, \Lambda \vdash \texttt{PROC-EQ} \rhd \Psi}{\Sigma, \Lambda \vdash \texttt{PROC-EQ qua PROC-ITEM} \rhd \emptyset, \Psi}$$

10.3.4 Process declarations

A PROC-DECL is a process name, a sequence of parameter sorts, and a sequence of permitted alphabet items.

```
PROC-DECL       ::= proc-decl PROC-NAME PROC-ARGS PROC-ALPHA
```

$$\boxed{\Sigma, \Lambda \vdash \texttt{PROC-DECL} \rhd PR}$$

Given a CASL signature Σ, a CSP signature Λ and a syntactic PROC-DECL process declaration, we compute a single-element *ProcSet* PR mapping the declaration's process name to its parameter sort list and permitted alphabet.

We require that the process name is not already known, i.e. multiple declarations of the same process are disallowed.

$$\frac{\vdash \texttt{PROC-NAME} \rhd P \quad P \notin Dom(PR) \quad \Sigma \vdash \texttt{PROC-ARGS} \rhd args \quad \Sigma, CH \vdash \texttt{PROC-ALPHA} \rhd \alpha}{\Sigma, (CH, PR) \vdash \texttt{proc-decl PROC-NAME PROC-ARGS PROC-ALPHA} \rhd \{P \mapsto (args, \alpha)\}}$$

A PROC-NAME is just a CASL SIMPLE-ID

```
PROC-NAME       ::= proc-name SIMPLE-ID
```

$$\boxed{\vdash \texttt{PROC-NAME} \rhd P}$$

Given a syntactic PROC-NAME, we (trivially) compute a process name.

$$\frac{\vdash \texttt{SIMPLE-ID} \rhd P}{\vdash \texttt{proc-name SIMPLE-ID} \rhd P}$$

A PROC-ARGS is a sequence of CASL sort names, defining the parameter sorts expected in a reference to a named process.

```
PROC-ARGS       ::= proc-args SORT*
```

$$\boxed{\Sigma \vdash \texttt{PROC-ARGS} \rhd args}$$

Given a CASL signature Σ and a syntactic PROC-ARGS sequence of CASL sorts, we (trivially) compute a sequence *args* of CASL sorts. We require that each sort has been previously declared, i.e. it is known in Σ.

$$\frac{\begin{array}{cc}\vdash \texttt{SORT}_1 \rhd s_1 & s_1 \in \Sigma_S \\ \cdots & \cdots \\ \vdash \texttt{SORT}_n \rhd s_n & s_n \in \Sigma_S\end{array}}{\Sigma \vdash \texttt{proc-args SORT}_1 \cdots \texttt{SORT}_n \rhd \langle s_1, \cdots, s_n \rangle}$$

A PROC-ALPHA, a process' permitted communication alphabet, is a sequence of COMM-TYPEs; these are undistinguished identifiers, to be differentiated into a set of sort names and channel names.

```
PROC-ALPHA      ::= proc-alphabet COMM-TYPE*
```

$$\boxed{\Sigma, CH \vdash \texttt{PROC-ALPHA} \rhd \alpha}$$

Given a CASL signature Σ, a *ChanMap* *CH*, and a syntactic PROC-ALPHA, we produce a communication alphabet α describing the permitted communications of the process. Each COMM-TYPE produces one corresponding *CommType* object; the distinction between sorts and typed channel names occurs in the COMM-TYPE rule.

Note that the COMM-TYPE list may contain duplicate entries (though obviously the set produced has no such duplication), and that the alphabet will be closed under CASL subsort when compared with other alphabets (see section 10.2.4.1).

Note the similarity between this rule and the one for EVENT-SET (section 10.5.1).

$$\frac{\Sigma, CH \vdash \texttt{COMM-TYPE}_1 \rhd \varepsilon_1 \quad \cdots \quad \Sigma, CH \vdash \texttt{COMM-TYPE}_1 \rhd \varepsilon_n}{\Sigma, CH \vdash \texttt{proc-alphabet COMM-TYPE}_1 \cdots \texttt{COMM-TYPE}_n \rhd \{\varepsilon_1, \ldots, \varepsilon_n\}}$$

10.3.5 Process equations

A PROC-EQ process equation binds a parametrised process name to a process term.

```
PROC-EQ        ::= proc-eq PARM-PROCNAME PROCESS
```

$$\boxed{\Sigma, \Lambda \vdash \texttt{PROC-EQ} \rhd \Psi}$$

Given a CASL signature Σ, a CSP signature Λ and a syntactic PROC-EQ, we compute a set Ψ containing one semantic object representing a process equation. First, we compute a process name P and process-global variable set G; given this, we compute a process term p and its constituent alphabet α_p; given this, we check that the subsort closure $\downarrow(\alpha_p)$ of the process term's constituent alphabet is a subset of the subsort closure $\downarrow(snd(PR(P)))$ of the process' permitted alphabet. (In an environment containing $P \in ProcName$ and $PR \in ProcSet$, $snd(PR(P))$ is the permitted alphabet from the process declaration of P.) See sections 10.2.3 and 10.2.4.1 for further discussion of communication alphabet checking. Note that the empty set is passed to the PROCESS rule as its local variable set (see section 10.2.2).

Note that in this rule we use the abbreviated forms P-PN and P for PARM-PROCNAME and PROCESS respectively.

$$\frac{PR \vdash \texttt{P-PN} \rhd P, G \qquad \Sigma, (CH, PR), G, \emptyset \vdash \texttt{P} \rhd p, \alpha_p \qquad \downarrow(\alpha_p) \subseteq \downarrow(snd(PR(P)))}{\Sigma, (CH, PR) \vdash \texttt{proc-eq P-PN P} \rhd \{P(G) = p\}}$$

A PARM-PROCNAME is a process name and a (possibly empty) sequence of variable names.

```
PARM-PROCNAME  ::= parm-procname PROC-NAME PROC-VARS
```

$$\boxed{PR \vdash \texttt{PARM-PROCNAME} \rhd P, X}$$

Given a *ProcSet* PR and a syntactic PARM-PROCNAME, we compute a process name P and a variable set X which will become the process' global variable set. We require that the process name has been previously declared, i.e. it is known in PR.

$$\frac{\vdash \texttt{PROC-NAME} \rhd P \qquad P \in Dom(PR) \qquad PR, P \vdash \texttt{PROC-VARS} \rhd X}{PR \vdash \texttt{parm-procname PROC-NAME PROC-VARS} \rhd P, X}$$

A PROC-VARS is a sequence of CASL variable names occurring in a parametrised process name.

```
PROC-VARS      ::= proc-vars PROC-VAR*
```

$$\boxed{PR, P \vdash \texttt{PROC-VARS} \rhd X}$$

Given a *ProcSet* PR, a process name P, and a sequence of syntactic PROC-VARs, we compute a variable set X mapping sorts to the variable names as specified by the named process' profile.

In an environment containing $P \in ProcName$ and $PR \in ProcSet$, $fst(PR(P))$ is the parameter sort list from the process declaration of P (if it has been declared). This is used here both

to check that the length of the variable list is appropriate, and to determine the sort for each variable in the resultant variable set.

The rule encodes the following requirements: the number of variable names being declared for the process name must equal the declared number of sorts in the process' parameter list; there must be no repetition of variable names (see section 8.3.8). This last requirement is encoded in the way this rule linearly builds a variable set for the parameter names, passing the growing set as context to each instance of the PROC-VAR rule (see below), which then rejects any repeated names.

$$\frac{\begin{array}{c}\emptyset, (fst(PR(P)))_1 \vdash \texttt{PROC-VAR}_1 \rhd X_1 \\ \cdots \\ n = \#(fst(PR(P))) \qquad X_{n-1}, (fst(PR(P)))_n \vdash \texttt{PROC-VAR}_n \rhd X_n\end{array}}{PR, P \vdash \texttt{proc-vars PROC-VAR}_1 \cdots \texttt{PROC-VAR}_n \rhd X_n}$$

A PROC-VAR just a CASL variable name, which we require to have not been declared already within the local context.

```
PROC-VAR        ::= proc-var VAR
```

$$\boxed{X, s \vdash \texttt{PROC-VAR} \rhd X'}$$

Given a variable set X, a CASL sort s and a syntactic PROC-VAR, we compute an extension X' to X for the specified variable name and sort. We disallow multiple instances of the same variable name to occur in a parametrised process name for reasons discussed in section 8.3.8.

$$\frac{\vdash \texttt{VAR} \rhd x \qquad \neg \exists s' \in Dom(X) \bullet x \in X(s')}{X, s \vdash \texttt{proc-var VAR} \rhd X + (x, s)}$$

10.4 Process terms

This section contains the static semantics rules for the various processes and process operators which can build a CSP-CASL process term — i.e. those whose abstract syntax is defined in section 9.1.2.

All of the rules in this section have the same form:

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{PROCESS} \rhd p, \alpha}$$

Given a CASL signature Σ, a CSP signature Λ, a set G of process-global variables, a set L of process-local variables, and a syntactic PROCESS, we compute a process term p and its constituent alphabet α.

The rules differ from each other in the following manners:

- in the syntactic category under consideration (obviously);
- in the context-specific side-requirements arising;
- in the construction of α;
- in the treatment of local variables — some rules introduce new ones, and the sequential composition rule reinitialises them in its second half, for example.

10.4.1 PROCESS

A CSP-CASL process term PROCESS may be a named process reference, a basic process, or an application of one of the various process operators.

```
PROCESS        ::= NAMED-PROC | BASIC-PROC | PREFIX-PROC
                 | INTPRE-PROC | EXTPRE-PROC | SEQ-PROC
                 | INTCHOICE-PROC | EXTCHOICE-PROC | INTPAR-PROC
                 | SYNPAR-PROC | GENPAR-PROC | ALPHAPAR-PROC
                 | HIDING-PROC | RENAMING-PROC | CONDITION-PROC
```

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{PROCESS} \rhd p, \alpha}$$

PROCESS is the disjoint union of fifteen more specialised syntactic categories, i.e. NAMED-PROC, BASIC-PROC, etc.; as such, and as described in section 6.3.2.3, we require fifteen *qualified* rules, one per sub-category, each devolving PROCESS to one of the more specialised possibilities. Following the convention of [BCH+04], we show just one rule, and note that the rest are entirely similar and may be trivially inferred.

$$\frac{\Sigma, \Lambda, G, L \vdash \texttt{NAMED-PROC} \rhd p, \alpha_p}{\Sigma, \Lambda, G, L \vdash \texttt{NAMED-PROC qua PROCESS} \rhd p, \alpha_p}$$

10.4.2 NAMED-PROC

A NAMED-PROC is a reference to a process by name, and should include appropriately-sorted terms passed as the parameters required by that process.

```
NAMED-PROC     ::= named-proc PROC-NAME TERM*
```

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{NAMED-PROC} \rhd p, \alpha}$$

As noted in section 10.3.5, in an environment containing $P \in \mathit{ProcName}$ and $PR \in \mathit{ProcSet}$, $\mathit{fst}(PR(P))$ is the parameter sort list from the process declaration of P (if it has been declared).

Then, for the sake of brevity in the rule which follows, let us define $\kappa : \mathit{FQTerm} \times \mathit{Int} \to \mathit{FQTerm}$ as:

$$\kappa(t, n) := \mathit{cast}(t, \mathit{fst}(PR(P))_n))$$

Thus, κ looks up the nth parameter sort in the named process' parameter sort list, and attempts to cast the fully qualified term t to that sort (see section 10.2.4.2).

The rule encodes the following requirements: the process name must be a valid identifier; the process name must have already been declared (otherwise the $PR(P)$ lookup will fail); the number of parameters passed in this named process reference must equal the declared number of sorts in the process' parameter list; each parameter must be a well-formed CASL term given the current environment; each parameter term's overall sort must be compatible with the corresponding sort in the process declaration (see section 10.2.4.2).

We present $G \cup L$, as context to the TERM rule: each CASL term here may refer to any process-global and process-local variables visible at this point.

The constituent alphabet α computed by this rule is just the permitted alphabet of the named process, as defined in its process declaration.

Note that in this rule we use the abbreviated forms P-N and T for PROC-NAME and TERM respectively.

$$\frac{\begin{array}{ccc} & (\Sigma, G \cup L) \vdash \mathtt{T}_1 \rhd t_1 & \kappa(t_1, 1) \neq \mathit{error} \\ \vdash \mathtt{P\text{-}N} \rhd P & \cdots & \cdots \\ \#(\mathit{fst}(PR(P))) = n & (\Sigma, G \cup L) \vdash \mathtt{T}_n \rhd t_n & \kappa(t_n, n) \neq \mathit{error} \end{array}}{\Sigma, (CH, PR), G, L \vdash \mathtt{named\text{-}proc\ P\text{-}N\ T}_1 \cdots \mathtt{T}_n \rhd P(\kappa(t_1, 1), \ldots, \kappa(t_n, n)), \mathit{snd}(PR(P))}$$

10.4.3 BASIC-PROC

A CSP-CASL basic process BASIC-PROC is either an atomic primitive process or Run or Chaos.

```
BASIC-PROC     ::= SKIP-PROC | STOP-PROC | DIV-PROC | RUN-PROC
                 | CHAOS-PROC
SKIP-PROC      ::= skip-proc
STOP-PROC      ::= stop-proc
DIV-PROC       ::= div-proc
RUN-PROC       ::= run-proc EVENT-SET
CHAOS-PROC     ::= chaos-proc EVENT-SET
```

$$\boxed{\Sigma, \Lambda, G, L \vdash \mathtt{BASIC\text{-}PROC} \rhd p, \alpha}$$

We once again elide trivial rules corresponding to the disjoint union of a number of syntactic categories (see section 10.4.1), and focus on the specialised rules.

$$\boxed{\Sigma, \Lambda, G, L \vdash \mathtt{SKIP\text{-}PROC} \rhd p, \alpha}$$

A SKIP-PROC, is a simple atomic process; it never communicates, so its constituent alphabet α is the empty set.

$$\frac{}{\Sigma, \Lambda, G, L \vdash \mathtt{skip\text{-}proc} \rhd \mathit{SKIP}, \emptyset}$$

$$\boxed{\Sigma, \Lambda, G, L \vdash \mathtt{STOP\text{-}PROC} \rhd p, \alpha}$$

A STOP-PROC, is a simple atomic process; it never communicates, so its constituent alphabet α is the empty set.

$$\frac{}{\Sigma, \Lambda, G, L \vdash \mathtt{stop\text{-}proc} \rhd \mathit{STOP}, \emptyset}$$

$$\boxed{\Sigma, \Lambda, G, L \vdash \mathtt{DIV\text{-}PROC} \rhd p, \alpha}$$

A DIV-PROC, is a simple atomic process; it never communicates, so its constituent alphabet α is the empty set.

$$\frac{}{\Sigma, \Lambda, G, L \vdash \mathtt{div\text{-}proc} \rhd \mathit{DIV}, \emptyset}$$

$$\boxed{\Sigma, \Lambda, G, L \vdash \mathtt{RUN\text{-}PROC} \rhd p, \alpha}$$

A RUN-PROC is an atomic process with an associated EVENT-SET; its constituent alphabet α is the computed event set.

$$\frac{\Sigma, \Lambda \vdash \texttt{EVENT-SET} \rhd es}{\Sigma, \Lambda, G, L \vdash \texttt{run-proc EVENT-SET} \rhd RUN(es), es}$$

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{CHAOS-PROC} \rhd p, \alpha}$$

A CHAOS-PROC is an atomic process with an associated EVENT-SET; its constituent alphabet α is the computed event set.

$$\frac{\Sigma, \Lambda \vdash \texttt{EVENT-SET} \rhd es}{\Sigma, \Lambda, G, L \vdash \texttt{chaos-proc EVENT-SET} \rhd CHAOS(es), es}$$

10.4.4 PREFIX-PROC

A PREFIX-PROC consists of a communication event EVENT and a process term PROCESS . The communication event may introduce a new binding to the set of process-local variables.

```
PREFIX-PROC    ::= prefix-proc EVENT PROCESS
```

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{PREFIX-PROC} \rhd p, \alpha}$$

The EVENT rule receives as context $G \cup L$: if the event involves a CASL term (i.e. it is a *term prefix* or a *channel send*), that CASL term may refer to any process-global and process-local variables visible at this point.

If the event is a *channel nondeterministic send* or a *channel receive*, the EVENT rule will introduce a new variable binding, encoded in the variable set X; otherwise, X is the empty set. In either case, it is added to the local variable set L provided as context to the PROCESS rule. Thus, that rule computes its process term in an expanded context.

The constituent alphabet α computed by this rule is the union of the computed event set and that of the process term.

$$\frac{\Sigma, CH, G \cup L \vdash \texttt{EVENT} \rhd e, \alpha_e, X \qquad \Sigma, \Lambda, G, L \cup X \vdash \texttt{PROCESS} \rhd p, \alpha_p}{\Sigma, (CH, PR), P, G, L \vdash \texttt{prefix-proc EVENT PROCESS} \rhd e \rightarrow p, \alpha_e \cup \alpha_p}$$

10.4.5 INTPRE-PROC

An INTPRE-PROC consists of a single variable declaration SVAR-DECL and a process term PROCESS; it introduces a new binding to the set of process-local variables.

```
INTPRE-PROC    ::= intpre-proc SVAR-DECL PROCESS
```

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{INTPRE-PROC} \rhd p, \alpha}$$

Note the similarity between this rule and that of PREFIX-PROC, above. This rule always introduces a new variable binding via the SVAR-DECL rule, which is then added to the local variable set passed as context to the PROCESS rule. Thus, that rule computes its process term in an expanded context. The constituent alphabet α computed by this rule is that of the process term plus the sort of the newly introduced variable.

$$\frac{\Sigma_S \vdash \texttt{SVAR-DECL} \rhd x, s \qquad \Sigma, \Lambda, G, L + (x, s) \vdash \texttt{PROCESS} \rhd p, \alpha_p}{\Sigma, \Lambda, G, L \vdash \texttt{intpre-proc SVAR-DECL PROCESS} \rhd \sqcap x :: s \rightarrow p, \alpha_p \cup \{s\}}$$

10.4.6 EXTPRE-PROC

An EXTPRE-PROC consists of a single variable declaration SVAR-DECL and a process term PROCESS; it introduces a new binding to the set of process-local variables.

```
EXTPRE-PROC    ::= extpre-proc SVAR-DECL PROCESS
```

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{EXTPRE-PROC} \rhd p, \alpha}$$

See comments for INTPRE-PROC, above.

$$\frac{\Sigma_S \vdash \texttt{SVAR-DECL} \rhd x, s \qquad \Sigma, \Lambda, G, L + (x, s) \vdash \texttt{PROCESS} \rhd p, \alpha_p}{\Sigma, \Lambda, G, L \vdash \texttt{extpre-proc SVAR-DECL PROCESS} \rhd \Box x :: s \rightarrow p, \alpha_p \cup \{s\}}$$

10.4.7 SEQ-PROC

A SEQ-PROC consists of two process terms, the second of which is considered in a context having an empty process-local variable set.

```
SEQ-PROC       ::= seq-proc PROCESS PROCESS
```

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{SEQ-PROC} \rhd p, \alpha}$$

Note that we pass the empty local variable set to the second PROCESS rule; this encodes the necessary reinitialisation of process-local variables in a sequential composition — see section 8.1.7. The constituent alphabet α computed by this rule is the union of those of the two process terms.

$$\frac{\Sigma, \Lambda, G, L \vdash \texttt{PROCESS}_1 \rhd p, \alpha_p \qquad \Sigma, \Lambda, G, \emptyset \vdash \texttt{PROCESS}_2 \rhd q, \alpha_q}{\Sigma, \Lambda, G, L \vdash \texttt{seq-proc PROCESS}_1\ \texttt{PROCESS}_2 \rhd p\ ;\ q, \alpha_p \cup \alpha_q}$$

10.4.8 INTCHOICE-PROC

An INTCHOICE-PROC consists of two process terms, both considered in the same context.

```
INTCHOICE-PROC ::= intchoice-proc PROCESS PROCESS
```

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{INTCHOICE-PROC} \rhd p, \alpha}$$

An INTCHOICE-PROC contains two PROCESSes, both of which are considered in the same context as the INTCHOICE-PROC itself. The constituent alphabet α computed by this rule is the union of those of the two process terms.

$$\frac{\Sigma, \Lambda, G, L \vdash \texttt{PROCESS}_1 \rhd p, \alpha_p \qquad \Sigma, \Lambda, G, L \vdash \texttt{PROCESS}_2 \rhd q, \alpha_q}{\Sigma, \Lambda, G, L \vdash \texttt{intchoice-proc PROCESS}_1\ \texttt{PROCESS}_2 \rhd p \sqcap q, \alpha_p \cup \alpha_q}$$

10.4.9 EXTCHOICE-PROC

An EXTCHOICE-PROC consists of two process terms, both considered in the same context.

```
EXTCHOICE-PROC ::= extchoice-proc PROCESS PROCESS
```

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{EXTCHOICE-PROC} \rhd p, \alpha}$$

An EXTCHOICE-PROC contains two PROCESSes, both of which are considered in the same context as the EXTCHOICE-PROC itself. The constituent alphabet α computed by this rule is the union of those of the two process terms.

$$\frac{\Sigma, \Lambda, G, L \vdash \texttt{PROCESS}_1 \rhd p, \alpha_p \qquad \Sigma, \Lambda, G, L \vdash \texttt{PROCESS}_2 \rhd q, \alpha_q}{\Sigma, \Lambda, G, L \vdash \texttt{extchoice-proc PROCESS}_1\ \texttt{PROCESS}_2 \rhd p \,\Box\, q, \alpha_p \cup \alpha_q}$$

10.4.10 INTPAR-PROC

An INTPAR-PROC consists of two process terms, both considered in the same context.

```
INTPAR-PROC    ::= intpar-proc PROCESS PROCESS
```

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{INTPAR-PROC} \rhd p, \alpha}$$

An INTPAR-PROC contains two PROCESSes, both of which are considered in the same context as the INTPAR-PROC itself. The constituent alphabet α computed by this rule is the union of those of the two process terms.

$$\frac{\Sigma, \Lambda, G, L \vdash \texttt{PROCESS}_1 \rhd p, \alpha_p \qquad \Sigma, \Lambda, G, L \vdash \texttt{PROCESS}_2 \rhd q, \alpha_q}{\Sigma, \Lambda, G, L \vdash \texttt{intpar-proc PROCESS}_1\ \texttt{PROCESS}_2 \rhd p \,|||\, q, \alpha_p \cup \alpha_q}$$

10.4.11 SYNPAR-PROC

A SYNPAR-PROC consists of two process terms, both considered in the same context.

```
SYNPAR-PROC    ::= synpar-proc PROCESS PROCESS
```

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{SYNPAR-PROC} \rhd p, \alpha}$$

A SYNPAR-PROC contains two PROCESSes, both of which are considered in the same context as the SYNPAR-PROC itself. The constituent alphabet α computed by this rule is the union of those of the two process terms.

$$\frac{\Sigma, \Lambda, G, L \vdash \texttt{PROCESS}_1 \rhd p, \alpha_p \qquad \Sigma, \Lambda, G, L \vdash \texttt{PROCESS}_2 \rhd q, \alpha_q}{\Sigma, \Lambda, G, L \vdash \texttt{synpar-proc PROCESS}_1\ \texttt{PROCESS}_2 \rhd p \,||\, q, \alpha_p \cup \alpha_q}$$

10.4.12 GENPAR-PROC

A GENPAR-PROC consists of two process terms, both considered in the same context, and an event set.

```
GENPAR-PROC    ::= genpar-proc EVENT-SET PROCESS PROCESS
```

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{GENPAR-PROC} \rhd p, \alpha}$$

A GENPAR-PROC contains two PROCESSes, both of which are considered in the same context as the GENPAR-PROC itself, and an EVENT-SET, representing communications on which the process terms must synchronise. The constituent alphabet α computed by this rule is the union of the event set and those of the two process terms.

$$\frac{\Sigma, \Lambda \vdash \texttt{EVENT-SET} \rhd es \qquad \begin{array}{l}\Sigma, \Lambda, G, L \vdash \texttt{PROCESS}_1 \rhd p, \alpha_p \\ \Sigma, \Lambda, G, L \vdash \texttt{PROCESS}_2 \rhd q, \alpha_q\end{array}}{\Sigma, \Lambda, G, L \vdash \texttt{genpar-proc EVENT-SET PROCESS}_1\ \texttt{PROCESS}_2 \rhd p [\![es]\!] q, \alpha_p \cup \alpha_q \cup es}$$

10.4.13 ALPHAPAR-PROC

An ALPHAPAR-PROC consists of two process terms, both considered in the same context, and two event sets. The event sets must be 'included in' the constituent alphabets of the process terms.

```
ALPHAPAR-PROC  ::= alphapar-proc EVENT-SET EVENT-SET PROCESS PROCESS
```

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{ALPHAPAR-PROC} \rhd p, \alpha}$$

An ALPHAPAR-PROC contains two PROCESSes, both of which are considered in the same context as the ALPHAPAR-PROC itself, and two event sets, representing communications in which they may engage, and on the union of which they must sychronise. We require that the subsort closure $\downarrow (es_1)$ of the left-hand event set is a subset of the subsort closure $\downarrow (\alpha_p)$ of the constituent alphabet of the left-hand process; similarly, we require that the subsort closure $\downarrow (es_2)$ of right-hand event set is a subset of the subsort closure $\downarrow (\alpha_q)$ of the constituent alphabet of the left-hand process — see section 10.2.3. The constituent alphabet α computed by this rule is the union of those of the two process terms — we do not need to include the event sets as they are, by the previous sentence, already there.

Note that in this rule we use the abbreviated forms ES and P for EVENT-SET and PROCESS respectively.

$$\frac{\begin{array}{lll}\Sigma, \Lambda \vdash \texttt{ES}_1 \rhd es_1 & \Sigma, \Lambda, G, L \vdash \texttt{P}_1 \rhd p, \alpha_p & \downarrow(es_1) \subseteq \downarrow(\alpha_p) \\ \Sigma, \Lambda \vdash \texttt{ES}_2 \rhd es_2 & \Sigma, \Lambda, G, L \vdash \texttt{P}_2 \rhd q, \alpha_q & \downarrow(es_2) \subseteq \downarrow(\alpha_q)\end{array}}{\Sigma, \Lambda, G, L \vdash \texttt{alphapar-proc ES}_1\ \texttt{ES}_2\ \texttt{P}_1\ \texttt{P}_2 \rhd p [\![es_1 || es_2]\!] q, \alpha_p \cup \alpha_q}$$

10.4.14 HIDING-PROC

A HIDING-PROC consists of a process term and an event set representing communications which are not visible to the HIDING-PROC's environment.

```
HIDING-PROC    ::= hiding-proc PROCESS EVENT-SET
```

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{HIDING-PROC} \rhd p, \alpha}$$

The constituent alphabet α computed by this rule is the union of the event set and that of the process term.

$$\frac{\Sigma, \Lambda \vdash \texttt{EVENT-SET} \rhd es \qquad \Sigma, \Lambda, G, L \vdash \texttt{PROCESS} \rhd p, \alpha_p}{\Sigma, \Lambda, G, L \vdash \texttt{hiding-proc PROCESS EVENT-SET} \rhd p \setminus es, \alpha_p \cup es}$$

10.4.15 RENAMING-PROC

A RENAMING-PROC consists of a process term and a RENAMING, which is just a sequence of RENAMING-ITEMs.

```
RENAMING-PROC  ::= renaming-proc PROCESS RENAMING
```

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{RENAMING-PROC} \rhd p, \alpha}$$

The constituent alphabet α computed by this rule is the union of that of the process term, and that of the renaming (which just contains all of the sorts referenced in the renaming).

$$\frac{\Sigma \vdash \texttt{RENAMING} \rhd R, \alpha_R \qquad \Sigma, \Lambda, G, L \vdash \texttt{PROCESS} \rhd p, \alpha_p}{\Sigma, \Lambda, G, L \vdash \texttt{renaming-proc PROCESS RENAMING} \rhd p[R], \alpha_p \cup \alpha_R}$$

10.4.16 CONDITION-PROC

A CONDITION-PROC consists of a FORMULA and two PROCESSes.

```
CONDITION-PROC ::= condition-proc FORMULA PROCESS PROCESS
```

$$\boxed{\Sigma, \Lambda, G, L \vdash \texttt{CONDITION-PROC} \rhd p, \alpha}$$

The constituent alphabet α computed by this rule is the union of those of the two process terms, plus the sorts arising from the CASL formula (see section 10.1.2).

Note that in this rule we use the abbreviated forms F and P for FORMULA and PROCESS respectively.

$$\frac{\begin{array}{rl} & \Sigma, \Lambda, G, L \vdash \texttt{P}_1 \rhd p, \alpha_p \\ (\Sigma, G \cup L) \vdash \texttt{F} \rhd \varphi \qquad & \Sigma, \Lambda, G, L \vdash \texttt{P}_2 \rhd q, \alpha_q \end{array}}{\Sigma, \Lambda, G, L \vdash \texttt{condition-proc F P}_1\ \texttt{P}_2 \rhd \text{if } \varphi \text{ then } p \text{ else } q, \alpha_p \cup \alpha_q \cup sorts(\varphi)}$$

10.5 Supporting elements

This section contains the static semantics rules for the 'supporting elements' of a CSP-CASL specification which are neither at the 'top level', nor processes — i.e. those whose abstract syntax is defined in section 9.1.3. In particular, here we present rules for communication events and event sets, single variable declarations, and renamings.

10.5.1 EVENT-SET and COMM-TYPE

An EVENT-SET is a sequence of COMM-TYPEs; these are undistinguished identifiers, to be differentiated into a set of sort names and channel names.

```
EVENT-SET      ::= event-set COMM-TYPE*
COMM-TYPE      ::= comm-type SIMPLE-ID
```

$$\boxed{\Sigma, \Lambda \vdash \texttt{EVENT-SET} \rhd es}$$

Given a CASL signature Σ, a CSP signature Λ, and a syntactic EVENT-SET, we compute an event set es of $\varepsilon \in \mathit{CommType}$, considering each COMM-TYPE independently.

Note that the COMM-TYPE list may contain duplicate entries (though obviously the set produced has no such duplication), and that the event set will be closed under CASL subsort when compared with other alphabets (see section 10.2.4.1).

$$\frac{\begin{array}{c}\Sigma, CH \vdash \texttt{COMM-TYPE}_1 \rhd \varepsilon_1 \\ \cdots \\ \Sigma, CH \vdash \texttt{COMM-TYPE}_1 \rhd \varepsilon_n\end{array}}{\Sigma, (CH, PR) \vdash \texttt{event-set COMM-TYPE}_1 \cdots \texttt{COMM-TYPE}_n \rhd \{\varepsilon_1, \ldots, \varepsilon_n\}}$$

$$\boxed{\Sigma, CH \vdash \texttt{COMM-TYPE} \rhd \varepsilon}$$

Given a CASL signature Σ, a *ChanSet* CH, and a syntactic COMM-TYPE, we compute a communication type $\varepsilon \in \mathit{CommType}$. The following two rules distinguish between sort and channel communictation types by looking for the COMM-TYPE in the sort names in the sort part of the CASL signature Σ, and in the channel names in the domain of the *ChanSet* CH. For a given COMM-TYPE, at most one of these rules will apply — see section 10.3.2.

$$\frac{\vdash \texttt{SIMPLE-ID} \rhd s \qquad s \in S_\Sigma}{\Sigma, CH \vdash \texttt{event-set-item SIMPLE-ID} \rhd s}$$

$$\frac{\vdash \texttt{SIMPLE-ID} \rhd c \qquad c \in \mathit{Dom}(CH)}{\Sigma, CH \vdash \texttt{event-set-item SIMPLE-ID} \rhd (c, CH(c))}$$

10.5.2 EVENT

An EVENT is either a term prefix event, a channel send event, a channel nondeterministic send event, or a channel receive event.

```
EVENT          ::= term-event TERM
                 | chan-send CHAN-NAME TERM
                 | chan-nondet-send CHAN-NAME VAR SORT
                 | chan-recv CHAN-NAME VAR SORT
```

$$\boxed{\Sigma, CH, X \vdash \texttt{EVENT} \rhd e, \alpha, X'}$$

Given a CASL signature Σ, a *ChanSet* CH, a variable set X and a syntactic EVENT, we compute an event e, a constituent alphabet α containing any communication types arising from the event, and a variable set X' containing any newly introduced local variables. There are four rules: one per event type.

Term prefix

A term prefix event contains a TERM which is checked in the given context and produces a fully qualified term $t \in \mathit{FQTerm}$. The constituent alphabet α of the event contains just the sort of t; no new variables are introduced.

$$\frac{(\Sigma, X) \vdash \texttt{TERM} \rhd t}{\Sigma, CH, X \vdash \texttt{term-event TERM} \rhd t, \{\mathit{sort}(t)\}, \emptyset}$$

Channel send

A channel send event contains a CHAN-NAME and a TERM; the latter is checked in the given context and produces a fully qualified term $t \in FQTerm$. We attempt to cast the term to the sort of the channel as described in section 10.2.4.2 — this implicitly encodes the requirement that the channel name has been declared, otherwise the lookup in CH will fail. The constituent alphabet α of the event contains the sort of the cast term and the channel name tagged with the sort of the cast term; no new variables are introduced.

Note that in this rule we use the abbreviated forms C-N and T for CHAN-NAME and TERM respectively.

$$\frac{\vdash \texttt{C-N} \rhd c \qquad (\Sigma, X) \vdash \texttt{T} \rhd t \qquad cast(t, CH(c)) \neq error}{\Sigma, CH, X \vdash \texttt{chan-send C-N T} \rhd c!t, \{sort(cast(t, CH(c))), (c, sort(cast(t, CH(c))))\}, \emptyset}$$

Channel nondeterministic send

A channel nondeterministic send event contains a CHAN-NAME, a VAR and a SORT. We require that the sort has been previously declared, and that it is a subsort of the declared sort of the channel (and thus, we implicitly require that the channel name has been previously declared). The constituent alphabet α of the event contains the specified sort and the channel name tagged with the specified sort; the new variable set X' contains one element, associating the declared variable name with the specified sort.

$$\frac{\vdash \texttt{CHAN-NAME} \rhd c \quad \vdash \texttt{VAR} \rhd x \quad \vdash \texttt{SORT} \rhd s \quad s \in \Sigma_S \quad s \leq CH(c)}{\Sigma, CH, X \vdash \texttt{chan-nondet-send CHAN-NAME VAR SORT} \rhd c!x :: s, \{s, (c, s)\}, \{s \mapsto \{x\}\}}$$

Channel receive

A channel receive event contains a CHAN-NAME, a VAR and a SORT. We require that the sort has been previously declared, and that it is a subsort of the declared sort of the channel (and thus, we implicitly require that the channel name has been previously declared). The constituent alphabet α of the event contains the specified sort and the channel name tagged with the specified sort; the new variable set X' contains one element, associating the declared variable name with the specified sort.

$$\frac{\vdash \texttt{CHAN-NAME} \rhd c \quad \vdash \texttt{VAR} \rhd x \quad \vdash \texttt{SORT} \rhd s \quad s \in \Sigma_S \quad s \leq CH(c)}{\Sigma, CH, X \vdash \texttt{chan-recv CHAN-NAME VAR SORT} \rhd c?x :: s, \{s, (c, s)\}, \{s \mapsto \{x\}\}}$$

10.5.3 SVAR-DECL

An SVAR-DECL is 'single variable declaration', introducing a new variable name having a particular sort.

```
SVAR-DECL      ::= svar-decl VAR SORT
```

$$\boxed{S \vdash \texttt{SVAR-DECL} \rhd x, s}$$

Given a set S of CASL sorts and a syntactic SVAR-DECL, we (trivially) compute a variable name x and a CASL sort s. We require that the sort s computed from SORT is already known, i.e. it is a member of S.

$$\frac{\vdash \texttt{VAR} \rhd x \qquad \vdash \texttt{SORT} \rhd s \qquad s \in S}{S \vdash \texttt{svar-decl VAR SORT} \rhd x, s}$$

10.5.4 RENAMING and RENAMING-ITEM

A RENAMING is a sequence of RENAMING-ITEMs, which are undistinguised CASL IDs, to be resolved to names of CASL binary functions or predicates.

```
RENAMING      ::= renaming RENAMING-ITEM+
RENAMING-ITEM ::= renaming-item ID
```

$$\boxed{\Sigma \vdash \texttt{RENAMING} \rhd R, \alpha}$$

Given a CASL signature Σ and a syntactic RENAMING (which is a sequence of RENAMING-ITEMs), we compute a renaming R (which is a sequence of renaming items r_i) and a communication alphabet α containing the sorts which appear the in the renaming items.

$$\frac{\begin{array}{c}\Sigma \vdash \texttt{RENAMING-ITEM}_1 \rhd r_1, \alpha_{r_1} \\ \cdots \\ \Sigma \vdash \texttt{RENAMING-ITEM}_1 \rhd r_n, \alpha_{r_n}\end{array}}{\Sigma \vdash \texttt{renaming RENAMING-ITEM}_1 \cdots \texttt{RENAMING-ITEM}_n \rhd \langle r_1, \cdots r_n \rangle, \alpha_{r_1} \cup \cdots \cup \alpha_{r_n}}$$

$$\boxed{\Sigma \vdash \texttt{RENAMING-ITEM} \rhd r, \alpha}$$

Given a CASL signature Σ containing sets TF, PF, and P of total functions, partial functions and predicates (respectively), and a syntactic RENAMING-ITEM, which is just a CASL ID, we compute a renaming item $r \in (Id \times Sort \times Sort)$ and a communication alphabet α containing the sorts seen in r. The RENAMING-ITEM is expected to be the name of a previously declared binary function or predicate, determined via lookup in Σ; the corresponding renaming item r is then a triple consisting of the function or predicate name and the two sorts appearing in its profile.

The lookup is fairly complex, as we require that the name is unique among the binary functions and predicates.

$$\frac{\vdash \texttt{ID} \rhd i \qquad \exists!(s_1, s_2) \in Sort \times Sort \bullet \exists! X \in \{TF_{(\langle s_1 \rangle, s_2)}, PF_{(\langle s_1 \rangle, s_2)}, P_{\langle s_1, s_2 \rangle}\} \bullet i \in X}{(S, TF, PF, P, \leq) \vdash \texttt{renaming-item ID} \rhd (i, s_1, s_2), \{s_1, s_2\}}$$

Chapter 11

CASL subsorts and Local Top Elements

Contents

11.1 Introduction

As noted in 2.2.1, in CASL we may declare a sort to be a **subsort** of another sort (the *supersort*), in order to capture the idea that instances of the subsort are 'special cases' of the supersort. Given a CSP-CASL specification it is important to check that any subsort relationships defined in the data part have *local top elements* — see section 4.2.4 and [Rog06]. Informally, this means that whenever a sort is a subsort of two other sorts, we require that the two supersorts are themselves subsorts of a common supersort – the common supersort is then the corresponding top element.

In this chapter, we define this notion more precisely, formulate a simple algorithm for the computation of local top elements, and present a Haskell implementation of the algorithm. The implementation is used to look for local top elements in subsort relations, as part of the static checks performed by the CSP-CASL tool — see chapter 10.

Note that the definitions, algorithm and implementation presented here relate to local top elements within the context of arbitrary binary relations on a single set. This is a small departure from, and more general than, their original formulation in [Rog06], in which the context is CASL subsorts (a preorder). It turns out that the concepts of local top elements and preorder are orthogonal, and may be considered separately. Naturally, when dealing with CASL subsorts in practice we must remember that they form a preorder – but we may take advantage of the fact that any required computation is independent of the local top elements question. We consider the implications of preorders in more detail in section 11.4.

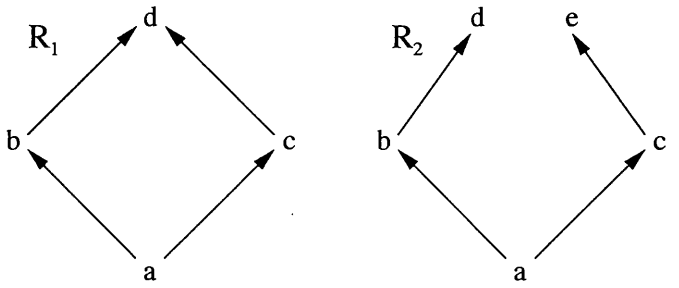


Figure 11.1: Graphs of relations R_1 and R_2 (example 11.4).

11.2 Local top elements

Definition 11.1: A binary relation R on a set S is a subset of $S \times S$. We write $(a, b) \in R$ as aRb.

Definition 11.2: Given a set S and a binary relation $R \subseteq S \times S$, R has *local top elements* iff:

$$\forall\, a, b, c \in S \bullet aRb \wedge aRc \wedge b \neq c \Rightarrow \exists\, d \in S \bullet bRd \wedge cRd$$

To work towards an algorithm for checking for local top elements, we define a binary relation's *top element obligations* and the *corresponding top elements* as follows.

Definition 11.3 (Obligations and top elements)**:** Given a set S and a binary relation R, we define the *top element obligations of* R as:

$$C_R = \{(a, \{b, c\}) \in S \times FinSet(S) \mid aRb \wedge aRc\}$$

By abuse of notation, we write an individual obligation $(a, \{b, c\})$ as ${}^{b}a^{c}$. (Note then that ${}^{b}a^{c}$ and ${}^{c}a^{b}$ are the same obligation.)

For each ${}^{b}a^{c} \in C_R$ we define the *corresponding top elements of* ${}^{b}a^{c}$ as:

$$T_C = \{d \in S \mid bRd \wedge cRd\}.$$

Then, R has local top elements iff $\forall\, C \in C_R \bullet T_C \neq \emptyset$. Checking for the presence of local top elements is thus a matter of computing the top element obligations for the relation, and then checking that each has a non-empty set of corresponding top elements.

Example 11.4: Consider the relations

$$R_1 = \{(a, b), (a, c), (b, d), (c, d)\} \text{ (over } \{a, b, c, d\})$$
$$R_2 = \{(a, b), (a, c), (b, d), (c, e)\} \text{ (over } \{a, b, c, d, e\})$$

Figure 11.1 shows these relations as directed graphs, with an arc from x to y iff xRy.

Clearly $C_{R_1} = C_{R_2} = \{{}^{b}a^{c}\}$. In R_1, $T_{{}^{b}a^{c}} = \{d\}$; ${}^{b}a^{c}$ is the only top element obligation in R_1, and it has a corresponding top element, so R_1 has local top elements. In R_2, however, $T_{{}^{b}a^{c}} = \emptyset$, i.e. R_2 does not have local top elements (as ${}^{b}a^{c}$ witnesses).

11.3 Algorithm and Haskell implementation

An algorithm to check for local top elements arises naturally from the definitions given above. Given a set S and a binary relation R on that set, we first compute C_R, the set of top element obligations of R. Then, for each ${}^{b}a^{c} \in C_R$, we compute $T_{{}^{b}a^{c}}$, the set of corresponding top elements. R does not have local top elements iff $T_C = \emptyset$ for any C.

This algorithm may be implemented in Haskell very directly, thanks to Haskell's excellent support for higher order functions and rich standard library. We make extensive use of the standard `Set` type and its facilities to `map` over sets and `filter` out particular elements, and of Haskell's excellent pattern-matching capabilities. We find this implementation to be an excellent example of Haskell's ability to directly encode high-level problems.

11.3.1 Data types

We start by defining data types for binary relations and obligations. A `Relation` relates two types which may differ; a `BinaryRelation` is simply a relation on a single type. `Obligation` is an algebraic data type where `Obligation x y z` $= {}^{y}x^{z}$ in the notation given above.

```
import qualified Data.Set as S
import List
import Maybe

type Relation a b = S.Set (a, b)
type BinaryRelation a = Relation a a

data Obligation a = Obligation a a a
```

As noted in definition 11.3, it is clear that ${}^{b}a^{c} = {}^{c}a^{b}$. We encode this in Haskell by declaring our `Obligation` type to be an instance of the `Eq` type class, and providing a suitable implementation of ==, the equality check function:

```
instance Eq a => Eq (Obligation a) where
    (Obligation n m o) == (Obligation x y z) =
        (n==x) && ((m,o)==(y,z) || (m,o)==(z,y))
```

In order to be able to use the standard library's `Set` type for `Obligations`, `Obligation` also needs to be an instance of the `Ord` type class (defining not just equality, but ordering). We use a simple lexicographic ordering, with an exception for equality as noted above. (If we just used the simple lexicographic ordering, with no special case for equality, a `Set` of `Obligations` could in fact contain both ${}^{b}a^{c} = {}^{c}a^{b}$. The reason for Haskell's use of `Ord` rather than the more obvious `Eq` in the `Set` type is apparently related to performance concerns in the implementation of a number of `Set` functions — see for example the discussion of `fromDistinctAscList`, below.)

```
instance Ord a => Ord (Obligation a) where
    compare (Obligation n m o) (Obligation x y z)
        | (Obligation n m o) == (Obligation x y z) = EQ
        | otherwise = compare (n,m,o) (x,y,z)
```

Finally, so that we may easily obtain string representations of obligations (e.g. for debugging), we also make it an instance of the `Show` class:

```
instance Show a => Show (Obligation a) where
    show (Obligation x y z) = show [x,y,z]
```

Having defined a suitable type for our key data type, `Obligation`, much of the work is done. The `Set` type will now handle such matters as ignoring duplicates, and provides high level functionality such as `map` and `filter`.

11.3.2 Computation of obligations and top elements

We start with a couple of utility functions upon which the algorithm is built. First, transformation of a set into its cartesian product. This is naturally expressed in Haskell as a list comprehension; the only complications are `toAscList` and `fromDistinctAscList`[1] (necessary because it is a *list* comprehension – not a set comprehension), and the presence of an `Ord a` restriction in the type signature (arising due to use of the `Set` type).

```
cartesian :: Ord a => S.Set a -> S.Set (a,a)
cartesian x = S.fromDistinctAscList [(i,j) | i <- xs, j <- xs]
    where xs = S.toAscList x
```

Second, `stripMaybe`, to turn a set of `Maybe a` into a set of `a`. `Maybe` is a polymorphic algebraic data type representing 'possibly present' data: values of type `Maybe a` may be either `Just a` (data is present) or `Nothing` (data is not present). This function discards the `Nothings` and extracts the contents of the `Justs`. Most of the work is in fact by the library function `Maybe.catMaybes`, which does exactly that on a list (not a set).

```
stripMaybe :: Ord a => S.Set (Maybe a) -> S.Set a
stripMaybe x = S.fromList $ Maybe.catMaybes $ S.toList x
```

Computation of top element obligations is then expressed easily. We take a `BinaryRelation a`, i.e. a set of `(a,a)` pairs. We compute its cartesian product, giving us all possible pairings of pairs. We map over those pairings looking for ones of the right 'shape', turning each into either a `Nothing` (not an obligation) or a `Just Obligation`. Finally we strip the `Maybes` and are left with a set of obligations over `a`.

```
obligations :: Ord a => BinaryRelation a -> S.Set (Obligation a)
obligations r = stripMaybe $ S.map isObligation (cartesian r)
    where isObligation ((w,x),(y,z)) =
              if (w==y) && (x /= z) && (w /= z) && (w /= x)
              then Just (Obligation w x z)
              else Nothing
```

For each obligation, we need to compute its corresponding top elements. We take as input the obligation and the cartesian product of the entire (original) binary relation. We filter that cartesian product down to only those elements containing top element candidates for the obligation; then we map over that, extracting the top elements. The set we return is a set of top elements for the specified candidate.

[1] `fromDistinctAscList` assumes the list is ascending and contains no duplicates. The latter condition is obviously met since our input is a set; the former is met by the list comprehension and use of `toAscList`. `fromDistinctAscList` is O(n), as opposed to the 'safe' `fromList`, which makes so such assumptions but only performs O(n log n).

```
findTops :: Ord a => BinaryRelation (a,a) -> (Obligation a) -> S.Set a
findTops c cand = S.map get_top (S.filter (is_top cand) c)
    where is_top (Obligation _ y z) ((m,n),(o,p))=((m==y)&&(o==z)&&(n==p))
          get_top ((_,_),(_,p)) = p
```

Finally, we compute the local top elements of a relation by computing its obligations and then, for each obligation, computing its set of corresponding top elements.

```
localTops :: Ord a => BinaryRelation a -> S.Set (Obligation a, S.Set a)
localTops r = S.map (\x -> (x, m x)) (obligations r)
    where m = findTops $ cartesian r
```

Checking the result of this function for empty sets on the right hand side is trivial and omitted here.

11.4 Local top elements of a preorder

As mentioned in section 11.1, the CASL subsorting relation forms a *preorder*. In this section, we consider the implications for our algorithm and its implementation.

Definition 11.5: A binary relation $\lesssim$ on a set S is a **preorder** if it is:

- *reflexive*: $a \in S \Rightarrow a \lesssim a$
- *transitive*: $a \lesssim b \wedge b \lesssim c \Rightarrow a \lesssim c$

Given an arbitrary binary relation R over a set S, we can always obtain a preorder over S by taking R's *reflexive transitive closure* $R^{+=}$.

Thus, given a set of sorts and their subsort relation, if we aim to check for local top elements, we must first transform the relation into its reflexive transitive closure, and look for local top elements there.

Example 11.6: Consider again the relations R_1 and R_2 (example 11.4. They are neither transitive nor reflexive, but under reflexive transitive closure we obtain the corresponding preorders:

$$\lesssim_1 = \{(a,b),(a,c),(b,d),(c,d),(a,a),(b,b),(c,c),(d,d),(a,d)\}$$
$$\lesssim_2 = \{(a,b),(a,c),(b,d),(c,e),(a,a),(b,b),(c,c),(d,d),(e,e),(a,d),(a,e)\}$$

Figure 11.4 shows these preorders as graphs (arcs arising through transitive closure are heavy; those arising through reflexive closure are dotted). Then:

$$C_{\lesssim_1} = C_{R_1} \cup \{{}^{b}a^{d}, {}^{c}a^{d}, {}^{a}a^{b}, {}^{a}a^{c}, {}^{a}a^{d}, {}^{b}b^{d}, {}^{c}c^{d}\}$$
$$C_{\lesssim_2} = C_{R_2} \cup \{{}^{b}a^{d}, {}^{c}a^{e}, {}^{d}a^{e}, {}^{a}a^{b}, {}^{a}a^{c}, {}^{a}a^{d}{}^{a}a^{e}, {}^{b}b^{d}, {}^{c}c^{d}\}$$

It is easy to check that $\lesssim_1$, like R_1, has local top elements. The obligations ${}^{b}a^{d}$, ${}^{c}a^{d}$ introduced by transitive closure both have d as top element — see below.

Similar comments may be made of $\lesssim_2$ which, like R_2, does not have local top elements: ${}^{b}a^{c}$ still has no top element, though again, all obligations introduced by transitive closure do (e.g. ${}^{b}a^{d}$, with top element d), as do all those introduced through reflexive closure (e.g. ${}^{a}a^{c}$, with top element c).

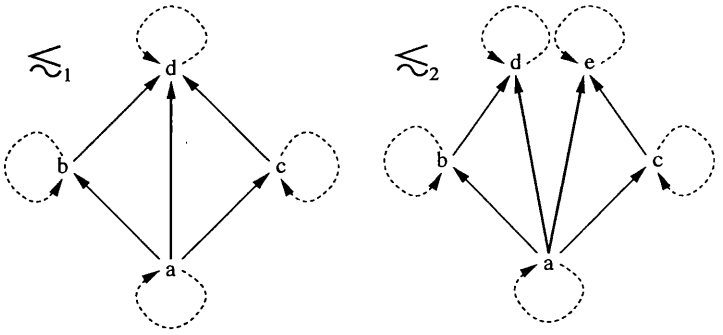


Figure 11.2: Graphs of preorders $\lesssim_1$ and $\lesssim_2$ (example 11.6).

Now, in the above example we saw that reflexive closure only introduced trivially satisfiable obligations; indeed, it is easy to see that this is true in general: every obligation introduced by reflexive closure has form ${}^{x}x^{y}$, with y a top element. If we write $r(R)$ as the reflexive closure of a relation R then we have:

$$C_{r(R)} = C_R \cup \{{}^{x}x^{y} \mid x, y \in S \bullet xRy\}$$

with y a top element for ${}^{x}x^{y}$ trivially. Thus, for the purpose of checking for local top elements, we do not in fact need to compute a relation's reflexive closure; however, the same cannot be said for transitive closure: in general it *will* introduce new obligations not trivially satisfied.

In practice, however, the problem of computing a relation's transitive closure has already been solved for us: the relation we actually want to check for local top elements is the subsort relation on CASL sorts, arising from a specification's data part, which is parsed and statically analysed by existing machinery in HETS. It turns out that this relation is then made available to us, already subjected to transitive closure: as such, the algorithm described in section 11.3 is adequate to check a CSP-CASL specification for local top elements.

Chapter 12

Tool Implementation

Contents

In this chapter we discuss the implementation of our CSP-CASL tool within the framework of the HETS toolset. In section 12.1 we present an overview of the HETS codebase and consider its overall structure; sections 12.2, 12.3 and 12.4 then describe our implementation of CSP-CASL parsing, pretty printing and static analysis respectively, within that framework; finally, in section 12.5 we describe the testing strategies employed during development. An exhaustive description of the CSP-CASL codebase is beyond the scope of this work; as such, we focus on key features, general concerns, and some particular areas of interest.

12.1 Overview of HETS implementation

At time of writing, the HETS codebase[1] consists of 568 Haskell modules and 655 other files (e.g. documentation and test cases) in a tree of 131 directories.

The core of HETS is entirely logic-independent, and provides a framework for heteregeneous specification over an arbitrary logic graph. The module `hets.hs` is the main program, to be compiled to the binary executable `Hets`; then there are various directories providing, e.g.: command-line interface, interpretation of command-line arguments and file I/O (`Driver`); graphical user interface (`GUI`); abstract syntax, parsing and pretty-printing of HETCASL structured and architectural specifications and specification libraries (`Syntax`); static analysis of HETCASL structured specifications (`Static`); and institution-based infrastructure for presenting a logic to HETS (`Logic`).

There are a number of common tasks which any CASL-related logic-specific code must perform, in particular (though not only) parsing, pretty printing and static analysis. Rather than

[1] Downloadable from the HETS home page at `http://www.informatik.uni-bremen.de/cofi/hets/`

have each logic 'reinvent the wheel' much code is factored out into reusable logic-generic modules. In particular, the directory `Common` contains a large number of modules providing functionality which almost any CASL-related logic will require; for example:

- `Common/Lexer.hs`, `Common/Id.hs`, `Common/Keywords.hs`, `Common/Token.hs`, `Common/AnnoState.hs` — lexical analysis and parsing of CASL-family languages (see section 12.2).
- `Common/Doc.hs`, `Common/DocUtils.hs` — pretty printing facilities (see section 12.3).
- `Common/Lib/State.hs`, `Common/Result.hs` — state and result monads for static analysis (see section 12.4).

All of the above modules (and a few others) are used by the CSP-CASL code discussed later. Furthermore, a logic which extends or otherwise interfaces with CASL will typically also reuse code which is nominally specific to CASL basic specifications; in our case, the CSP-CASL code makes reference to:

- `CASL/AS_Basic_CASL.hs` — abstract syntax of CASL basic specifications; reused by the CSP-CASL abstract syntax, which references `TERM`, `SORT`, `VAR`, etc.
- `CASL/Formula.hs` — parsers for CASL formulae and terms; reused by the CSP-CASL parser.
- `CASL/Sign.hs`, `CASL/StatAna.hs` — CASL signatures and static analysis; reused by CSP-CASL static analysis.

Finally, logic-specific code resides in immediate subdirectories of the root directory, each named after the logic in question. For example, there are logic-specific directories `CASL` (CASL basic specifications), `Haskell` (for HASCASL), `Propositional` (for propositional logic) and of course `CspCASL`. Each such directory contains code for dealing with logic-specific basic specifications — the equivalents of CASL basic specifications (see section 2.2.1); such basic specifications may then be included structurally in heterogeneous specifications as described in chapter 5.

The exact contents of the logic-specific directories vary from logic to logic, but typically include modules for:

- Representation of abstract syntax, implemented by algebraic data types, values of which represent terms in the logic's abstract syntax.
- Parsing, implemented using Parsec and the CASL-family parsing functions mentioned above.
- Pretty printing, using `Common/Doc.hs`, `Common/DocUtils.hs` and others.
- Static analysis of abstract syntax terms, which involves manipulation of (stateful) signatures and generation of debug, warning, and error messages. This also typically requires the definition of data types for semantic objects distinct from abstract syntax terms; depending on the logic's complexity, these might be defined in a separate module, or along with the static analysis.
- Integration with HETS' structuring mechanisms, by instantiating the type class `Logic` defined in `Logic/Logic.hs`.

Each logic also requires certain entries in the top-level `Makefile` enabling its code to be built with the rest of the system, and possibly some modules in the `Comorphisms` directory, defining morphisms between logics (beyond the scope of this thesis).

Note also that a particular logic can in some cases introduce requirements outside its logic-specific directory; e.g. and in particular, pretty printing of logic-specific symbols in LaTeX may (and in the case of CSP-CASL, does) require modifications to the general pretty-printing code — see section 12.3.

12.2 Implementation of parser

The parser's job is to transform an input text into a term over the abstract syntax; this, then, involves two artifacts: the data types used to represent such a term, and the functions used to perform the transformation. In this section, we discuss these two aspects, describing our overall approach, various relevant features of Parsec and HETS, and some interesting nuances.

12.2.1 Representation of abstract syntax

CSP-CASL's abbreviated abstract syntax (see section 9.2) is encoded using algebraic data types in the modules `CspCASL/AS_CspCASL.der.hs` (for everything above the level of a process term) and `CspCASL/AS_CspCASL_Process.der.hs` (for the rest). The encoding is very direct; for example, the grammar fragment:

```
EVENT          ::= term-event TERM
                 | chan-send CHAN-NAME TERM
                 | chan-nondet-send CHAN-NAME VAR SORT
                 | chan-recv CHAN-NAME VAR SORT
```

is implemented in `AS_CspCASL_Process.der.hs` as:

```
data EVENT = TermEvent (TERM ()) Range
           | ChanSend CHANNEL_NAME (TERM ()) Range
           | ChanNonDetSend CHANNEL_NAME VAR SORT Range
           | ChanRecv CHANNEL_NAME VAR SORT Range
           deriving (Show,Eq)
```

This representative example illustrates a number of important points. It defines a single data type, `EVENT`, with four constructors (`TermEvent`, etc.), with varying parameters. Values created using the various constructors are differentiated where necessary by pattern matching on the constructor name (e.g. see section 12.3).

We reuse CASL abstract syntax elements such as `TERM` (which is parametrised, here with the empty type `()`), `SORT` and `VAR`. The `EVENT` data type is a derived instance of the type classes `Show` and `Eq`, automatically providing string representations of, and equality comparisons on, its values.

Note that each constructor includes a `Range` parameter; this data type (defined in `Common/Id.hs`) stores syntactic positional information, enabling very specific error messages (see section 12.4 and examples in chapter 13). This is partially enabled by the 'type sensitive preprocessor'

Drift[2]: in this case, `AS_CspCASL_Process.der.hs` is transformed at build-time to `AS_CspCASL_Proc` with automatically-derived implementations of the `getRange` function (see section 12.2.2.3).

The abstract syntax data types refer to each other; in the above example, the data type `CHANNEL_NAME` is just a type synonym, defined in the same module, for the CASL data type `SIMPLE_ID`:

```
type CHANNEL_NAME = SIMPLE_ID
```

Similarly, the `EVENT` data type is itself of course referenced in the constructor `PrefixProcess` for a term-prefix process term, one of nineteen `PROCESS` constructors, most of which we omit here (*cf.* abbreviated abstract grammar in section 9.2.2):

```
data PROCESS
  = ...
  | PrefixProcess EVENT PROCESS Range
  ...
  deriving (Eq, Show)
```

12.2.2 The parsers

CSP-CASL's logic-specific parsers are found in the modules `CspCASL/Parse_CspCASL.hs` and `CspCASL/Parse_CspCASL_Process.hs` (divided as described at the start of section 12.2.1). Parsing CSP-CASL, like all parsing in HETS, is implemented using Parsec (see chapter 7); this basic functionality is augmented by HETS machinery supporting cross-cutting concerns such as handling *annotations*[3] and tracking the positions of syntactic elements for use in error reporting — as well as more mundane matters of parsing particular symbols such as semicolons and the like. HETS' key type for parsers is `AParser`: this data type, defined in `Common/AnnoState.hs`, builds on Parsec's monadic `GenParser` data type to provide annotation-collecting arbitrary-lookahead backtracking recursive-descent parsers. Every single function for parsing CSP-CASL uses `AParser`.

The code follows the concrete grammar given in section 9.3 remarkably closely, both in its coarse structure and its treatment of individual syntactic categories; this is an unsurprising and intended result of using Parsec. We now consider some interesting or instructive aspects.

12.2.2.1 Integration with HETS at the logic level

The 'top level' CSP-CASL parser is `cspBasicSpec`:

```
cspBasicSpec :: AParser st CspBasicSpec
cspBasicSpec = do chans <- option [] $ chanDecls
                  items <- processItems
                  return (CspBasicSpec chans items)
```

It takes no parameters, and returns a value of type `AParser st CspBasicSpec`, which we read as 'a parser which maintains some state (i.e. annotations) and returns an abstract syntax

[2] `http://repetae.net/computer/haskell/DrIFT/`

[3] CSP-CASL in its current form only uses annotations for comments, whereas in CASL annotations may carry extra 'side channel' semantic content, e.g. that a particular axiom is in fact a proof obligation; it is anticipated that later incarnations of CSP-CASL will also use annotations for such purposes.

term of type `CspBasicSpec`'. Note that the input text is completely abstracted away, as is usual with Parsec.

The body of the parser is a `do`-block encoding the intended largely sequential behaviour of the parser. There are three steps: first, parse the `CHAN-DECLS` part; second, parse the `PROC-ITEMS` part; finally, wrap the results in a `CspBasicSpec` value and package that into the monad. We consider `procItems` in section 12.2.2.2; `chanDecls` follows:

```
chanDecls :: AParser st [CHANNEL_DECL]
chanDecls = do choice [asKey channelS, asKey channelsS]
               cds <- chanDecl `sepBy` anSemi
               return cds
```

There are two things to notice about how channel declarations are handled. First: we use three standard combinators from the Parsec library:

- `option` (in `cspBasicSpec`) tries to apply its second argument; if that fails without consuming input, its first argument is returned. Here, it implements channel section optionality: if the specification doesn't begin with a `channels` or `channel` keyword, the list of channel declarations is empty.
- `choice` tries to apply the parsers in a list of parsers, in order, until one of them succeeds. Here, it implements the choice between the keyword `channels` and its singular form — in common with the choice of `ops` vs. `op`, `preds` vs. `pred`, etc. in CASL.
- `sepBy` applies zero or more occurrences of its first argument, separated by its second, returning the list of results. Here, it parses a channel declaration list consisting of zero or more individual channel declarations (not considered further here) separated by semicolons.

  (The `sepBy1` variant requires *at least one* application of its first argument; for CSP-CASL channels it seems reasonable to allow zero; there seems no strong reason to disallow an empty channel section which happens to begin with a `channels` keyword.)

Second: we use the `asKey` and `anSemi` parsers from HETS. The former parses the specified keyword, possibly followed by an annotation; the latter just parses a semicolon, possibly followed by an annotation; furthermore, *both* parsers are lookahead: if the parse fails, *they consume no input*. This last point is important in conjunction with the use of `option` as described above; otherwise, we would have to explicitly use the `try` combinator to force lookahead here (see below).

Finally, we ask: what calls `cspBasicSpec`? `CspCASL/Logic_CspCASL.hs` defines the CSP-CASL logic's interface to the rest of HETS, such that CSP-CASL basic specifications may be used in a heterogeneous setting. Every logic has such a module, consisting essentially of boilerplate with slots to be filled with references to various logic-specific types and functions. In the case of CSP-CASL and `cspBasicSpec`, the relevant part is:

```
instance Syntax CspCASL CspBasicSpec SYMB_ITEMS SYMB_MAP_ITEMS
  where parse_basic_spec CspCASL = Just cspBasicSpec
        parse_symb_items CspCASL = Just $ symbItems csp_casl_keywords
        parse_symb_map_items CspCASL = Just $ symbMapItems csp_casl_keywords
```

In outline: the data type `CspCASL` (not shown), when initiated with three parameters including `CspBasicSpec` (the top-level abstract syntax datatype, corresponding to `CSP-BASIC-SPEC` in section 9.2.1), is an instance of the `Syntax` type class; its basic specification parser is

`cspBasicSpec`. (The lines referring to symbols and symbol-maps in the above relate to structured specification, and are beyond the scope of this discussion.)

12.2.2.2 Combinators for arbitrary lookahead while parsing process items

Let us now consider process items. The relevant part of the concrete grammar (section 9.3) is very simple:

```
PROC-ITEMS      ::= PROC-ITEM
                  | PROC-ITEM PROC-ITEMS

PROC-ITEM       ::= PROC-DECL
                  | PROC-EQ
```

The implementation is remarkably similar, and an excellent illustration of the power of Parsec and higher-order programming in general:

```
procItems :: AParser st [PROC_ITEM]
procItems = many1 procItem

procItem :: AParser st PROC_ITEM
procItem = try procDecl
           <|> procEq
```

`procItems` returns a parser which parses a list of `PROC_ITEM`s; `procItem` returns a parser which parses just one. `procItems` simply calls Parsec's `many1` combinator: this returns a parser which performs one or more applications of its argument. The idea that 'the process part of a CSP-CASL specification is just a list of process items' is thus easily encoded.

Now, the process items are process equations and process declarations, freely mixed (linear visibility is enforced at the static semantic level); they are syntactically differentiated by an arbitrary lookahead (see section 9.3.1.2) performed by the `procItem` parser. The `try` combinator attempts to apply its single argument: in this case, it attempts to parse a process declaration. If it fails, it consumes no input: thus, arbitrary lookahead, clearly encoded and independent of the parser actually doing the looking ahead. The combinator `<|>` (pronounced 'predictive') applies its first argument, but if that fails *without consuming any input*, applies its second (see section 7.2.4). Thus, the overall behaviour of the `procItem` parser is that if `procDecl` fails, `procEq` is called (with no `try`: if that fails, we have a parse error).

`procDecl` (not shown) is somewhat similar to `cspBasicSpec`, above: its optional parameter list is implemented using `option`; the process' alphabet is parsed using HETS' `commaSep1` combinator (i.e. it is a comma-separated list with at least one element); etc. The key point regarding lookahead is that if the process item being parsed happens to be a process *equation*, i.e. if `procDecl` *will* fail, it has to parse a parameter name and an (optional) list of parameters before it *does* fail (by reading an equals sign when it expects a colon); this is arbitrary lookahead, because the parameter list can be any length — however, it seems unlikely that any realistic/otherwise tractable process would have a long enough parameter list for this to be a problem. The alternative to this lookahead would be to restructure `procItem` so it parses the name and parameter list and *then* differentiates, with a one-symbol lookahead; while this would not be *so* onerous, we argue that keeping our current implementation is a reasonable application of Hoare's famous maxim that "premature optimisiation is the root of all evil" [Hyd06] (on the other hand, this has implications for error messages — see section 13.3.3).

Symbol	Usage
channel	Channel section keyword
channels	Channel section keyword (alternative)
process	Process section keyword
RUN	'Run' process
CHAOS	'Chaos' process
DIV	'Divergence' process
SKIP	'Skip' process
STOP	'Stop' process
!	Channel send and channel nondeterministic send processes
?	Channel receive process
::	Various prefix processes
->	Various prefix processes
;	Sequential composition
|||	Interleaving
||	Syncrhonous and alphabetised parallel
[|	Generalised parallel
|]	Generalised parallel
[	Alphabetised parallel
]	Alphabetised parallel
[]	External choice and external prefix choice
|~|	Internal choice and internal prefix choice
\	Hiding process
[[	Renaming process
]]	Renaming process

Table 12.1: CSP-CASL reserved keywords and symbols

12.2.2.3 Keyword and identifier parsing; encoding syntactic range

Consider the `term_event` parser:

```
term_event :: AParser st EVENT
term_event = do t <- CASL.Formula.term csp_casl_keywords
                return (TermEvent t (getRange t))
```

This encodes the first option of the concrete syntax rule for `EVENT` (see section 12.2.1), to parse an event which consists simply of a CASL term. There are three things to note:

- Part of the work is done by existing HETS machinery, in the form of the `term` parser from `CASL/Formula.hs`.
- `term` is parametrised by a list of logic-specific keywords, disallowed as identifiers in the term; in this case it is `csp_casl_keywords`, defined in `CspCASL/CspCASL_Keywords.hs`. Most (but not all: see section 12.3) CSP-CASL-specific keywords and symbols are defined in that same module, including, e.g., `channelS` and `channel` seen in section 12.2.2.1. See table 12.1 for a full list of CSP-CASL-specific reserved words.
- The call `getRange t` takes the `TERM` value returned by `term` and returns a `Range` value encoding the term's position in the input text; this is then simply stored in the `TermEvent`'s `Range` slot (see section 12.2.1).

`term_event` is a simple example of how syntactic range information is stored in a CSP-CASL abstract syntax value; a more interesting example is seen in `chan_nondet_send`, parsing a channel nondeterministic send event (e.g. '$c!x :: s$' in $c!x :: s \rightarrow SKIP$):

```
chan_nondet_send :: AParser st EVENT
chan_nondet_send = do cn <- varId csp_casl_keywords
                      asKey chan_sendS
                      v <- var
                      asKey svar_sortS
                      s <- sortId csp_casl_keywords
                      return (ChanNonDetSend cn v s (compRange cn s))
```

Here, `varId` and `sortId` are HETS-provided parsers for CASL variable and sort names respectively, again parametrised with a list of disallowed keyword strings. `chan_SendS` and `svar_sortS`, from `CspCASL_Keywords.hs` are the symbols `!` and `::` respectively.

Now, a channel nondeterministic event is composed of five constituent syntactic entities; as such, the `ChanNonDetSend` value's `Range` is computed in terms of the ranges of those consistuents. Specifically, our `compRange` function takes two values with `Ranges` (specifically, values of type class `PosItem`) and returns a `Range` covering their total span (i.e. its start is the start of the first parameter; its end is the end of the second). This range can then be usefully exposed in error and debug messages such as the following (where, e.g., `15.13` means 'line 15, column 13').

```
*** Error /tmp/examples/eg1.cspcasl:15.13-15.28,
```

Chapter 13, on sample runs of our tool, presents a full range of further examples.

12.2.2.4 Encoding of precedence in grammar, and corresponding parser structure

Finally, let us consider the implementation of operator precedence in CSP-CASL process terms (see section 8.2.1). This is, in fact, largely a matter of grammar design: the concrete grammar in section 9.3 encodes operator precedences in a standard manner [ALSU06]. However, as noted in section 9.3.1.1, the concrete grammar for process terms is *left recursive*, which is a problem for a recursive descent parser. Consider the following concrete grammar fragment, encoding the difference in precedence level of (internal/external) choice and sequential composition.

```
CHOICE-PROC      ::= SEQ-PROC
                   | CHOICE-PROC [] SEQ-PROC
                   | CHOICE-PROC "|~|" SEQ-PROC
```

A direct implementation in Parsec of this rule would look something like this:

```
choice_proc lp = try seq_proc
                 <|> try (do lp <- choice_proc
                             asKey external_choiceS
                             rp <- seq_proc
                             return (ExternalChoice lp rp (compRange lp rp))
                 <|> do lp <- choice_proc
                        asKey internal_choiceS
                        rp <- seq_proc
                        return (InternalChoice lp rp (compRange lp rp))
```

(Note use of `compRange` again, this time covering entire process terms). The problem here is that if that first call to `seq_proc` fails, the next thing tried is a call to `choice_proc`, leading immediately into a nonterminating recursion. To solve this problem, we apply a standard transformation to the grammar, yielding equivalent but non-left-recursive rules, as demonstrated for `CHOICE-PROC` in section 9.3.1.1. The transformed concrete grammar fragment is:

```
CHOICE-PROC    ::= SEQ-PROC CHOICE-PROC1

CHOICE-PROC1   ::= [] SEQ-PROC CHOICE-PROC1
                 | "|~|" SEQ-PROC CHOICE-PROC1
                 | epsilon
```

It is *this* form of the grammar (not explicitly written in its entirety in this thesis) which is implemented in our tool; for `CHOICE-PROC`, we have:

```
choice_proc :: AParser st PROCESS
choice_proc = do sp <- seq_proc
                 p <- choice_proc' sp
                 return p

choice_proc' :: PROCESS -> AParser st PROCESS
choice_proc' lp = do asKey external_choiceS
                     rp <- seq_proc
                     p <- choice_proc' (ExternalChoice lp rp (compRange lp rp))
                     return p
                  <|> do asKey internal_choiceS
                     rp <- seq_proc
                     p <- choice_proc' (InternalChoice lp rp (compRange lp rp))
                     return p
                  <|> return lp
```

This is a fairly direct encoding of the transformed grammar fragment shown above; the most interesting thing to note is that `choice_proc'` (corresponding to the `CHOICE-PROC1` rule) is parametrised with the `PROCESS` returned by the `sequence_process` parser: that value needs to be passed 'forwards' for inclusion in the overall result of the parser — as either the left hand side of an `ExternalChoice` or `InternalChoice` value, or just on its own if those parsers fail (note use of `asKey` which, as noted above, consumes no input on failure — hence, no `try` here).

Finally, it is worth noting that while Parsec's `ParsecExpr` module provides a simple mechanism for building expression parsers in which precedence and associativity are specified declaratively, this does not provide a simple solution for parsing process terms; the problem is that some process terms (e.g. generalised and alphabetised parallel) involve complex multi-part operators which are not expressible using the `ParsecExpr` mechanism. Thus, despite some promising exploratory work on a restricted subset of the grammar, it is in fact necessary to encode the final grammar manually.

12.3 Implementation of pretty printing

Pretty printing of CSP-CASL specifications is implemented using HETS' standard mechanisms, based on [Hug95], and reuses CASL's existing pretty printing facilities where possible. The

basic machinery, defined in `Common/Doc.hs` and `Common/DocUtils.hs`, consists of the data type `Doc` representing pretty-printable documents; various atomic `Doc` values; various combinators for composing `Doc`s; and the `Pretty` type class, to be instantiated by any pretty-printable type.

In this context, a pretty printer for a type is implemented by declaring the type to be an instance of the `Pretty` type class, and writing the `pretty` function thus required, to transform a value of the type into a corresponding `Doc` value. When a logic is integrated into HETS as outlined earlier in this chapter, specifications written in that logic may then be pretty printed to ASCII, LaTeX or HTML by passing appropriate command-line switches when running HETS (all CSP-CASL examples in this thesis are typeset by HETS; see also section 13.2).

As a representative example illustrating some key points, consider the pretty printer functions for process items:

```
instance Pretty PROC_ITEM where pretty = printProcItem

printProcItems :: [PROC_ITEM] -> Doc
printProcItems ps = foldl ($+$) empty (map pretty ps)

printProcItem :: PROC_ITEM -> Doc
printProcItem (Proc_Decl pn args alpha) =
    (pretty pn) <> (printArgs args) <+> colon <+> (pretty alpha) <+> semi
        where printArgs [] = empty
              printArgs a = parens $ ppWithCommas a
printProcItem (Proc_Eq pn p) = (pretty pn) <+> equals <+> (pretty p)
```

The first line declares `PROC_ITEM` to be pretty-printable, using the `printProcItem` function.

`printProcItems` prints a list of `PROC_ITEM`s; it maps `pretty` over every element of that list then, starting with the empty `Doc`, folds the list of `Doc`s thus produced down to a single `Doc` value, combining the elements pairwise using the `$+$` combinator; this last combines two `Doc` values into a new one, inserting a newline between them.

`printProcItem`, to handle a single `PROC_ITEM` value, illustrates the use of pattern matching over a type's constructors, as mentioned in section 12.2.1. There are two cases: one for `Proc_Decl` values (process declarations), and one for `Proc_Eq`s (process equations). The main points to note are:

- Calls to pretty printers of the various constituent values (e.g. process name, process argument list), none of which are shown here.
- Use of the <+> and <> combinators, to place two `Doc`s besides each other (with and without a space, respectively); *cf.* `$+$`, above.
- The `ppWithCommas` combinator, to pretty print a list of values, separated by commas.
- The atomic `Doc` values `colon`, `semi`, and `equals` representing those symbols.

On the whole, defining pretty printing in HETS is reasonably straightforward. The only real difficulty encountered relates to symbols. The issue is that in many cases, a symbol's LaTeX representation is very different from its plain text one; a CSP-CASL-specific example is the external choice operator, whose ASCII form is `[]`, but whose LaTeX is `\Box`, rendered as □.

Now, HETS provides a mechanism addressing such symbols: in `Doc.hs`, the value `latexSymbols` maps ASCII representations of symbols to their LaTeX equivalents; an output-context-aware pretty printer for such symbols may then be created using the `symbol` function — so in our example, rather than render external choice as:

```
text external_choiceS
```

we render it as:

```
symbol external_choiceS
```

which results in a render time lookup in `latexSymbols` if and only if the desired output is LaTeX.

This works well, up to the point of ASCII symbol-clash, at which point problems emerge. The root cause of these problems is that `latexSymbols` is a map from ASCII representations to LaTeX equivalents — thus, a given ASCII key can have only one corresponding LaTeX value. With CSP-CASL this causes a problem with alphabetised parallel, and results in a divergence between our pretty printed format and CSP's. Specifically:

- In CSP, alphabetised parallel is pretty-printed with a single | in the middle, e.g. $p \,[\![x \mid y]\!]\, q$, whereas the machine-readable a version, uses two `|` characters, e.g. `p[x||y]q`.
- In CSP-CASL, we have $p[\![x||y]\!]q$ for pretty printed, and `p[x||y]q` for machine readable, i.e. two bars in both versions.

The reason for this divergence arises from the deficiency noted above: we could normally use the `latexSymbols` machinery to define a symbol whose ASCII differs from its LaTeX, but in this case that is impossible because of a clash with sychronous parallel, whose ASCII representation is also `||`, but whose LaTeX is `\mid\mid`. `latexSymbols` may include a map for synchronous parallel, or for alphabetised parallel, but not for both; thus, we must choose between a single bar or two bars in the LaTeX version, where our choice applies to both cases. As the two bars seem more fundamental to synchronous parallel than the single bar seems fundamental to alphabetised parallel, we make the compromise described here.

A possible solution to this problem would be raise our symbol handling from the syntactic to the semantic level: we would define a type, a value of which would contain all of the various representations of a particular symbol; thus, we would no longer refer to a symbol primarily via its ASCII representation (as we do now), but rather via a value of this new type. This would decouple ASCII and LaTeX, and in particular allow a given ASCII string to have multiple associated LaTeX strings, depending on the actual *semantic* content intended. This would solve the problem described above, but would require non-trivial refactoring of HETS, beyond its CSP-CASL-specific parts.

12.4 Implementation of static semantics

The CSP-CASL static semantics presented in chapters 10 and 11 is implemented in the module `CspCASL/StatAna.hs`, whose functions transform abstract syntax values to values of semantic types defined mostly in `CspCASL/SignCSP.hs`. The analysis functions also perform the various checks required, possibly raising errors which halt further computation on the specification, or warnings, which allow computation to continue but indicate possible cause for concern. We present the features and techniques by example, considering the analysis of several particular syntactic categories.

12.4.1 Representation and analysis of CSP-CASL basic specifications

A CSP-CASL basic specification is represented using types defined in `CspCASL/SignCSP.hs`:

```
type CspCASLSign = Sign () CspSign

data CspSign = CspSign { chans :: ChanNameMap
                       , procSet :: ProcNameMap
                       } deriving (Eq, Show)
```

A `CspCASLSign` value, representing a CSP-CASL signature, is a CASL basic signature (`Sign`, defined in `CASL/Sign.hs`) extended with a `CspSign` value representing a CSP signature — see section 10.2.1. A `CspSign` value contains a map from channel names to CASL sort names, and a map from process names to process profiles (we omit the details of process profiles here):

```
type ChanNameMap = Map.Map CHANNEL_NAME SORT
type ProcNameMap = Map.Map PROCESS_NAME ProcProfile
```

The top-level static analysis function, `ana_BASIC_CSP`, analyses a `CspBasicSpec` value (a basic specification in the abstract syntax) in the context of a CSP-CASL signature, wrapped in an instance of the `State` monad (see below). This function is 'plugged in' to the HETS machinery in a manner similar to that described for abstract syntax and parsing, earlier in this chapter.

```
ana_BASIC_CSP :: CspBasicSpec -> State CspCASLSign ()
ana_BASIC_CSP cc = do checkLocalTops
                      mapM anaChanDecl (channels cc)
                      mapM anaProcItem (proc_items cc)
                      return ()
```

Thus: there is one `anaChanDecl` call per `CHAN-DECL` (see section 10.3.2), and one `anaProcItem` call per `PROC-ITEM` (see section 10.3.3). The former is worthy of close consideration.

12.4.2 Analysis of channel declarations

The purpose of the function `anaChanDecl` (below) is to transform syntactic `CHAN-DECL` values into (modifications of) semantic `ChanNameMap` values (see section 12.4.1). The function corresponds reasonably closely to the `CHAN-DECL` rule in section 10.3.2, except that new elements are added to the channel name map not here, but in `anaChannelName` (discussed below); this is a specific example of a general point of interest, namely that there is an *almost* direct translation from static semantics rules to their implementation, but it is not as direct as that for syntax. This is unsurprising: Parsec provides (rather successfully) a domain specific language for recursive descent parsing, enabling our designs to be implemented directly — that is, after all, the point of a domain specific language [Hud96]. For our static analysis, we have no such DSL, and must implement the rules in more 'traditional' Haskell, which leads to heavy use of monadic folds, etc., in code where the most natural place to do something (e.g. add entries to a channel map) is not necessarily the most natural place to specify that in the design. Overall, however, the coarse structure is at least essentially identical.

```
anaChanDecl :: CHANNEL_DECL -> State CspCASLSign ()
anaChanDecl (ChannelDecl chanNames chanSort) = do
    checkSorts [chanSort]
```

```
sig <- get
let ext = extendedInfo sig
    oldChanMap = chans ext
newChanMap <- Monad.foldM (anaChannelName chanSort) oldChanMap chanNames
vds <- gets envDiags
put sig { extendedInfo = ext { chans = newChanMap }
        , envDiags = vds }
return ()
```

The first line of the body calls `checkSorts` (a CASL static analysis function) which, given a list of SORTs, adds an error to the state for every sort not known in the signature's CASL part; here, it checks that the declared channel sort is known. This is exactly as seen in the CHAN-DECL rule.

The `get` call, and the corresponding `put` at the end of the function, respectively get and put a `CspCASLSign` value stored in a `State` monad [Wad95] (`anaChanDecl`'s type signature tells us that the state stored is of type `CspCASLSign`). The `CspCASLSign` 'in the state' at the start of the function is thus bound to the name `sig`. Later, the `put` call replaces that state with an copy of `sig` whose `extendedInfo` and `envDiags` fields are updated to new values. Such a get/put pair is typical of a static analysis function which manipulates the signature; a function which needs only to examine the signature (e.g. to look up a channel's sort) need only perform the `get`, of course.

Having acquired the signature, `extendedInfo` projects the `CspSign` part, bound to the name `ext`, and finally the signature's channel map is bound to the name `oldChanMap`. This forms the basis of an updated channel map, constructed by folding (`anaChannelName chanSort`) over the list of CHAN_NAMEs in `chanNames`. Consider `anaChannelName`'s type signature:

```
anaChannelName :: SORT -> ChanNameMap -> CHANNEL_NAME ->
                  State CspCASLSign ChanNameMap
```

Given a sort, a channel name map, and a channel name, `anaChannelName` returns a new channel name map, computed in a state monad containing a CSP-CASL signature. We omit its definition here, but it should be clear that it returns a copy of the input map, with a new entry mapping the channel name to the sort. This, then, implements the CHAN-NAMES and CHAN-NAME rule in section 10.3.2.

Any duplicated channel names cause `anaChannelName` to raise errors and warnings (not shown); `anaChanDecl` then collects those diagnostic messages in a value bound to the name `vds`, and stores them in the modified signature. Such is the mechanism for storing diagnostic messages in HETS static analysis; if, at the end of static analysis, there are any errors, they will be reported *en masse*, and execution will cease. We will see an example of the mechanism for *raising* error messages shortly.

12.4.3 Analysis of process terms

`anaProcTerm` analyses process terms, whose type signature,

```
anaProcTerm :: PROCESS -> ProcVarMap -> ProcVarMap -> State CspCASLSign
    CommAlpha
```

corresponds closely with the PROCESS rules' general form (see start of section 10.4): input consists of a process term, and global/local variable sets (here called ProcVarMaps); the required signature context comes from the state monad. The function yields the process term's constituent alphabet; anaProcEq, which analyses a process *equation*, checks this against the process' permitted alphabet (see section 10.2.3) — we examine the similar check in the context of alphabetised parallel, shortly.

The body of anaProcTerm is a large case distinction, with one case per PROCESS constructor; we concentrate on just one case, namely alphabetised parallel:

```
anaProcTerm proc gVars lVars = case proc of
  ...
  AlphabetisedParallel p esp esq q _ ->
    do pComms <- anaProcTerm p gVars lVars
       pSynComms <- anaEventSet esp
       checkCommAlphaSub pSynComms pComms proc "alphabetised parallel, left"
       qSynComms <- anaEventSet esq
       qComms <- anaProcTerm q gVars lVars
       checkCommAlphaSub qSynComms qComms proc "alphabetised parallel, right"
       return (pComms `S.union` qComms)
  ...
```

Recalling that such a process term is written $p|[es_p||es_q]|q$, and comparing with the ALPHAPAR-PROC rule in section 10.4.13, this is easily understood: we (recursively) analyse the left hand process term p, binding its constituent alphabet to the name pComms; we analyse the left hand event setp esp, binding its alphabet to pSynComms; then we check that the latter is included in the fomer (under subsort closure); then we repeat for the right hand side. The overall product of the analysis is the union of the process terms' constituent alphabets. The interesting part, of course, is checkCommAlphaSub.

12.4.4 Analysis of communication alphabets

```
checkCommAlphaSub :: CommAlpha -> CommAlpha -> PROCESS -> String ->
                     State CspCASLSign ()
checkCommAlphaSub sub super proc context = do
  sig <- get
  let extras = ((closeCspCommAlpha sig sub) `S.difference` (closeCspCommAlpha sig super
      ))
  if S.null extras
    then do return ()
    else do let err = ("Communication alphabet subset violations (" ++
                       context ++ "): " ++ (show $ S.toList extras))
            addDiags [mkDiag Error err proc]
            return ()
```

This function, which is also called for the similar check on process equations, checks that the subsort closure of one communication alphabet (sub) is a subset of the subsort closure of another (super). The set of elements violating that requirement (collected via set difference) is reported via a Diagnosis value (Common/Result.hs), added to the state's collection of diagnostic messages.

Specifically, the error message is represented by the `Diagnosis` value returned from the `mkDiag Error err proc` call. `Error` indicates that the diagnosis is an error (as opposed to a `Warning`, `Hint` or `Debug`). `err` is the error string, built from a context-specific string (see analysis of `AlphabetisedParallel`, above) and a string representation of the set of `CommType` values violating the requirement. Finally, `proc` is the process term containing the violation, and will be included in the error output for the user — in particular, its range will be obtained by a `getRange` call, and printed out. Thus, any value to be included in a diagnostic message in this manner must implement the `PosItem` type class (see section 12.2.2.3). In the case of `AlphabetisedParallel`, `proc` is the alphabetised parallel process term; in the case of a process equation check, it will be the entire process term on the right hand side of the equation.

For example, the following specification contains a violation of the alphabet subset condition on an alphabetised parallel process: the constituent alphabet of the named process term Q is Q's permitted communication alphabet, T, but the alphabetised parallel operator requires that the subsort closure of the right-hand $\{S\}$ event set be a subset of that set — which is obviously not the case (though if we had $S < T$ it would be).

logic CSPCASL
spec ALPHAERROR =
    data sorts S, T
    process $P : S, T$;
        $Q : T$;
        $P = SKIP \,||\, (\, RUN\, (\, S\,)\, [\![S \,||\, S]\!]\, Q\,)$
end

The output by HETS is then:

```
[gimbo@mane Hets] ./hets CspCASL/test/alpha_error.cspcasl
logic CspCASL
Analyzing spec AlphaError
*** Error CspCASL/test/alpha_error.cspcasl:6.26-6.45,
Communication alphabet subset violations (alphabetised parallel, right): [S]
'RUN ( S ) [ S || S ] Q'
hets: user error (Stopped due to errors)
```

The error message tells us we have an alphabet subset violation, that it occurs on the right hand side of an alphabetised parallel process term, and that the list of offending communication types contains just one element, `S`. The offending process is then printed in its entirety (note that only the alphabetised parallel process term is printed, not the entire process equation); its location is shown on the line beginning `***`, i.e. line 7, columns 18 to 32, of `/tmp/alpha_error.cspcasl`. See chapter 13 for more example system runs.

12.4.5 Analysis of CASL terms

Analysis of CASL terms is an interesting and non-trivial example of integration with HETS' existing static analysis machinery. CASL terms occur in CSP-CASL prefix process terms (representing a value to be communicated) and named process references (representing values for global parameters of the named process). We will consider the latter.

The `NAMED-PROC` rule given in section 10.4.2 checks that the number of parameters is as expected, and then for each parameter attempts to cast it to the appropriate sort, as described

in section 10.2.4.2. The relevant part of `anaNamedProc`, the function which implements this rule, is as follows (full version omitted):

```
    if (length terms) == (length varSorts)
      then do mapM (anaNamedProcTerm procVars) (zip terms varSorts)
              return permAlpha
      else do let err = "wrong number of arguments in named process"
              addDiags [mkDiag Error err proc]
              return S.empty
```

The term list length check is obvious; the terms and expected sorts are zipped together[4] and passed individually (along with `procVars`, the combined global and local variable sets), to the function `anaNamedProcTerm`, which checks a single term/sort pair:

```
anaNamedProcTerm :: ProcVarMap -> ((TERM ()), SORT) -> State CspCASLSign ()
anaNamedProcTerm pm (t, expSort) = do
    mt <- anaTermCspCASL pm t
    case mt of Nothing -> return () -- CASL term analysis failed
               (Just at) -> do ccTermCast at expSort -- attempt cast; don't need result
                               return ()
```

`anaNamedProcTerm` is quite simple: it calls `anaTermCspCASL`, to perform CASL static analysis of the term and, if that was successful, calls `ccTermCast`, which attempts to cast it to the required sort. The result of that cast is itself a term (see below), but that value is not required here: we just need to perform the cast, so it may raise an error if appropriate. `anaTermCspCASL` is a thin and uninteresting wrapper around `anaTermCspCASL'`:

```
anaTermCspCASL' :: CspCASLSign -> (TERM ()) -> Result (TERM ())
anaTermCspCASL' sig t = do
    let allIds = unite [mkIdSets (allOpIds sig) $ allPredIds sig]
        mix = emptyMix { mixRules = makeRules (globAnnos sig) allIds }
    resT <- resolveMixfix (putParen mix) (mixResolve mix)
                (globAnnos sig) (mixRules mix) t
    oneExpTerm (const return) sig resT
```

`anaTermCspCASL'` takes a CSP-CASL signature and a term, and returns a (fully qualified) term in the `Result` monad (which is, above this level, integrated with the `State` monad). Now, all of the work here is being done by existing CASL machinery; it is complicated mainly by the necessary presence of *mixfix analysis*, in which terms written in 'mixfix notation' (which "generalises infix, prefix, and postfix notation to allow arbitrary mixing of argument positions and identifier tokens" [BM04, §3.1]), identified but left unresolved during parsing, are resolved.

The details of this process are largely opaque to us: we extract CASL operation and predicate names from the signature (`allOpIds sig`, etc.) and bind them to the name `allIds`; we create a 'mixfix analysis context' `mix` from that; `resolveMixFix` then performs any mixfix resolution required on the term, and finally, `oneExpTerm` tests that the term can be uniquely resolved in the face of overloading; the result of `oneExpTerm` is a fully qualified term, which `anaTermCspCASL'` then returns.

Casting that fully qualified term to the desired sort is more straightforward — a fairly direct encoding of the case distinction in the definition of $cast : FQTerm \times Sort \rightarrow FQTerm$, from section 10.2.4.2.

[4] `zip` combines two lists into a list of pairs; its type signature is `zip : [a] -> [b] -> [(a,b)]`

`ccTermCast` returns a value of type `Maybe (TERM ())` because the cast can fail: see the last case, which returns `Nothing`. Conversely, if the fully qualified term `t`'s sort `termSort` is the same as the desired sort `cSort`, we just return t (or rather, `Just t`, where `Just` indicates success, vs. `Nothing`'s failure); otherwise, we check the CASL signature's subsort relation to see if `termSort < cSort` or vice versa; in both cases, we wrap the term in another term, using the `Sorted_term` constructor for an upcast, and the `Cast` constructor for a downcast. Note the use of `getRange` again to pass the input term's `Range` value to the `TERM` constructors.

```
ccTermCast :: (TERM ()) -> SORT -> State CspCASLSign (Maybe (TERM ()))
ccTermCast t cSort =
    if termSort == (cSort)
      then return (Just t)
      else do sig <- get
              if Rel.member termSort cSort (sortRel sig)
                then do let err = "upcast term to " ++ (show cSort)
                        addDiags [mkDiag Debug err t]
                        return (Just (Sorted_term t cSort (getRange t)))
                else if Rel.member cSort termSort (sortRel sig)
                       then do let err = "downcast term to " ++ (show cSort)
                               addDiags [mkDiag Debug err t]
                               return (Just (Cast t cSort (getRange t)))
                       else do let err = "can't cast term to sort " ++ (show cSort)
                               addDiags [mkDiag Error err t]
                               return Nothing
        where termSort = (sortOfTerm t)
```

12.5 Automated testing

We conclude this chapter with a brief discussion of the issue of testing. This section is necessarily brief as our testing strategy, at least within the context of HETS, has been essentially 'ad hoc' and 'by hand': tests have been written and performed individually when deemed necessary to test a particular feature, typically the one most recently implemented.

We would much prefer that this were not the case: a good set of automated tests gives valuable assurance that a codebase does what is required, and continues to do so over time and continued development. Working from a formal specification, as we have done here, certainly helps raise assurance that what is required, and only that, has been implemented (which is, of course, one of the intentions of unit testing), and Haskell's all-encompassing type system virtually forces one to write code which is *in some sense* correct — however, automated tests would still be extremely desirable.

The basic problem is that HETS does not currently provide a framework for automated testing: there are some test specifications in a particular directory, where `make test` will check that each is handled by the tool without errors, but this is not very detailed and ignores the issue of *negative testing*, where we test not just the positive cases ('yes, it worked as expected') but also the negative ones ('yes, it failed as expected'). For a parser and static analyser, negative tests are arguably as important as positive ones.

Now, at an earlier stage in this project, before our codebase was integrated with HETS, we did in fact have a framework for automatically performing positive and negative tests on the parsing and pretty printing functions. The framework's basic data unit was a test case containing:

1. a name for the test case ('`Prefix skip`' in the example below);
2. the name of the parser function with which to process the fragment ('`Process`' below);
3. the sense of the test: ++ (positive) or -- (negative);
4. the expected output, where a positive test would expect an ASCII pretty print of the abstract syntax term parsed, and a negative test would expect an error message;
5. a dividing line consisting of 10 dashes;
6. the fragment of CSP-CASL to be tested.

Note that this was an entirely standalone program, independent of HETS. As such, an entire CSP-CASL specification could not be tested, because there was no way to deal with the data part; similarly, CASL entities such as sort names, terms, and formulae, could only be simple identifiers. Nonetheless, such a framework proved useful at a time when the parser was in a daily state of flux.

For example, the following is a simple positive test case targetting the process term parser:

```
Prefix skip
Process
++
a -> SKIP || SKIP
----------
a -> SKIP || SKIP
```

Running the tool on this test case produces the following output:

```
Performing tests
Prefix skip (++Process) passed
```

This is deliberately terse: the intent is that a large batch of tests which pass should do so fairly quietly — only test failures should stand out.

A negative test case, where we expect the parse to fail, is:

```
Prefix skip
Process
--
"Prefix skip" (line 1, column 9):
unexpected "|"
expecting "%" or words
----------
a -> [] || SKIP
```

Here we expect the error output shown before the CSP-CASL source; a successful test run looks very similar to the one above:

```
Performing tests
Prefix skip (--Process) passed
```

Naturally, a test which fails should report details of the failure. Here we force a failure by changing the above negative test so the CSP-CASL source will, in fact, parse:

```
Prefix skip
Process
--
"Prefix skip" (line 1, column 9):
unexpected "|"
expecting "%" or words
----------
a -> SKIP || SKIP
```

Now the output tells us that the test failed, with 'unexpected parse success' — our negative test failed, because the parse was successful, contrary to expectations:

```
Performing tests
Prefix skip (--Process) failed - unexpected parse success
-> expected:
"Prefix skip" (line 1, column 9):
unexpected "|"
expecting "%" or words
-> got:
a -> SKIP || SKIP
```

Note that the failure report tells us what was expected, and what was obtained — in this case, because the parse succeeded, the (unexpected/undesirable) output is the process term pretty printed in ASCII.

Now, clearly there is no technical reason why such a test framework could not be employed in HETS; however, this has not yet been done. There are two possible approaches:

1. An external tool which calls `hets`, collects its output, examines output files, etc. — such a tool could be reasonably quickly developed in a scripting language such as Python, for example.
2. An integrated testing tool written in Haskell which calls the appropriate HETS functions directly in order to run tests and collect and analyse the outputs. This is essentially how the framework above works, which is the main reason why it was not carried forward when our work was integrated with HETS: the HETS codebase is decidely non-trivial in nature, and it was far from obvious how to achieve this in a timely manner.

Reinstating our automated testing framework is certainly desirable future work — not only for the CSP-CASL tools but also, we would argue, for the rest of HETS too.

An interesting possible enhancement would be to take the output of the ASCII pretty printer and put *that* back through the parser; this would enable us to ask two pertinent questions:

1. Does the output of the pretty printer parse?
2. Is the output of the pretty printer a fixed point, with respect to repeated parsing/pretty printing?

Naturally, we would like the answer to both of these question to be 'yes'.

Part III

Evaluation & Conclusion

Chapter 13

Selected System Runs

Contents

In this chapter we demonstrate our implementation by presenting selected runs of the system on input files demonstrating various aspects of interest.

13.1 Screenshot: HETS in action

In figure 13.1 we see the obligatory screenshot of the tool in action, processing two of the examples presented later in this chapter. In the foreground terminal, `hets` has just processed `tcs4.cspcasl` (see section 13.2.4), with debug output enabled via the `-v5` command line option. In the other terminal, we have just built `hets` and run it on `tcs3.cspcasl` (see section 13.2.3). In the background lurks emacs, showing the source of `tcs3.cspcasl` and some static analysis code.

Note that HETS' ordinary mode of operation includes a GUI for proof management; however, as CSP-CASL support extends thus far only to parsing and static analysis, we do not invoke the GUI, and are only interested in CLI output.

13.2 Examples from [Rog06]

In chapter 4, we illustrated CSP-CASL with four examples first seen in [Rog06]. These provide simple positive tests, i.e. specifications which successfully parse and pass static analysis — and an opportunity to demonstrate some of `hets`' command line options.

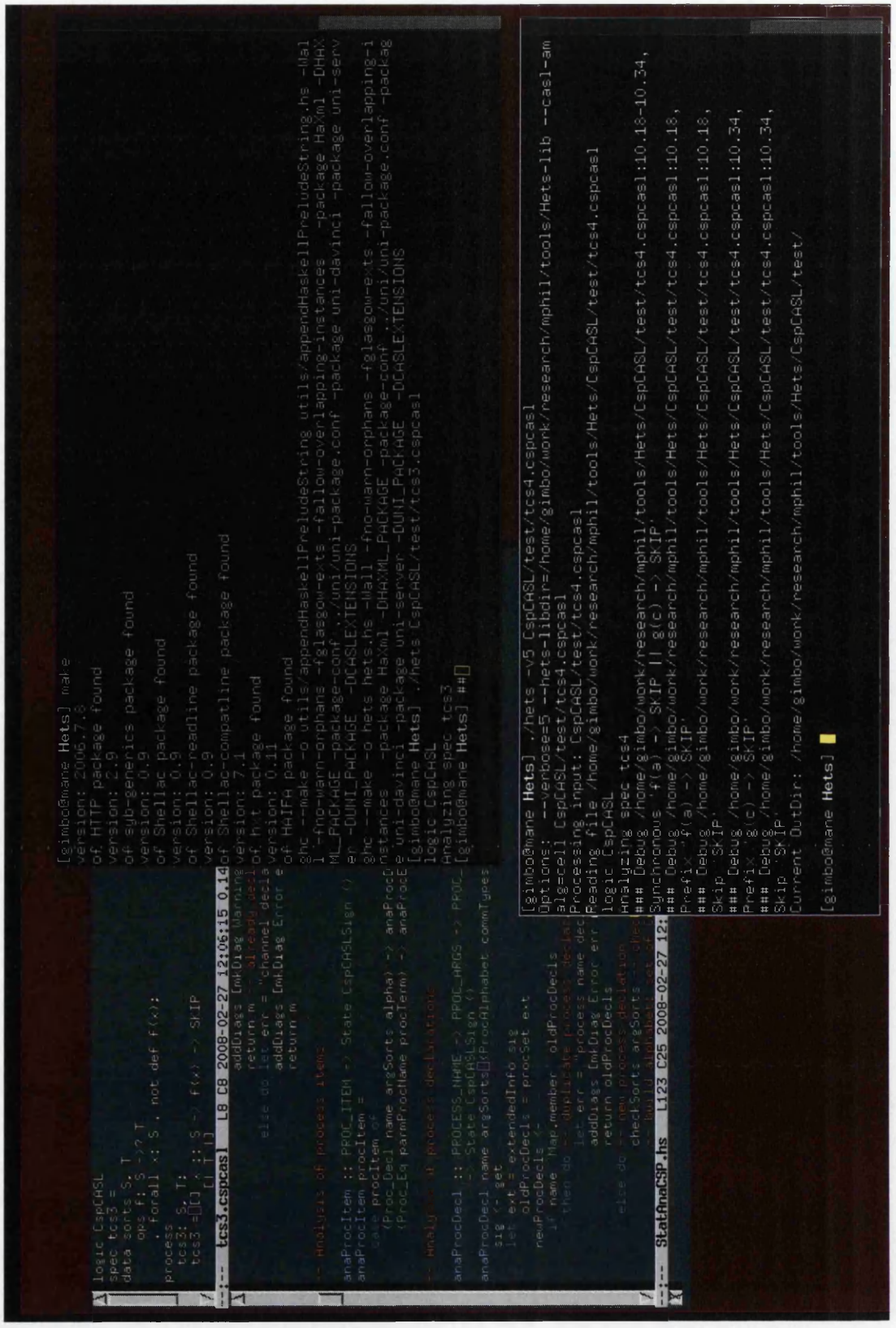


Figure 13.1: Screenshot of HETS in action, parsing CSP-CASL

13.2.1 tcs1.cspcasl — almost the simplest possible

This example illustrates successful parse and analysis of a very simple specification, almost the simplest one could write.

Input:

```
logic CspCASL
spec tcs1 =
  data sort S, T
       ops c: S; d: T
  process
    tcs1: S, T
    tcs1 = c -> SKIP || d  -> SKIP
```

Output:

```
[gimbo@mane Hets] ./hets CspCASL/test/tcs1.cspcasl
logic CspCASL
Analyzing spec tcs1
```

13.2.2 tcs2.cspcasl — debug output

This example illustrates debug output, enabled by invoking the `hets` executable with the `-v5` option (meaning 'verbosity = 5'; `-v3` activates warnings but not debug — see below).

Input:

```
logic CspCASL
spec tcs2 =
data sorts S < T
     ops c: S; d: T;
     .   c = d;
process
    tcs2: S, T;
    tcs2 = c -> SKIP || d -> SKIP
```

Output:

```
[gimbo@mane Hets] ./hets -v5 CspCASL/test/tcs2.cspcasl
Options:  --verbose=5 --hets-libdir=../Hets-lib --casl-amalg=cell
    CspCASL/test/tcs2.cspcasl
Processing input: CspCASL/test/tcs2.cspcasl
Reading file CspCASL/test/tcs2.cspcasl
logic CspCASL
Analyzing spec tcs2
### Debug CspCASL/test/tcs2.cspcasl:8.17-8.30,
Synchronous 'c -> SKIP || d -> SKIP'
### Debug CspCASL/test/tcs2.cspcasl:8.17,
Prefix 'c -> SKIP'
### Debug CspCASL/test/tcs2.cspcasl:8.17,
Skip 'SKIP'
### Debug CspCASL/test/tcs2.cspcasl:8.30,
Prefix 'd -> SKIP'
### Debug CspCASL/test/tcs2.cspcasl:8.30,
Skip 'SKIP'
Current OutDir: CspCASL/test/
```

Discussion: Our implementation currently adds a `Debug` diagnostic message for every process term traversed: the order of the debug messages demonstrates the order of traversal: a node is visited, then its children, from left to right.

13.2.3 tcs3.cspcasl — pretty printing ASCII

This example contains a considerably more complex process term, with external prefix choice (note use of the local variables `x` and `y` in CASL terms), term prefix (with a CASL function), generalised parallel, and conditional. ASCII pretty print is enabled with the `-o pp.het` option.

Input:

```
logic CspCASL
spec tcs3 =
data sorts S, T
     ops f: S ->? T
     . forall x: S . not def f(x);
process
  tcs3: S, T; tcs3 = [] x :: S -> f(x) -> SKIP [| T |]
                     [] y :: T -> (if def y then SKIP else STOP)
```

Output:

```
[gimbo@mane Hets] ./hets -o pp.het CspCASL/test/tcs3.cspcasl
logic CspCASL
Analyzing spec tcs3
```

Discussion: This produces the output file `CspCASL/test/tcs3.cspcasl.pp.het`, with the following contents (long line wrapped by hand):

```
library library

logic CspCASL

spec tcs3 =
     data sorts S, T
          op    f : S ->? T
          . forall x : S . not def f(x)
     process tcs3 : S, T ;
             tcs3 = [] x :: S -> f (x) -> SKIP [| T |]
                                [] y :: T -> if def y then SKIP else STOP
end
```

13.2.4 tcs4.cspcasl — pretty printing LaTeX

This example illustrates LaTeX pretty print, enabled with the `-o pp.tex` option.

Input:

```
logic CspCASL
spec tcs4 =
data sorts A, B, C < S
     ops a: A; b1, b2: B; c: C;
         f: A ->? A; g: C->? C
```

```
    .   a = b1 . b2 = c
    .   forall x: A . not def f(x) . forall x: C . not def g(x);
process
  tcs4: A, C; tcs4 = f(a) -> SKIP || g(c) -> SKIP
```

Output:

```
[gimbo@mane Hets] ./hets -o pp.tex CspCASL/test/tcs4.cspcasl
logic CspCASL
Analyzing spec tcs4
```

Discussion: This produces the output file `CspCASL/test/tcs4.cspcasl.pp.tex`, with the following contents (long times truncated):

```
\begin{hetcasl}
\KW{library} \SId{library}\\
\\
\KW{logic} \SId{CspCASL}\\
\\
\SPEC \=\SIdIndex{tcs4} \Ax{=}\\
\> \KW{data} \=\SORTS \=\Id{A}, \Id{B}, \Id{C} \Ax{<} \Id{S}\\
\>\> \OPS \=\IdDeclLabel{\Id{a}}{a} \Ax{:} \Id{A};\\
\>\>\> \IdDeclLabel{\Id{b1}}{b1}, \IdDeclLabel{\Id{b2}}{b2} \Ax{:}  ...
\>\>\> \IdDeclLabel{\Id{c}}{c} \Ax{:} \Id{C};\\
\>\>\> \IdDeclLabel{\Id{f}}{f} \Ax{:} \=\Id{A} \Ax{\rightarrow?} \I ...
\>\>\> \IdDeclLabel{\Id{g}}{g} \Ax{:} \=\Id{C} \Ax{\rightarrow?} \I ...
\>\> \Ax{\bullet} \=\IdApplLabel{\Id{a}}{a} \Ax{=} \IdApplLabel{\Id ...
\>\> \Ax{\bullet} \=\IdApplLabel{\Id{b2}}{b2} \Ax{=} \IdApplLabel{\ ...
\>\> \Ax{\bullet} \=\Ax{\forall} \Id{x} \Ax{:} \Id{A} \Ax{\bullet}  ...
\>\> \Ax{\bullet} \=\Ax{\forall} \Id{x} \Ax{:} \Id{C} \Ax{\bullet}  ...
\> \KW{process} \=\Id{tcs4} \Ax{:} \=\Id{A}, \Id{C} ;\\
\>\> \Id{tcs4} \Ax{=} \IdApplLabel{\Id{f}}{f}(\IdApplLabel{\Id{a}}{ ...
\KW{end}
\end{hetcasl}
```

which renders as follows (the version in chapter 4 has been post-processed by hand):

library LIBRARY

logic CSPCASL

spec TCS4 =
  data sorts $A, B, C < S$
    ops $a : A$;
      $b1, b2 : B$;
      $c : C$;
      $f : A \rightarrow ? A$;
      $g : C \rightarrow ? C$
    • $a = b1$
    • $b2 = c$
    • $\forall x : A \bullet \neg\, def\, f(x)$
    • $\forall x : C \bullet \neg\, def\, g(x)$
  process $tcs4 : A, C$;
      $tcs4 = f(a) \rightarrow SKIP \;||\; g(c) \rightarrow SKIP$
end

13.2.5 tcs1.het.cspcasl — separation of process and data parts

This example demonstrates handling an explicitly heterogeneous specification, namely the heterogeneous version of `tcs1.cspcasl` (see section 5.4).

Input:

```
spec D =
    sort S, T
    ops c: S; d: T
logic CspCASL
spec tcs1 =
  data D
  process
    tcs1: S, T;
    tcs1 = c -> SKIP || d  -> SKIP
```

Output:

```
[gimbo@mane Hets] ./hets CspCASL/test/tcs1.het.cspcasl
logic CASL
Analyzing spec D
logic CspCASL
Analyzing spec tcs1
```

Discussion: Note that output reports first analysing specification `D` in logic `CASL` (the default), then analyses specification `tcs1` in logic `CspCASL`.

13.3 Examples demonstrating particular features

In this section we demonstrate further features of the tool, paying particular attention to violations of the syntax and static semantics formalised in chapters 9 to 11.

13.3.1 Parse error: English text instead of CSP-CASL

Input:

```
logic CspCASL
spec tcs1 =
  data sort S, T
       ops c: S; d: T
  process
    Just some random text.
```

Output:

```
[gimbo@mane Hets] ./hets CspCASL/test/parse_err1.cspcasl
*** Error,
CspCASL/test/parse_err1.cspcasl:6.10:
unexpected "s"
expecting "(", "%", "=e=" or "="
```

Discussion: As we would hope, text utterly unrelated to CSP-CASL is detected as such. Since `Just` is a valid process name, the error occurs at the start of `some`: we would expect either an open parenthesis (indicating the start of an argument list), a colon (for a process declaration),

or an equals sign (for a process equation). In this regard, the error message is less helpful than we would like, as it does not indicate that a colon might be expected here. This is an area for future improvement — nonetheless, the exact location of the error is correctly identified as line 6, column 10.

13.3.2 Parse error: forgotten semicolon separator in channel list

Input:

```
logic CspCASL
spec tcs1 =
  data sort S, T
       ops c: S; d: T
  channels
    t : S
    x : S
  process
    tcs1: S, T;
    tcs1 = c -> SKIP || d  -> SKIP
```

Output:

```
[gimbo@mane Hets] ./hets CspCASL/test/parse_err2.cspcasl
*** Error,
CspCASL/test/parse_err2.cspcasl:7.5:
unexpected "x"
expecting "[", "%", ";" or "process"
```

Discussion: With a forgotten semicolon between the two channel declarations, the parser expects either for the process part to begin, or for there to be an annotation (beginning with `%`, or for the last channel sort seen (i.e. `T`) to be incomplete (e.g.`T[a]` is a valid CASL sort name).

13.3.3 Parse error: forgotten semicolon terminator after process declaration

Input:

```
logic CspCASL
spec tcs1 =
  data sort S, T
       ops c: S; d: T
  process
    tcs1: S, T
    tcs1 = c -> SKIP || d  -> SKIP
```

Output:

```
[gimbo@mane Hets] ./hets CspCASL/test/parse_err3.cspcasl
*** Error,
CspCASL/test/parse_err3.cspcasl:6.9:
unexpected ":"
expecting casl char, "_", "(", "%", "=e=" or "="
```

Discussion: Again, the error is detected, but the message is less than ideal: it reports that the `:` after `tcs1` is unexpected, which is strange given that the actual error occurs afterwards. The reason may be understood with reference to the discussion in section 12.2.2.2: the `procDecl`

parser would parse `tcs1 : S, T` successfully, but upon encountering the next token, `tcs1`, would fail; at this point, thanks to the `try` combinator, input is rewound and the `procEq` parser reads `tcs1`, failing at the colon.

In general, formation of error messages is a difficult problem in the context of parsing: we would like error messages which tell us where the error is, but often an error isn't recognised at that location; this is certainly an area where future work could be highly beneficial to users of the tool.

13.3.4 Parse error: unexpected symbol — with a good error message

This example is a variant of `tcs3.cspcasl`, above, where the event set in the generalised parallel operator has been replaced with a synchronous parallel symbol.

Input:

```
logic CspCASL
spec tcs3 =
data sorts S, T
     ops f: S ->? T
     . forall x: S . not def f(x);
process
  tcs3: S, T;
  tcs3 = [] x :: S -> f(x) -> SKIP [| || |]
         [] y :: T -> (if def y then SKIP else STOP)
```

Output:

```
[gimbo@mane Hets] ./hets CspCASL/test/paErAlpha.cspcasl
*** Error,
CspCASL/test/paErAlpha.cspcasl:8.39:
unexpected "|"
expecting "%", communication type or "|]"
```

Discussion: here the error is much clearer: an unexpected `|` where we expected an annotation, a communication type (a sort or channel name), or the closure of the generalised parallel operator. 'Communication type' is added to this list by Parsec's `<?>` combinator; the relevant code fragment is:

```
<|> do asKey genpar_openS
       es <- event_set <?> "communication type"
       asKey genpar_closeS
       rp <- choice_proc
       p <- par_proc' (GeneralisedParallel lp es rp (compRange lp rp))
```

13.3.5 Static analysis of channel and process declarations

This example demonstrates the implementation of all of the requirements on channel declarations and process declarations stated in sections 10.3.2 and 10.3.4.

Input:

```
logic CspCASL
spec saErDecls =
  data sort S,T
  channels
    x : U;
    y : S; y : T; y: S;
    w : S;
    S : T
  process
    P: S, T;
    P: S, T;
    Q(U): U;
```

Output:

```
[gimbo@mane Hets] ./hets -v3 CspCASL/test/saErDecls.cspcasl
Options:  --verbose=3 --hets-libdir=../tools/Hets-lib --casl-amalg=cell
    CspCASL/test/saErDecls.cspcasl
Processing input: CspCASL/test/saErDecls.cspcasl
Reading file CspCASL/test/saErDecls.cspcasl
logic CspCASL
Analyzing spec saErDecls
*** Error CspCASL/test/saErDecls.cspcasl:5.9,
unknown sort 'U'
*** Error CspCASL/test/saErDecls.cspcasl:6.12,
channel declared with multiple sorts 'y'
### Warning CspCASL/test/saErDecls.cspcasl:6.19,
channel redeclared with same sort 'y'
*** Error CspCASL/test/saErDecls.cspcasl:8.5,
channel name already in use as a sort name 'S'
*** Error CspCASL/test/saErDecls.cspcasl:11.5,
process name declared more than once 'P'
*** Error CspCASL/test/saErDecls.cspcasl:12.7,
unknown sort 'U'
*** Error CspCASL/test/saErDecls.cspcasl:12.11,
not a sort or channel name 'U'
hets: user error (Stopped due to errors)
```

Discussion: Note the use of the `-v3` in order to display warnings.

- `CHAN-TYPE` rule: a channel's type must be a known sort; violated by `U` at (5,9).
- `CHAN-NAMES` rule: a channel must not be declared multiple times with different sorts; violated by `y : T` at (6,12).
- `CHAN-NAMES` rule: raise a warning for multiple declaration of the same channel with the same sort; triggered by `y : S` at (6,19).
- `CHAN-NAME` rule: a channel's name must not be an already known sort name; violated by `S : T` at (8,5).
- `PROC-DECL` rule: a process must not be declared more than once; violated by `P` at (11,5).
- `PROC-ARGS` rule: every sort in a process declaration's parameter sort list must be known; violated by `U` at (12,7).

- PROC-ALPHA and COMM-TYPE rules: every symbol in a process declaration's permitted alphabet must be a known sort or channel name; violated by U at (12, 11).

13.3.6 Static analysis of process equations

This example demonstrates the implementation of some of the requirements on process equations stated in section 10.3.5.

Input:

```
logic CspCASL
spec saErProcEqs =
  data sort S, T
       ops x: S
  process
    P: S;
    P = RUN(T)
    Q(x) = SKIP
    Z(S): S;
    Z(n, y) = x -> SKIP
```

Output:

```
[gimbo@mane Hets] ./hets CspCASL/test/saErProcEqs.cspcasl
logic CspCASL
Analyzing spec saErProcEqs
*** Error CspCASL/test/saErProcEqs.cspcasl:7.9-7.14,
Communication alphabet subset violations (process equation): [T]
'RUN ( T )'
*** Error CspCASL/test/saErProcEqs.cspcasl:8.5,
process equation for unknown process 'Q'
*** Error CspCASL/test/saErProcEqs.cspcasl:10.5,
too many process arguments 'Z'
hets: user error (Stopped due to errors)
```

Discussion:

- PROC-EQ rule: the subsort closure of the equation's process term must be a subset of the subsort closure of the process' permitted alphabet; violated by the communication alphabet [T], generated by the process term RUN(T) (7,9 to 7,14).
- PARM-PROCNAME rule: the process name in a parametrised process name must have been declared already; violated by process name Q (8,5).
- PROC-VARS rule: the number of variable names in the parametrised process name must equal the number of parameter sorts in the process' declaration; violated by the process named Z at (10,5).

13.3.7 Static analysis of process terms

This example demonstrates the implementation of some of the requirements on process terms stated in section 10.4.

Input:

```
logic CspCASL
spec saErProcEqs =
  data sort S, T
       ops y: T
  process
    P:S;
    Q(S): S;
    P = Q(y)
    Q(x) = x -> RUN(U) || [] t :: U -> t -> SKIP
    Z:S;
    Z = [] t :: S -> SKIP ; t -> SKIP
    W:S;
    W = SKIP [ S || ] SKIP
```

Output:

```
[gimbo@mane Hets] ./hets CspCASL/test/saErProcTerms.cspcasl
logic CspCASL
Analyzing spec saErProcEqs
*** Error CspCASL/test/saErProcTerms.cspcasl:8.11,
can't cast term to sort S '(op y : T)'
*** Error CspCASL/test/saErProcTerms.cspcasl:9.21,
not a sort or channel name 'U'
*** Error CspCASL/test/saErProcTerms.cspcasl:9.35,
unknown sort 'U'
*** Error CspCASL/test/saErProcTerms.cspcasl:11.29,
no operation with 0 arguments found for 't'
*** Error CspCASL/test/saErProcTerms.cspcasl:13.9-13.23,
Communication alphabet subset violations (alphabetised parallel, left): [S]
'SKIP [ S || ] SKIP'
hets: user error (Stopped due to errors)
```

Discussion:

- NAMED-PROC rule: every parameter in a named process reference must be castable to a term of the sort specified by the corresponding slot in the named process' parameter sort list; violated by y at (8,11), which we attempt to cast from T to S, failing as they are not in a subsort relation with each other.

- RUN, EVENT-SET and COMM-TYPE rules: every symbol in an event set must be a known sort or channel name; violated by U at (9,21).

- EXTPRE-PROC and SVAR-DECL rules: the sort referenced in a single variable declaration must already be known; violated by U at (9,35).

- SEQ-PROC rule: the second process in a sequential composition receives an empty local variable set; demonstrated by non-recognition of t at (11,29).

- ALPHAPAR-PROC rule: the subsort closure of an event set in an alphabetised parallel process must be a subset of the subsort closure of the corresponding process term; violated by event set [S] on left hand side of alphabetised parallel process SKIP [S ||] SKIP at (13,9 to 13,23).

13.3.8 Static analysis of local top elements

These two examples demonstrate the implementation of the checks for local top elements described in chapter 11 and triggered by the CSP-BASIC-SPEC rule (section 10.3.1).

First, the simplest possible example without local top elements:

Input:

```
logic CspCASL
spec saErProcEqs =
  data sort a < b ; a <c
  process
    P:;
    P = SKIP
```

Output:

```
[gimbo@mane Hets] ./hets CspCASL/test/saErLocalTops.cspcasl
logic CspCASL
Analyzing spec saErProcEqs
*** Error CspCASL/test/saErLocalTops.cspcasl:4.3-7.1,
local top element obligation (a<c,b) unfulfilled ''
hets: user error (Stopped due to errors)
```

Discussion: analysis has revealed that a is a subsort of both b and c, i.e. there is an obligation ${}^{b}a^{c}$; sadly, the obligation is unfulfilled, so further processing cannot take place.

In our second example, the obligation ${}^{b}a^{c}$ is met by the presence of a local top element, d — a supersort of both b and c:

Input:

```
logic CspCASL
spec saErProcEqs =
  data sort a < b ; a < c ; b < d ; c < d
  process
    P:;
    P = SKIP
```

Output:

```
[gimbo@mane Hets] ./hets CspCASL/test/saOKLocalTops.cspcasl
logic CspCASL
Analyzing spec saErProcEqs
```

13.4 Example from EP2

In this section we present a major example from an industrial case study ([GRS05], on the EP2 banking system [EP202]). The CSP-CASL specification shown in figures 13.4 (data part) and 13.4 (process part) specifies the communications which occur between two components of the system, the *Terminal* and *Acquirer*, in order to initialise the Terminal. The details of the specification are not relevant here, but it is interesting to note that while this specification has been presented (in some form) twice already, in [GRS05] and [OIR07] (which included a proof of its deadlock freedom), this is the first time this specification has been parsed by a tool.

This is certainly the largest example processed by the tool thus far; doing so was instructive, and led to the following observations:

- In the process part of the version published in [OIR07], the reference to the data part by specification name was incorrect due to a spelling error; this was not noticed before tool processing.
- The version published in [OIR07] was missing the `else STOP` at the end of the large if / then / else cascade; CSP-CASL as described in this thesis requires both the `then` and the `else` part, in order to avoid dangling-else problems, so the tool also rejected this.
- The version published in [OIR07] had an error in its CASL syntax. The condition in each of the conditional processes is a CASL formula which checks sort membership; in the LaTeX output and in [OIR07], they appear as `x : S`; however, the correct input format is in fact `x in S`.
- The concrete syntax has developed in various forms since [OIR07] was written, as reflected in this thesis; for example, we use `x : : s` rather than `x : s` in channel receive processes — thus some updates were required there.
- Finally, in order for the specification to parse correctly, it proved necessary to parenthesise the entirety of the outermost conditional process; upon investigation, this turns out to be an issue with the encoding of precedence in the grammar. As such, the tool (which correctly implements that grammar) has been useful in drawing attention to the problem, whose resolution remains future work.

library LIBRARY

logic CASL

spec D_ACL_GETINITINITIALISATION =

sorts *D_SI_Init_SessionStart*,
D_SI_Init_SessionEnd, *D_SI_Init_ConfigDataRequest*, *D_SI_Init_ConfigDataResponse*,
D_SI_Init_ConfigDataNotification, *D_SI_Init_ConfigDataAcknowledge*,
D_SI_Init_RemoveConfigDataNotification, *D_SI_Init_RemoveConfigDataAcknowledge*,
D_SI_Init_ActivateConfigDataNotification, *D_SI_Init_ActivateConfigDataAcknowledge*
$< D\_SI\_Init$

$\forall x : D\_SI\_Init\_SessionEnd; y : D\_SI\_Init\_ConfigDataRequest$
$\bullet \neg x = y$

$\forall x : D\_SI\_Init\_SessionEnd; y : D\_SI\_Init\_ConfigDataNotification$
$\bullet \neg x = y$

$\forall x : D\_SI\_Init\_SessionEnd; y : D\_SI\_Init\_RemoveConfigDataNotification$
$\bullet \neg x = y$

$\forall x : D\_SI\_Init\_SessionEnd; y : D\_SI\_Init\_ActivateConfigDataNotification$
$\bullet \neg x = y$

$\forall x : D\_SI\_Init\_ConfigDataRequest; y : D\_SI\_Init\_ConfigDataNotification$
$\bullet \neg x = y$

$\forall x : D\_SI\_Init\_ConfigDataRequest; y : D\_SI\_Init\_RemoveConfigDataNotification$
$\bullet \neg x = y$

$\forall x : D\_SI\_Init\_ConfigDataRequest; y : D\_SI\_Init\_ActivateConfigDataNotification$
$\bullet \neg x = y$

$\forall x : D\_SI\_Init\_ConfigDataNotification; y : D\_SI\_Init\_RemoveConfigDataNotification$
$\bullet \neg x = y$

$\forall x : D\_SI\_Init\_ConfigDataNotification; y : D\_SI\_Init\_ActivateConfigDataNotification$
$\bullet \neg x = y$

$\forall x : D\_SI\_Init\_RemoveConfigDataNotification; y : D\_SI\_Init\_ActivateConfigDataNotification$
$\bullet \neg x = y$

ops *seM* : *D_SI_Init_SessionEnd*;
cdrM : *D_SI_Init_ConfigDataRequest*;
cdnM : *D_SI_Init_ConfigDataNotification*;
rcdnM : *D_SI_Init_RemoveConfigDataNotification*;
acdnM : *D_SI_Init_ActivateConfigDataNotification*

end

Figure 13.2: EP2 example: data part

```
logic CSPCASL

spec GETINITIALISATIONDATA =
    data D_ACL_GETINITINITIALISATION
    channel C_SI_Init : D_SI_Init
    process Ter_Init : C_SI_Init ;
            Ter_ConfigurationManagement : C_SI_Init ;
            Acq_Init : C_SI_Init ;
            Acq_ConfigurationManagement : C_SI_Init ;
            Ter_Init = C_SI_Init ! sessionStart :: D_SI_Init_SessionStart →
            Ter_ConfigurationManagement
            Ter_ConfigurationManagement = C_SI_Init ? configMess :: D_SI_Init →
              if (configMess in D_SI_SessionEnd)
                 then SKIP
                 else if (configMess in D_SI_Init_ConfigDataRequest)
                         then C_SI_Init ! response :: D_SI_Init_ConfigDataResponse →
                                                      Ter_ConfigurationManagement
                         else if (configMess in D_SI_Init_ConfigDataNotification)
                                 then C_SI_Init ! acknowledge ::
                                                  D_SI_Init_ConfigDataAcknowledge →
                                                  Ter_ConfigurationManagement
                                 else if (configMess in D_SI_Init_RemoveConfigDataNotification)
                                         then C_SI_Init ! acknowledge ::
                                                          D_SI_Init_RemoveConfigDataAcknowledge →
                                                          Ter_ConfigurationManagement
                                         else if (configMess in D_SI_Init_ActivateConfigDataNotification)
                                                 then C_SI_Init ! acknowledge ::
                                                      D_SI_Init_ActivateConfigDataAcknowledge →
                                                      Ter_ConfigurationManagement
                                                      else STOP
            Acq_Init = C_SI_Init ? sessionStart :: D_SI_Init_SessionStart →
                                                   Acq_ConfigurationManagement
            Acq_ConfigurationManagement =
                C_SI_Init ! seM → SKIP
                □ C_SI_Init ! cdrM → C_SI_Init ? response ::
                          D_SI_Init_ConfigDataResponse → Acq_ConfigurationManagement
                □ C_SI_Init ! cdnM → C_SI_Init ? acknowledge ::
                          D_SI_Init_ConfigDataAcknowledge → Acq_ConfigurationManagement
                □ C_SI_Init ! rcdnM → C_SI_Init ? acknowledge ::
                          D_SI_Init_RemoveConfigDataAcknowledge →
                          Acq_ConfigurationManagement
                □ C_SI_Init ! acdnM → C_SI_Init ? acknowledge ::
                          D_SI_Init_ActivateConfigDataAcknowledge →
                          Acq_ConfigurationManagement
            System : C_SI_Init ;
            System = Acq_Init ||[ C_SI_Init ]|| Ter_Init
end
```

Figure 13.3: EP2 example: process part

Chapter 14

Conclusion

Contents

14.1 Summary

In this thesis we have given an account of the design, formalisation, and implementation of syntactic and static semantic rules for the specification language CSP-CASL. After a background section in which we surveyed CSP-CASL's context and a number of related matters, we summarised the language, informally discussing all of its syntactic and static semantic features, and provided examples. We then formalised its abstract and concrete syntax as context-free grammars, and formalised its static semantics using the Natural Semantics formalism. Following this, we formulated the problem of checking a CSP-CASL specification for local top elements, and presented an algorithm for solving this problem, and an implementation in the programming language Haskell. We then described the implementation of a tool, based on those formalisations, to check a CSP-CASL specification's syntax and static properties; the tool was written in Haskell as an extension to the toolset HETS. Finally, we demonstrated the tool in use, on a number of small examples, and a large example from an industrial case study.

This work advances the field of formal specification by providing initial tool support for a new specification language, in a framework which can, in time, be extended to provide full automated and interactive theorem proving on specifications written in that language. Furthermore, this thesis provides the only formal static semantics of a CSP dialect of which we are aware, and also the first formal treatment of the issue of local and global variables in a CSP dialect. By formalising these aspects of CSP-CASL, we advance the programme of CoFI, the Common Framework Initiative, by contributing to the development of another candidate CASL extension language for its family. Finally, this work provides a further successful case study of the use of Parsec for recursive descent combinator parsing, and of the extensibility of the HETS architecture.

14.2 Availability

The work described in this thesis is part of the standard HETS distribution, available from:

`http://www.informatik.uni-bremen.de/cofi/hets/`

HETS has fairly demanding requirements for installation, in terms of supporting software. Fortunately, it is easy to experiment with the parsing and static analysis parts of HETS via an online interface, at:

`http://www.informatik.uni-bremen.de/cgi-bin/cgiwrap/maeder/hets.cgi`

All of the examples in this thesis should be usable at that web page (possibly with the addition of a preceding '`logic CspCASL`' directive). An archive of the examples is downloadable from:

`http://www.cs.swan.ac.uk/~csandy/mphil/examples.tar`

14.3 Future work

There is always room for improvement. As this work has progressed, it has given rise to an astonishing number of ideas for possible extensions and modifications — 'wishlist features' with which to enhance the specification experience. We have also at several points drawn attention to shortcomings or problems with the language — though nothing major, we believe: some 'rough edges' which could be smoothed out. Time and usage will demonstrate which of the wishlist features are truly desirable; until then, we briefly survey the areas of possible future work.

14.3.1 Automated testing

As noted in section 12.5, at an earlier stage in development we found an automated testing framework highly beneficial with respect to grammar design and parser implementation; as such, it would be very desirable to reinstate such a framework, as described in that section — indeed, this could, if done properly, be of great benefit not only to CSP-CASL, but to HETS in general.

14.3.2 Improve error messages

As noted in section 13.3.3, error messages indicating parse errors are in some circumstances misleading in their reports of the location of the error. Production of useful error messages from parsers is well-known to be a difficult problem: "the error handler in a parser has goals that are simple to state but challenging to realize" [ALSU06] — and is in fact complicated by our current approach of encoding the grammar as directly as possible using Parsec's combinators.

As hinted at in section 13.3.3, the first step in improving this situation would be to apply standard grammar transformations to minimise the lookahead required at any point in our grammar; this would tend to localise error messages to the actual point of error, but at the expense of having a more complicated grammar, with a less obvious correspondence between abstract and

concrete syntax. Nonetheless, now that the formalisation of CSP-CASL has progressed past this initial stage, it is probably a price worth paying, for the sake of the users of the language.

14.3.3 Precedence encoding issue / conditional process

Related to the previous point, and as reported in section 13.4, there is a problem involving the encoding of precedences within our concrete grammar. This problem does not prevent the tool being used, and an informed and intelligent user can easily work out the solution of adding parentheses where we would not strictly expect to require them — but it is certainly a 'wart' in the design which we would like to eliminate. Resolution of this issue will involve careful analysis of the concrete grammar, preferably with the aid of an appropriate tool.

14.3.4 Universe of communications

In CSP it is conventional to refer to 'the entire alphabet' as Σ (see section 3.2.1), so one can easily write, e.g.

$$P \,[\![\, \Sigma \,]\!]\, Q$$

In CSP-CASL there is currently no easy way to do this: one must explicitly list all sort and channel names one wishes to use; thus, we suggest, as a simple extension to the language, a way to refer to 'the Universe of all possible communications' with some distinguished identifer, e.g. *Univ*; then we might write:

$$P \,[\![\, \mathit{Univ} \,]\!]\, Q$$

14.3.5 Process definitions

A convenient — and straightforward to implement — syntactic sugaring would be process *definitions*, which unite declaration and equation. For example, rather than the following:

logic CSPCASL
spec PROCDEFN =
  data sorts S, T
  process $P(S, T) : S, T$;
    $P(a, b) = a \rightarrow SKIP \,\Box\, b \rightarrow SKIP$
end

we might instead write:

logic CSPCASL
spec PROCDEFN $\doteq$
  data sorts S, T
  process $P(a : S, b : T) : S, T = a \rightarrow SKIP \,\Box\, b \rightarrow SKIP$
end

14.3.6 No explicit channel declarations section

For largely historical reasons, our current design demands that channels are declared in a special section preceding the specification's processes: earlier works on CSP-CASL followed this form, and it seemed reasonable to do the same here. However, there is no strong reason for doing so; in the interests of flexibility we propose relaxing this constraint, so that channel declarations may be mixed with process declarations and process equations freely (albeit subject to linear visibility, until such a time that that constraint is lifted).

14.3.7 Parametrised process operators/replicated forms

All of the CSP-CASL process operators presented in this thesis are unary or binary, i.e. they either modify a single process, or combine two processes to form a new one. However, in CSP we have *replicated forms* of certain operators, i.e. *parametrised* process operators; adding these to CSP-CASL would provide a useful language feature.

For example, the following:

$$\prod_{i:Nat=1}^{10} P(i)$$

might represent $P(1) \parallel P(2) \parallel \cdots \parallel P(10)$.

An initial version might be restricted to ranging over natural numbers (as in the above example), and imply importing the appropriate CASL standard library. It would, however, be natural to frame such operators so that rather than being limited to ranging over the integers or natural numbers (say), they were able to range over arbitrary CASL sorts having the right properties; in particular, free types would make good candidates in this setting. Then there are related questions of loop control, e.g. step size, direction, etc. — such aspects might be specified using CASL-style annotations, for example.

14.3.8 Pattern matching

CSP-CASL process names may be parametrised; this immediately leads to the observation that pattern matching on process equations, in the style of Haskell, provides a very useful syntactic sugaring. Again, how to achieve this in a general form over CASL sorts is unclear at this stage, but again, free types seem at least tractable. For example, rather than write:

logic CSPCASL
spec PATTMATCH =
    data free type *Nat* ::= 0 | *succ*(*Nat*)
    process *Counter*(*Nat*) : *Nat* ;
        Counter(*n*) = **if** *n* = 0 **then** 0 → *Counter* (0) **else** *succ*(*n*) → *Counter* (*n*)
end

we might then instead write:

logic CSPCASL
spec PATTMATCH =
    data free type *Nat* ::= 0 | *succ*(*Nat*)
    process *Counter*(*Nat*) : *Nat* ;

$Counter(0) = 0 \rightarrow Counter\ (\ 0\ )$
$Counter(succ(n)) = succ(n) \rightarrow Counter\ (\ n\ )$

end

14.3.9 Let-expressions

As described in section 8.3.5, a potentially interesting future extension to CSP-CASL would be the addition of let-expressions at the process operator level, providing a fully generalised process equation scoping mechanism similar to that seen for functions in Haskell (say).

14.3.10 Inference of process parameter sorts and process alphabets

Rather than require processes to be declared before use, we might allow process equations without corresponding declarations (see section 8.3.4). This requires *inference* of each process' parameter sorts and alphabet.

In [Ros98, §2.7], Roscoe notes that the version of CSP presented in that work does not assign particular alphabets to each process, and this differs from Hoare's original treatment in [Hoa85] — our approach, then, follows Hoare rather than Roscoe. Roscoe argues that the disadvantages of 'the alphabetized version of CSP' are the necessity to provide all alphabets (cluttering definitions), and additional theoretical complexity. In this thesis we have, we hope, addressed at least some of the theoretical issues (model semantics is addressed elsewhere, e.g. [Rog06]). To Roscoe's first criticism, however, we must concede: requiring process declarations certainly adds complexity and length to CSP-CASL specifications; unified process definitions, as described in section 14.3.5, would go some way to alleviating this; even so, a version of CSP-CASL which minimises the necessity for declaration is clearly desirable.

14.4 Evaluation

On the whole, this work has been a success. Whereas previously CSP-CASL was a 'blackboard' language, presented in several works and with a well-worked out mathematical basis, it lacked both a fixed formal syntax, and tool support. The aim of this work was to perform language design tasks leading to not only a fixed and formalised syntax for CSP-CASL, but also a fixed and formalised static semantics, and an implementation of those formalisations in an appropriate tool. All of these aims have been achieved, and for the first time it is now possible to check CSP-CASL specifications for syntactic and static semantic errors; furthermore, there now exists a single, and thorough, point of reference for those working with the language, or extending and adapting it in the future. As such, we consider the project to have been successful.

We have demonstrated the tool in this thesis, and for the specifications processed so far, the tool and the language both seem to us to work well. We anticipate that now that tool support is finally available, more specifications will be written in CSP-CASL, not for the purpose of testing the tool, but for the purpose of specifying things; we fully anticipate that this will lead to extremely interesting and valuable feedback, insight into our language design and how it might be improved and — alas — no doubt a few bug reports. This will be the true test of the work presented here.

Bibliography

[AJS05] Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors. *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004, Revised Invited Papers*, volume 3525 of *Lecture Notes in Computer Science*. Springer, 2005.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[AS02] David Aspinall and Donald Sannella. From Specifications to Code in CASL. In *AMAST '02: Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, pages 1–14, London, UK, 2002. Springer-Verlag.

[Bac78] John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[Bae05] J. C. M. Baeten. A Brief History of Process Algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987.

[BCH+04] Hubert Baumeister, Maura Cerioli, Anne Haxthausen, Till Mossakowski, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL semantics. In CASL Reference Manual [CoF04c], part III. Edited by D. Sannella and A. Tarlecki.

[BM04] Michel Bidoit and Peter D. Mosses. CASL *User Manual.* LNCS Vol. 2900 (IFIP Series). Springer, 2004. With chapters by Till Mossakowski, Donald Sannella, and Andrzej Tarlecki.

[BPS98] Bettina Buth, Jan Peleska, and Hui Shi. Combining methods for the livelock analysis of a fault-tolerant system. In *AMAST'98*, LNCS 1548, pages 124–139. Springer, 1998.

[Bri88] Ed Brinksma. *On the Design of Extended* LOTOS *– A Specification Language for Open Distributed Systems.* PhD thesis, Department of Informatics, University of Twente, Enschede, Netherlands, 1988.

[BS99] Bettina Buth and Mike Schrönen. Model-checking the architectural design of a fail-safe communication system for railway interlocking systems. In *FM'99*, LNCS 1709. Springer, 1999.

[BW88] Richard Bird and Philip Wadler. *An Introduction to Functional Programming.* Prentice Hall, 1988.

[BW90] J. C. M. Beaten and W. P. Weijland. *Process Algebra.* Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[CEW93] Ingo Classen, Hartmut Ehrig, and Dietmar Wolz. *Algebraic specification techniques and tools for software development: the ACT approach.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1993.

[CoF04a] CoFI Language Design Group. CASL summary. In CASL Reference Manual [CoF04c], part I. Edited by B. Krieg-Brückner and P. D. Mosses.

[CoF04b] CoFI Language Design Group. CASL syntax. In CASL Reference Manual [CoF04c], part II. Edited by B. Krieg-Brückner and P. D.Mosses.

[CoF04c] CoFI (The Common Framework Initiative). CASL *Reference Manual.* LNCS Vol. 2960 (IFIP Series). Springer, 2004.

[Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[EP202] *eft/pos 2000 Specification, version 1.0.1.* EP2 Consortium, 2002.

[FL08] Marc Fontaine and Michael Leuschel. Typechecking CSP Specifications using Haskell. In *AVOCS 2007: Seventh International Workshop on Automated Verification of Critical Systems*, proceedings to appear in Formal Aspects of Computing. Springer, 2008.

[Fok95] Jeroen Fokker. Functional Parsers. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 1–23. Springer, 1995.

[FSE03] *Process Behaviour Explorer — the ProBE User Manual.* Formal Systems (Europe) Ltd., 2003.

[FSE06] *Failures-Divergence Refinement — the FDR2 User Manual.* Formal Systems (Europe) Ltd., 2006.

[GB92] Joseph A. Goguen and Rod M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. *Journal of the ACM*, 39(1):95–146, 1992.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer, 1995.

[GRS05] Andy Gimblett, Markus Roggenbach, and Bernd-Holger Schlingloff. Towards a Formal Specification of an Electronic Payment System in CSP-CASL. In J. L. Fiadeiro, P. Mosses, and F. Orejas, editors, *Recent Trends in Algebraic Development Techniques, 17th International Workshop, WADT 2004, Selected Papers*, LNCS Vol. 3423, pages 61–78. Springer, 2005.

[GWM+93] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ.* Cambridge University Press, 1993.

[HB99] Michael G. Hinchey and J. P. Bowen. *High-Integrity System Specification and Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[HHJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: Being Lazy With Class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[HM96] Graham Hutton and Erik Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hoa06] C. A. R. Hoare. Why ever CSP? *Electronic Notes in Theoretical Computer Science*, 162:209–215, September 2006.

[Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, page 196, 1996.

[Hug95] John Hughes. The Design of a Pretty-printing Library. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 53–96, London, UK, 1995. Springer-Verlag.

[Hut92] Graham Hutton. Higher-order Functions for Parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[Hyd06] Randall Hyde. The Fallacy of Premature Optimization. *ACM Ubiquity*, 7(24):2–2, 2006.

[Kah87] Gilles Kahn. Natural Semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[KRS07] Temesghen Kahsai, Markus Roggenbach, and Bernd-Holger Schlingloff. Specification-based testing for refinement. In Mike Hinchey and Tiziana Margaria, editors, *Proceedings of SEFM 2007*, pages 237–247. IEEE Computer Society, 2007.

[KST97] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of extended ML: a gentle introduction. *Theoretical Computer Science*, 173(2):445–484, 1997.

[LM01] Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical Report Technical Report UU-CS-2001-35, Universiteit Utrecht, 2001.

[LZL99] Wenjun Li, Xiaocong Zhou, and Shixian Li. The typing of communicating sequential processes. In *TOOLS '99: Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems*, page 61, Washington, DC, USA, 1999. IEEE Computer Society.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[Mil89] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Mil99] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[MML07a] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set. In Michael Huth Orna Grumberg, editor, *TACAS 2007: Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, Proceedings*, LNCS Vol. 4424, pages 519–522. Springer, 2007.

[MML07b] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set. In Bernhard Beckert, editor, *VERIFY 2007, 4th International Verification Workshop*, volume 259 of *CEUR Workshop Proceedings*, pages 119–135. 2007.

[Mos96] Peter D. Mosses. CoFI: The Common Framework Initiative for algebraic specification. *Bulletin of the EATCS*, 59:127–132, June 1996. An updated version is [Mos01].

[Mos01] Peter D. Mosses. CoFI: The common framework initiative for algebraic specification and development. In G. Păun, G. Rozenberg, and A. Salomaa, editors, *Current Trends in Theoretical Computer Science: Entering the 21st Century*, pages 153–163. World Scientific, 2001.

[Mos03] Till Mossakowski. Foundations of heterogeneous specification. In Wirsing et al. [WPH03], pages 359–375.

[Mos04a] Till Mossakowski. HETCASL – Heterogeneous Specification. Language Summary. Technical report, Universitaet Bremen, 2004.

[Mos04b] Till Mossakowski. ModalCASL - Specification with Multi-Modal Logics. Language Summary. Technical report, Universitaet Bremen, 2004.

[Mos05] Till Mossakowski. Heterogeneous specification and the heterogeneous tool set. Habilitation thesis, Universitaet Bremen, 2005.

[MR07] Till Mossakowski and Markus Roggenbach. Structured CSP– A Process Algebra as an Institution. In J. L. Fiadeiro and P. Schobbens, editors, *Recent Trends in Algebraic Development Techniques, 18th International Workshop, WADT 2006, Revised Selected Papers*, LNCS Vol. 4409, pages 92–110. Springer, 2007.

[MR08] Till Mossakowski and Markus Roggenbach. An institution for processes and data. In Andrea Corradini and Fabio Gadducci, editors, *WADT 2008 – Preliminary Proceedings*, Techinal Report: TR-08-15, pages 13–14. Universita Di Pisa, Dipartimento Di Informatica, 2008.

[MRRS03] Till Mossakowski, Horst Reichel, Markus Roggenbach, and Lutz Schröder. Algebraic-coalgebraic specification in COCASL. In Wirsing et al. [WPH03], pages 376–392. Extended version submitted for publication.

[MRS03] Till Mossakowski, Markus Roggenbach, and Lutz Schröder. COCASL at work – modelling process algebra. In H. P. Gumm, editor, *Coalgebraic Methods in Computer Science, CMCS'03, Warsaw, Poland, Proceedings*, ENTCS Vol. 82.1. Elsevier, 2003.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. August 1990.

[MU05] Petra Malik and Mark Utting. Czt: A framework for z tools. In *ZB. Lecture*, pages 65–84. Springer-Verlag, LNCS, 2005.

[OIR07] Liam O'Reilly, Yoshinao Isobe, and Markus Roggenbach. Integrating Theorem Proving for Processes and Data. In Magne Haveraaen, John Power, and Monika Seisenberger, editors, *CALCO Young Researchers Workshop CALCO-jnr 2007 – Abstracts for Presentations*, pages 18–20. University of Bergen, 2007.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[Pey03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report.* Cambridge University Press, Cambridge, England, 2003.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[RMS04] Markus Roggenbach, Till Mossakowski, and Lutz Schröder. CASL libraries. In CASL *Reference Manual*, LNCS Vol. 2960 (IFIP Series), part V. Springer, 2004.

[Rog03] Markus Roggenbach. CSP-CASL: a new Integration of Process Algebra and Algebraic Specification. In F. Spoto, G. Scollo, and A. Nijholt, editors, *Algebraic Methods in Language Processing, AMiLP 2003*, TWLT Vol. 21, pages 229–243. Univ. of Twente, 2003.

[Rog06] Markus Roggenbach. CSP-CASL: a new Integration of Process Algebra and Algebraic Specification. *Theoretical Computer Science*, 354(1):42–71, 2006.

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[RSG+01] Peter Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, and Bill Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach.* Addison-Wesley, 2001.

[Sca98] Bryan Scattergood. The Semantics and Implementation of Machine-Readable CSP, 1998. DPhil thesis, University of Oxford.

[SM02] Lutz Schröder and Till Mossakowski. HASCASL: Towards integrated specification and development of Haskell programs. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methods and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, Proceedings*, LNCS Vol. 2422, pages 99–116. Springer, 2002.

[Spi89] J. M. Spivey. *The Z notation: a reference manual.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Sud05] Thomas A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[vD04] Dirk van Dalen. *Logic and Structure.* Springer, 2004.

[Wad85] Philip Wadler. How to replace failure by a list of successes. In *FPLCA 1985: Functional Programming Languages and Computer Architecture*, LNCS Vol. 201, pages 113–128. Springer, 1985.

[Wad92] Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1992. ACM.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995. Springer-Verlag.

[Wag02] Eric G. Wagner. Algebraic specifications: some old history and new thoughts. *fNordic Journal of Computing*, 9(4):373–404, 2002.

[WC01] J. C. P. Woodcock and A. L. C. Cavalcanti. A concurrent language for refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.

[WC02] J. C. P. Woodcock and A. L. C. Cavalcanti. The semantics of *circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184—203. Springer-Verlag, 2002.

[WD96] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[WPH03] M. Wirsing, D. Pattinson, and R. Hennicker, editors. *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, 2002, Revised Selected Papers*, LNCS Vol. 2755. Springer, 2003.

[XCS06] M. A. Xavier, A. L. C. Cavalcanti, and A. C. A. Sampaio. Type Checking *Circus* Specifications. In A. M. Moreira and L. Ribeiro, editors, *SBMF 2006: Brazilian Symposium on Formal Methods*, pages 105 – 120, 2006.